

Formal Security Analysis of the OpenID FAPI 2.0 Security Profile with FAPI 2.0 Message Signing, FAPI-CIBA, Dynamic Client Registration and Management

TECHNICAL REPORT

Pedram Hosseyni, Ralf Küsters, Tim Würtele

Institute of Information Security – University of Stuttgart, Germany

{pedram.hosseyni, ralf.kuesters, tim.wuertele}@sec.uni-stuttgart.de

I. INTRODUCTION

Building on our recent formal security analysis [24, 25] of the FAPI 2.0 Security Profile [14], we here extend the analysis effort to FAPI 2.0 Message Signing [22],¹ combined with Dynamic Client Registration (DCR, defined in RFC 7591 [42]), Dynamic Client Management (DCM, defined in RFC 7592 [43]), as well as FAPI-CIBA [49].² Overall, we model an ecosystem which uses all these profiles and extensions in parallel.

This report covers both phases of the aforementioned extension, which includes *modeling* FAPI 2.0 Message Signing, DCR, DCM, and FAPI-CIBA within the Web Infrastructure Model (WIM, see [31]), *defining precise security properties* based on that formal model, as well as *proving the formalized properties*.

In what follows, we first give a brief overview of the protocols and extensions contained in the model, as well as the attacker assumptions. This is followed by an overview of the security properties that we analyze. We continue with a brief introduction to the WIM, some remarks on the involved specifications, explanations and discussions of certain modeling decisions and assumptions. In the appendices, we give the full formal model and formalized security properties, and the full formal proof.

II. FAPI 2.0 PROTOCOL AND SECURITY GOALS

Here, we first describe the FAPI 2.0 Security Profile (Second Implementer’s Draft) [14] protocol and the accompanying FAPI 2.0 Attacker Model (Second Implementer’s Draft) [13]. Following this, we introduce DCR, DCM, FAPI 2.0 Message Signing, and FAPI-CIBA. For each of these extensions, we briefly mention the corresponding security goals and refer to Section III for a full overview of the security properties that we analyze.

A. Overview of FAPI 2.0 Security Profile

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., [3, 6, 9, 37]).

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 the client requests access to. *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

¹Our analysis of FAPI 2.0 Message Signing is based on commit 67246ac to the OIDF’s FAPI WG repository, and when we mention section numbers, we refer to that commit. However, we have of course incorporated the changes made to that document since then, up to commit 96c2fec (from August 13, 2023).

²Our analysis of FAPI-CIBA, when used with FAPI 2.0, is based on commit ac96a05 to the OIDF’s FAPI WG repository. At the time of this writing, that commit is not yet merged into the master branch. We use ac96a05 after consultation with the FAPI WG chairs. See <https://bitbucket.org/openid/fapi/pull-requests/417> for tracking of the corresponding pull request.

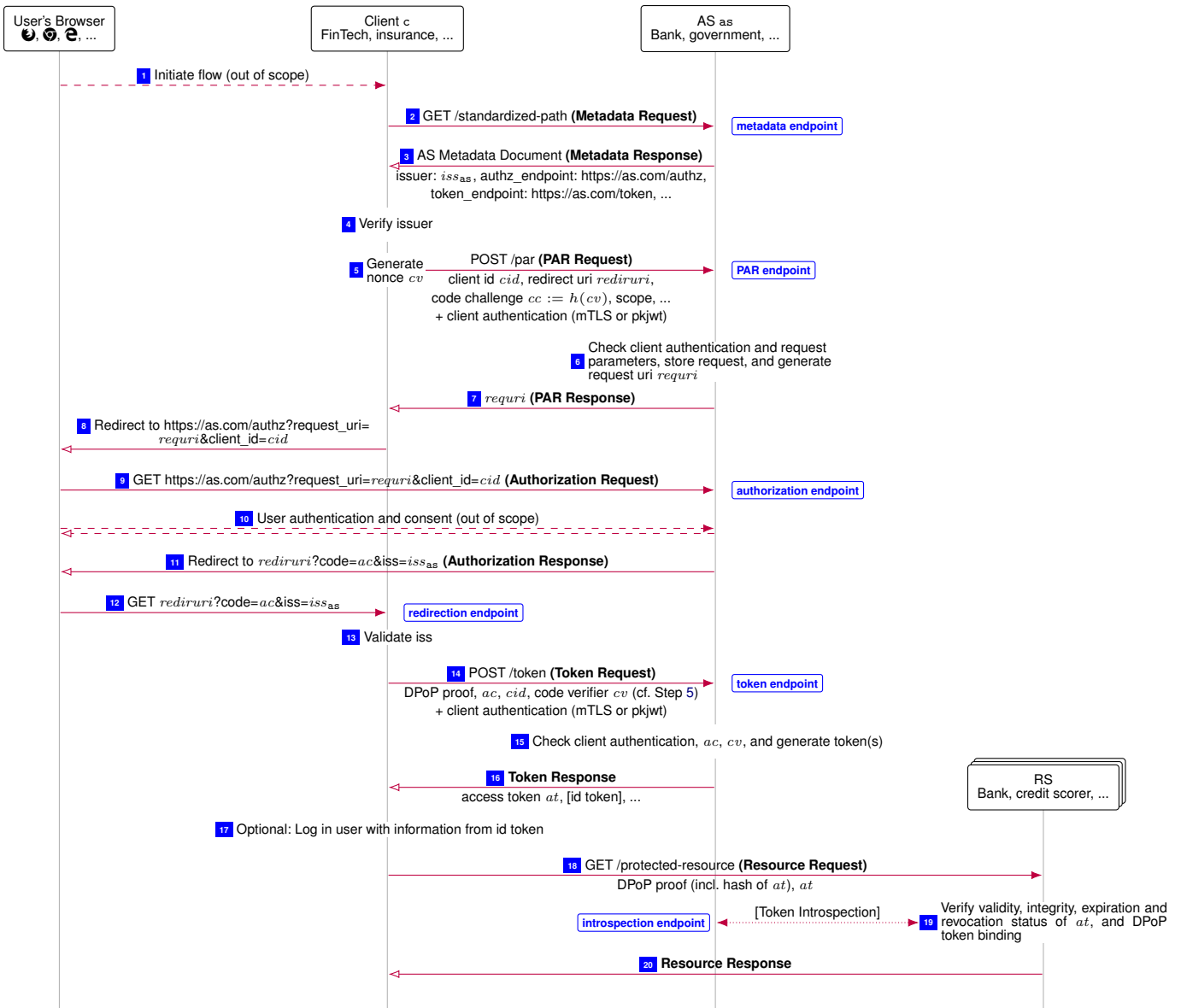


Figure 1. FAPI 2.0 Security Profile protocol flow (with DPoP sender constraining)

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 Security Profile 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 the client using a certain AS *as*, e.g., by clicking a “Login with *as*” button (Step 1).

FAPI 2.0 assumes that the client received the so-called *issuer identifier* iss_{as} of *as* (e.g., via configuration). That issuer identifier is used in FAPI 2.0 and other protocols to uniquely identify *as* [48]. However, to complete a FAPI 2.0 flow, the client needs additional knowledge on *as*, e.g., endpoint URLs. Therefore, it proceeds by fetching so-called *Authorization Server*

Metadata [29, 45] from *as* (Step [2]). 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) [34] 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) [44]. 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 requiri* and stores the requested scope, *cid*, $cc := h(cv)$, *rediruri*, and *requiri* (Step [6]); *requiri* 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 *requiri* and *cid* as request parameters (Step [8]). Following that redirect, the user’s browser visits *as* and in doing so, forwards *requiri* 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 [48] (i.e., the issuer identifier iss_{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 the client sent the PAR request in Step [5] to prevent mix-up attacks [20, 33, 36, 48]. 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 also include information for access token sender constraining, which we describe below.

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 [38]: token introspection [41], and structured access tokens, which contain the necessary information and are typically signed by the AS [5, 26]. 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 mandates ASs to authenticate their 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 [7] authentication, the client presents a TLS certificate containing the client’s identity, e.g., one of its domains, during TLS connection establishment. With *private_key_jwt* [46], the client adds a signed *JSON Web Token (JWT)* [26–28] to its messages. This JWT contains, among other things, 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)* [15], which is shown in Figure 1, and mTLS [7]. 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*, consisting of a signed JWT *dpopJWT*, containing the URL to which it is sent, a nonce, and a public verification key $pub(k)$ (of the client’s choice). *dpopJWT* is signed using the corresponding private key *k*. The AS then binds the access token to $pub(k)$. 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. FAPI 2.0 Attacker Model

Along with the actual protocol specification, the FAPI Working Group (FAPI WG) developed the FAPI 2.0 Attacker Model [13] which outlines security goals and assumptions on attackers under which these goals are expected to hold for FAPI 2.0 Security Profile. We only briefly recall these goals and assumptions here, and refer to our reports on the analysis of FAPI 2.0 Security Profile [24, 25] for a more extensive discussion. Furthermore, there are additional security goals for FAPI 2.0 Message Signing, see Section II-E.

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 note that these assumptions are considered not only for FAPI 2.0 Security Profile but also in the analysis of the extensions presented below.

A1. 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.

A2. The attacker can read authorization requests in plain (cf. Step [9] in Figure 1).

A3. 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.

A4. Resource requests (Step [18]) leak to the attacker in plain.

D. Dynamic Client Registration and Management

The aforementioned relationship between FAPI 2.0 clients and ASs can be established in various ways: in our initial work on the FAPI 2.0 Security Profile [24, 25], we assumed a pre-configured relationship which includes key material, client id, and the client type (i.e., which authentication and token binding mechanisms are used). However, for many real ecosystems, such manual configuration is not feasible (e.g., *Single Page Applications*) or simply too tedious. Hence, the *OAuth 2.0 Dynamic Client Registration Protocol* [42] has been developed which allows clients to register themselves with an AS in a defined way. To further accommodate for changes to the client’s configuration (e.g., key rollover), the *OAuth 2.0 Dynamic Client Registration Management Protocol* allows clients to change their configuration at an AS.

On a high level, an AS supporting dynamic client registration offers an additional endpoint to which clients send JWTs with their desired configuration, to which the AS replies with a similar JWT, containing the actually registered values (which may differ from the ones the client wished to register). The DCR specification also defines two optional security measures: 1) With *initial access tokens*, clients wanting to register themselves with an AS must present such an initial access token at the registration endpoint. The means by which a client obtains such an initial access token are, however, out of scope. And 2) *client assertions* are JWTs with claims about a client, signed by a third party which is trusted by the AS to have verified

these claims before signing the JWT. As mentioned, both mechanisms are optional, and none of the FAPI 2.0 specifications provide further guidance regarding DCR. We therefore modeled DCR without initial access tokens and without client assertions, effectively modeling a completely open ecosystem where everyone can register clients. Note that our security properties are formulated such that they only apply to protocol executions involving honest participants.

Similar to DCR, dynamic client management is realized by an AS endpoint which is very similar to the one for dynamic client registration. However, to ensure clients can only modify their own configuration, this endpoint requires a so-called registration access token, which is issued to the client upon registration (and may be updated during subsequent interactions with the dynamic client management endpoint).

Regarding security, there are no explicit security goals in either specification, so we expect dynamic client registration and management to not introduce additional attacks, i.e., we expect an ecosystem using FAPI 2.0 with dynamic client registration and management to achieve the same level of security as such an ecosystem without dynamic client registration and management (see [Section III](#)).

E. FAPI 2.0 Message Signing

While FAPI 2.0 Security Profile offers a secure profile regarding authorization, authentication, and session integrity, it does not – and does not aim to – offer any accountability properties. Depending on the use case, there may be legal requirements for certain protocol participants to be able to prove that another party sent a certain message. To accommodate such use cases, the FAPI 2.0 family of specifications features a standard called *FAPI 2.0 Message Signing* [22], which aims to add such accountability/non-repudiation guarantees to FAPI 2.0 ecosystems.

Similar to the FAPI 2.0 Security Profile, FAPI 2.0 Message Signing makes use of existing standards, adds some requirements to enhance security and interoperability, but does not introduce new grants, etc., and is somewhat modular to account for different ecosystems’ needs with regard to which messages need to be non-reputable (see [22, Sec. 5.1]). However, all of the defined profiles have in common that they add sender signatures to one or more of the messages sent in a flow of the FAPI 2.0 Security Profile.

The available profiles are:

Signed Authorization Requests For signed authorization requests, the *OAuth 2.0 JWT Secured Authorization Request (JAR)* [47] is employed. In the context of FAPI 2.0 Security Profile, the relevant message to be signed is the pushed authorization request (Step [5] in [Figure 1](#)).

Signed Authorization Responses Signed authorization responses are implemented using the *JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)* [32] (applied to Step [11] in [Figure 1](#)).

Signed Introspection Responses To sign the introspection response (Step [19] in [Figure 1](#)), the *JWT Response for OAuth Token Introspection* [35] specification is applied.

Signed HTTP Messages To sign resource requests and responses (Steps [18] and [20] in [Figure 1](#)), *HTTP Message Signatures* [2] are used. Note that this profile allows signing only one of resource request and resource response, or both.

All available profiles can be deployed alone or in any combination with the others. Hence, our model covers all possible combinations of said profiles. Regarding security, the message signing profiles should of course not weaken any of the security guarantees provided by the FAPI 2.0 Security Profile. In addition, each profile is expected to provide non-repudiation for the covered message(s), i.e., if a message is signed with some private key, the owner of that key cannot convincingly deny having signed the message. Note that any interpretation as to what “having signed a message” (or “owning a key”, for that matter) means in legal terms is outside the scope of FAPI 2.0 Message Signing and is up to each ecosystem to define, FAPI 2.0 Message Signing just aims to provide the aforementioned non-repudiation on a technical level.

We give a high-level description of the non-repudiation properties that we analyze in [Section III](#).

F. FAPI-CIBA

FAPI 2.0 provides a profile of the OpenID Connect Client-Initiated Backchannel Authentication Flow (CIBA) [11]. CIBA is designed for use cases in which the end user aims to authenticate and authorize a so-called Consumption Device, but uses a different device for providing consent. The user interacts with the OpenID Relying Party, i.e., the client in FAPI terms, through the Consumption Device.

On a high level, the Consumption Device starts the flow by sending the authentication request via the backchannel, i.e., directly instead of through a redirect, to the AS. This request contains information about the end user who is using the Consumption Device. The AS then contacts the end user and obtains authorization. After obtaining authorization, the Consumption Device can get access, id, and refresh tokens from the AS.

The Consumption Device has three options to get the tokens, called *delivery modes*: by polling the token endpoint after sending the authentication request (poll), by waiting for a message from the AS indicating that the user finished the authorization

process and then sending a token request (`ping`), or by relying on the AS to directly send the tokens (`push`). For the `ping` and `push` modes, the client needs to register an endpoint at the AS.

We now give some more detail, focussing on the FAPI-CIBA profile [49] in the context of FAPI 2.0:

FAPI-CIBA mandates that the AS only supports confidential clients, and prohibits the use of the `push` mode.

The authentication request sent by the Consumption Device to the AS can provide information about the end-user using one of three methods: By using a token that contains information about the end-user (`login_hint_token`), by using a previously issued ID token (`id_token_hint`), or some other value that can be used for identifying the end user, e.g., an email address (`login_hint`). This request needs to be authenticated as mandated by the FAPI 2.0 Security Profile (see Section II-B).

Upon receiving and validating this request, the AS responds with a unique identifier for the request (`auth_req_id`).

If the Consumption Device uses the `poll` mode, it obtains the tokens after the end user granted their authorization to the AS.

When using the `ping` mode, the AS sends a request to the previously mentioned endpoint at the Consumption Device. This request contains the request identifier `auth_req_id` and a token that the Consumption Device provides in the authentication request. After receiving this request, the Consumption Device obtains the tokens by sending a token request to the token endpoint of the AS.

Note that the token request needs to be authenticated and the access token needs to be sender-constrained as mandated by the FAPI 2.0 Security Profile (see Section II-B).

W.r.t. security, we expect FAPI-CIBA in the context of FAPI 2.0 to provide the same security as the FAPI 2.0 Security Profile with the authorization code flow.

III. SECURITY PROPERTIES

We analyze the protocols and extensions described in the previous section w.r.t. the security goals of the FAPI 2.0 Attacker Model (see Section II-C) as well as non-repudiation properties for signed messages (see Section II-E). In the following, we give an informal overview of the properties that we analyze and refer to Appendix C for the full formalized version.

Authorization An attacker should never be able to access resources of honest users (unless the user authorized such access), as long as the involved AS, RS, and client are honest.

Authentication An attacker should not be able to log in at an honest client under the identity of an honest user as long as the AS governing the identity is honest.

Session Integrity for Authorization An honest user is accessing their own resources and not the resources of the attacker, given that the AS, client, and RS are honest.

Session Integrity for Authentication An honest user, after logging in, is indeed logged in under their own account and not under the account of an attacker, given that the AS and client are honest.

Non-Repudiation for Signed Authorization Requests If an honest AS receives a signed PAR, and if the signature is valid for a key which an honest client registered at the AS, then this client indeed signed the PAR.

Non-Repudiation for Signed Authorization Responses If an honest client accepts a signed authorization response and if the authorization response is signed with the key of an honest AS, then this AS indeed signed the response.

Non-Repudiation for Signed Introspection Responses If an honest RS requests a signed introspection response and accepts a corresponding introspection response, then that response was signed, and if the signing key belongs to an honest AS, then this AS indeed signed the introspection response before.

Non-Repudiation for Signed Resource Requests/Responses If an honest RS processes a signed resource request and responds by providing access to some resource, and if the corresponding signing key belongs to an honest client, then this client indeed signed the resource request. Similarly, if the client accepts a signed resource response, and if the corresponding signing key belongs to the RS, then the RS signed the resource response.

IV. 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) [17], 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 [10, 16–18, 20, 21]. 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 [20], with the full model given in [Appendix E](#): 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., [1]), 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

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

where the body and the headers are empty. An HTTPS request for r is of the form $\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}}))$ 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 $\text{dec}_a(\text{enc}_a(x, \text{pub}(y)), y) = x$. With this, we have that, for example,

$$\text{dec}_a(\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}})), k_{\text{example.com}}) \equiv \langle r, k' \rangle$$

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

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

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.

V. MODELING DECISIONS AND ASSUMPTIONS

In the following, we describe and explain some of our modeling decisions, in particular those which pose what we call *over-approximations*, i.e., where our model is – if anything – less secure than a real implementation. Such over-approximations usually allow for a simpler model without jeopardizing expressiveness of security proofs. However, they need to be chosen carefully, as to not lead to false-positives, i.e., attacks on the model which would not work in a real implementation. In explaining these, we also give a few deeper insights into the WIM methodology.

Furthermore, some assumptions about certain details or optional features in the relevant specifications turned out to be necessary in order to prove our security properties. In this section, we also lay out these assumptions, explain why they are needed, and what possible alternative assumptions there are. Note however that strictly speaking, our security results only hold with our exact assumptions, we did not formally prove the listed alternatives to be equally secure. We note that the assumptions and modeling decisions laid out in [24] are also part of this model, unless stated otherwise.

We begin with some basic assumptions of the WIM methodology.

Cryptography. The WIM is a symbolic, Dolev-Yao-style model, i.e., bytestrings of any kind are represented as formal terms over a set of function symbols (e.g., $\text{sig}(\cdot, \cdot)$, $\text{enc}_s(\cdot, \cdot)$), nonces, and constants. The nonces are considered to be infinite-entropy random values, which means they can never be guessed, and must instead be learned, e.g., from received messages. Constants, on the other hand, are considered to be publicly known. Additionally, the semantics of cryptographic primitives are defined by an equational theory (see Figure 5 in the appendix).

The latter implies that cryptography is considered to be perfect: the attacker cannot break any cryptographic primitive, unless it learns the necessary keys (which are usually nonces).

Time. The WIM, and hence our model does not include any notion of time. Consequently, all time-based claims, values, and checks are omitted. Examples are not-before and expiration times of JWTs and tokens. Instead, we model all these values as being valid forever.

Note that strictly speaking, this is not an over-approximation: the WIM is a *possibilistic* model, i.e., anything that can happen – no matter how improbable – is considered to happen. Hence, even if we had a notion of time, the possibilistic nature of the model would still allow for arbitrarily complex attacks to happen in any non-zero time frame.

Canonicalization for HTTP Message Signatures. As mentioned before, the WIM is a symbolic model and bytestrings are represented as formal terms. Hence, there is no need for (and no point in) modeling encodings or canonicalization in the context of HTTP message signatures.

A. Dynamic Client Registration

Attacker-Chosen Client Ids. In our model, we let the attacker choose the client ids issued by (honest) ASs. This is modeled as follows: The attacker can – at any point – send a message containing any term t to an AS which does not “fit” any of the usual AS endpoints. Upon receiving such a message, the AS ensures that it has not yet registered a client under client id t , and adds t to its list of pending client ids (see Algorithm 14).

When the AS later processes a client’s registration request, it selects one of these pending client ids and issues it to the client (see [Algorithm 13](#)). Note that the possibilistic nature of the WIM implies that we capture all possible assignments of pending client ids at once, i.e., including the worst case (if there is any worst case – actually, the value of the client id does not matter at all for our security properties, as long as there are no two clients with the same client id at the same AS).

Initial Access Token. We model an open system without the use of initial access tokens for client registration (as defined by [RFC 7591 \[42\]](#)). As the provisioning of initial access tokens is out of scope, we would need to make additional assumptions and hence limit the applicability of our results to ecosystems which meet these assumptions.

Software Statements. On a similar note, our client model does not use software statements (see [Sec. 3.1.1 of RFC 7591 \[42\]](#)). Once again, we would need additional assumptions on how client statements are issued and provisioned, hence limiting the applicability of our results. Note, however, that the end user in our model does not “decide” which clients to authorize, they instead always choose to grant authorization, even to malicious clients; our security properties are formulated such that they only consider resources and identities involved in flows with honest clients. This allows us to consider all possible combinations of honest and malicious clients, interacting with the same end users and ASs at the same time, and proving that the flows involving honest parties are secure. Additionally, the information presented by the AS for the end user to identify a client is out of scope of all relevant specifications, hence modeling that decision process would require additional assumptions.

In other words: in a real-world ecosystem, the end user of course needs a way to verify which client they are authorizing. Software statements and/or initial access tokens (depending on how they are issued) can provide the AS with *reliable* information about a client which the AS can then show to the end user.

Processing of Client Information Response. Our client model verifies that the AS registered the client with the exact key material (for client authentication and access token sender constraining) provided by the client in its client registration request. We mention this seemingly obvious behavior as an assumption because [RFC 7591 \[42\]](#) technically allows for the AS to change any value, including keys, from the client’s registration request.

B. Dynamic Client Management

Triggering Client Configuration Changes. In our model of DCM, we make use of the WIM’s TRIGGER messages: these can be sent by the attacker to any party at any time. Upon receiving such a message, our client model non-deterministically chooses one of several actions (see [Algorithm 9](#)). One of these actions is CHANGE_CLIENT_CONFIG and if this option is selected, the client does the following: 1) it randomly chooses an AS it has an account at. 2) it randomly selects whether to update or to delete its configuration at that AS. In the latter case, the client sends an appropriate DELETE request to the AS. In the former case, the client continues: 3) the client generates fresh key pairs for mTLS and signing, and 4) sends an appropriate PUT request to update its configuration at the AS.

Note that due to the possibilistic nature of the WIM, this subsumes all possible interleavings of ongoing grants and client configuration changes.

Client Key Turnover. If a client updates its keys, we model the AS to immediately invalidate the client’s former keys (as we do not have a notion of time, the choice is to invalidate immediately or keep them valid forever, see above). While this will prevent the client from using access tokens for which the RS uses token introspection, it does not prevent the client from continuing to use structured, self-contained access tokens.

Note however that the attacker – being the scheduler for all network messages – can delay or even drop any message, including a client’s update request.

For a real ecosystem, one might want to consider two different types of client key turnover: one for regular, scheduled key rotation with some kind of grace period during which AS and RS accept the old, as well as the new keys; and another one for immediate revocation, e.g., after private key leaks.

Registration Access Tokens. In line with [RFC 7592 \[43\]](#), our AS model issues a registration access token to a client during client registration. Additionally, these registration access tokens are never rotated in our model, i.e., they are valid forever (unless the client deletes itself from the AS by means of dynamic client management).

C. FAPI 2.0 Message Signing

Configuration of Clients, ASs, and RSs. To keep the necessary configuration small, we use non-deterministic choices in many places in our model of FAPI 2.0 Message Signing. For example, for each grant, the client decides non-deterministically whether to sign the authorization request – similarly, the AS decides non-deterministically whether to require a signed authorization request for each grant. This works because we can then formulate our security property such that non-repudiation is only considered for grants in which both the client and the AS agree on signing the authorization request/requiring a signed authorization request.

I.e., our model allows for many flows which are not necessarily possible in a real implementation, but due to the way our security properties are formulated, we can still prove useful properties for the “interesting” flows, e.g., ones in which client and AS agree on whether an authorization request should be signed or not.

A similar approach is taken for all other parts of FAPI 2.0 Message Signing as well.

Provisioning of Verification Keys. Neither FAPI 2.0 Message Signing, nor the HTTP Message Signing specification [2], define the means by which the receiver of a signed message obtains the corresponding verification key. Hence, for our model, we have to make assumptions on how clients and resource servers obtain those keys.

As for clients, we assume that a client is configured with the public verification keys for each resource server it uses (in the model, we initialize clients with the public keys for all resource servers, identified by their domains – note that this is a consequence of allowing all clients to interact with all resource servers). Since a client knows which resource server it sent a resource request to, it can select the correct verification key for that resource server.

For resource servers, we assume that the public key to verify a client’s HTTP Message Signature is provided by the AS, either as part of a structured access token or in the AS’ introspection response. While we did not formally verify this, using AS-supplied public keys at the RS implicitly introduces another layer of access token sender constraining – however, since Work Package 1 (b) [24] already proves that access token sender constraining works as intended without HTTP Message Signatures, we do not miss any attacks by adding this second layer of sender constraining in our model.

Linking of Resource Request and Resource Response. At the time of this writing, the FAPI 2.0 Message Signing specification does not give definitive guidance on when to link a resource request to a resource response when signing the latter. Since the non-repudiation goals laid out in the FAPI 2.0 Message Signing specification do not require this linking, our model once again tries to be as insecure as possible (while still following the specifications). Hence, our RS model never includes parts of the resource request in its resource response signature.

D. FAPI-CIBA

1) General Notes

In the model, the consumption device is subsumed by the client and the authentication device is modeled as a modified browser, which can receive requests from an AS. Furthermore, we allow anyone to start a flow at a client by sending a request to the client containing user information.

We only model support for unsigned authentication requests (FAPI-CIBA does not require either party to support signed authentication requests in the context of FAPI 2.0). Thus, we omit the `backchannel_authentication_request_signing_also_values_supported` metadata information.

We note that within the model, an AS asks the end user for consent by sending a request to the end user’s browser. To make this possible, we assume that browsers have their own domains.

2) Assumption: User starts CIBA flow with their own identity

The session integrity properties do not hold if the user starts a CIBA flow with an attacker-controlled identity: In this case, the AS would contact the attacker and obtain authorization, and the AS would provide an access token for attacker resources or an id token containing information about the attacker. Thus, the consumption device that the user is using would provide access to attacker resources or consider the attacker identity to be logged in, breaking the session integrity properties.

In practice, the user might select an attacker-controlled identity by mistake, e.g., by a typing error, or in the case of publicly available consumption devices, the attacker could initiate the flow with an attacker-controlled identity and then leave the device ready for another user.

For the analysis, we assume that an honest user always starts a fresh flow at a client and only uses identities under their control.

Note that on a protocol level, neither the AS, nor the client can detect this scenario.

3) FAPI 2.0 CIBA Cross-Device Consent Phishing Attack

CIBA is susceptible to Cross-Device Consent Phishing Attacks (see also [30]). In its most basic form, the attacker starts a flow at a Consumption Device (CD), but selects the identity of an honest user. The authorization server asks the user whether they give their consent. An inattentive user might give their consent, providing the attacker access to their resources and logging in the attacker at the CD as themselves.

To mitigate this risk, CIBA specifies an optional binding message that is created by the client, displayed on the consumption device, and sent to the AS which displays the message to the end-user.³ Section 7.1 of the CIBA specification [11] states the following on the binding message:

³FAPI-CIBA [49] mandates ASs to require a binding message in the authentication request, unless the authorization request uniquely identifies the authorization context.

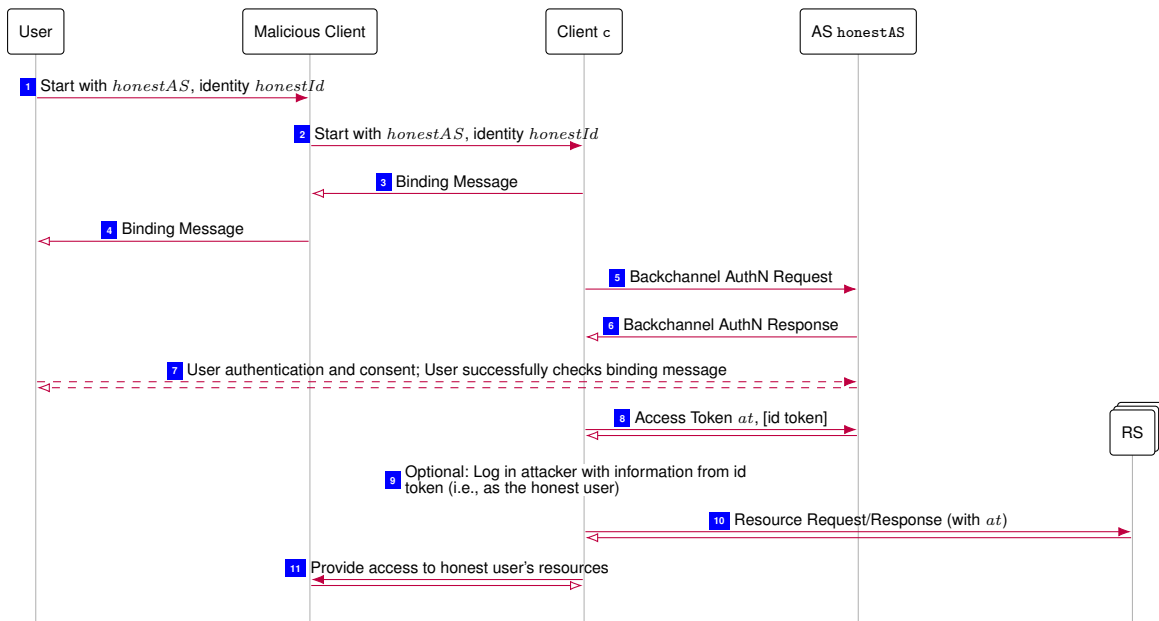


Figure 2. FAPI 2.0 CIBA Cross-Device Consent Phishing Attack (Simplified)

A human-readable identifier or message intended to be displayed on both the consumption device and the authentication device to interlock them together for the transaction by way of a visual cue for the end-user. This interlocking message enables the end-user to ensure that the action taken on the authentication device is related to the request initiated by the consumption device. The value SHOULD contain something that enables the end-user to reliably discern that the transaction is related across the consumption device and the authentication device, such as a random value of reasonable entropy [...]

This seems to suggest that if the end-user is paying attention to the binding message, they cannot accidentally provide the attacker access to their resources or log in the attacker under their own identity. However, if an end-user uses a malicious client (or consumption device), the client can display a binding message from a different flow, as shown in Figure 2.

Possible Real-World Fix. As suggested by [30], ensuring proximity between the client/CD that is being authorized and the end-user might prevent this class of attacks in most practical settings (we note that we do not analyze this fix, thus, we cannot assess its effectiveness).

Fix within the Model. Within the model, we assume that the end-user can identify the client that they are using unambiguously. For this, we assume that each client has a domain that the end-user can identify when starting a flow. At the consent step, our AS model provides the domain of the redirect endpoint of the client to the end-user, and the end-user checks whether they have received the binding message from this client.

In other words, we assume an authenticated channel between the end-user and the client, which is what the mitigations in [30] aim to achieve as well (or, to be precise: the mitigations aim to make it harder for an attacker to exploit the unauthenticated-ness of said channel). The WIM is a possibilistic model, so we have to assume an authenticated channel (instead of a “very hard to exploit unauthenticated channel”).

ACKNOWLEDGMENT

This research was funded in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 443324941, and by the OpenID Foundation.

REFERENCES

- [1] M. Abadi and C. Fournet. “Mobile Values, New Names, and Secure Communication”. In: *ACM POPL*. 2001, pp. 104–115.
- [2] A. Backman, J. Richer, and M. Sporny. *HTTP Message Signatures*. Internet-Draft draft-ietf-httpbis-message-signatures-17. Work in Progress. Internet Engineering Task Force, 2023. 117 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-httpbis-message-signatures/17/>.
- [3] Banco Central do Brasil. *Open Finance*. URL: https://www.bcb.gov.br/en/financialstability/open_finance.
- [4] R. Berjon et al., eds. *HTML5, W3C Recommendation*. 2014. URL: <http://www.w3.org/TR/html5/>.
- [5] V. Bertocci. *JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens*. RFC 9068. 2021. URL: <https://www.rfc-editor.org/info/rfc9068>.
- [6] *Budget Tracker & Planner*. URL: <https://mint.intuit.com/>.
- [7] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. RFC 8705. 2020. URL: <https://www.rfc-editor.org/info/rfc8705>.
- [8] L. Chen, S. Englehardt, M. West, and J. Wilander. *Cookies: HTTP State Management Mechanism*. Internet-Draft draft-ietf-httpbis-rfc6265bis-09. Work in Progress. Internet Engineering Task Force, 2021. 59 pp. URL: <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-09>.
- [9] Credit Sense. *Bank Account Aggregation Services Australia*. URL: <https://www.creditsense.com.au/bank-account-aggregation/>.
- [10] Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, N. Wenzler, and T. Würtele. “A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification”. In: *IEEE S&P*. 2022, pp. 215–234.
- [11] G. Fernandez, F. Walter, A. Nennker, D. Tonge, and B. Campbell. *OpenID Connect Client-Initiated Backchannel Authentication Flow - Core 1.0*. OpenID Foundation. 2021. URL: https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html.
- [12] D. Fett. “An Expressive Formal Model of the Web Infrastructure”. PhD thesis. 2018.
- [13] D. Fett. *FAPI 2.0 Attacker Model. Second Implementer’s Draft*. 2022. URL: https://openid.net/specs/fapi-2_0-attacker-model-02.html.
- [14] D. Fett. *FAPI 2.0 Security Profile. Second Implementer’s Draft*. 2022. URL: https://openid.net/specs/fapi-2_0-security-profile-ID2.html.
- [15] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, and D. Waite. *OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)*. Internet-Draft draft-ietf-oauth-dpop-08. Work in Progress. Internet Engineering Task Force, 2022. 41 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-oauth-dpop/08/>.
- [16] D. Fett, P. Hosseyni, and R. Küsters. “An Extensive Formal Security Analysis of the OpenID Financial-grade API”. In: *IEEE S&P*. 2019, pp. 1054–1072.
- [17] D. Fett, R. Küsters, and G. Schmitz. “An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System”. In: *IEEE S&P*. 2014, pp. 673–688.
- [18] D. Fett, R. Küsters, and G. Schmitz. “Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web”. In: *ESORICS*. Vol. 9326. LNCS. 2015, pp. 43–65.
- [19] D. Fett, R. Küsters, and G. Schmitz. “SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web”. In: *ACM CCS*. 2015.
- [20] D. Fett, R. Küsters, and G. Schmitz. “A Comprehensive Formal Security Analysis of OAuth 2.0”. In: *ACM CCS*. 2016.
- [21] D. Fett, R. Küsters, and G. Schmitz. “The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines”. In: *IEEE CSF*. 2017.
- [22] D. Fett and D. Tonge. *FAPI 2.0 Message Signing, Commit 67246ac*. OpenID Foundation. 2023. URL: https://bitbucket.org/openid/fapi/src/67246ac44d2ee136c184789b9757ba44df57c7b8/fapi-2_0-message-signing.md.
- [23] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. 2012. URL: <https://www.rfc-editor.org/info/rfc6749>.
- [24] P. Hosseyni, R. Küsters, and T. Würtele. *Formal Security Analysis of the OpenID Financial-grade API 2.0*. Tech. rep. WP 1(b). OpenID Foundation, 2022. URL: https://openid.net/wordpress-content/uploads/2022/12/Formal-Security-Analysis-of-FAPI-2.0_FINAL_2022-10.pdf.
- [25] P. Hosseyni, R. Küsters, and T. Würtele. “Formal Security Analysis of the OpenID FAPI 2.0: Accompanying a Standardization Process”. In: *CSF 2024*. To appear. 2024.
- [26] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. 2015. URL: <https://www.rfc-editor.org/info/rfc7519>.
- [27] M. Jones, B. Campbell, and C. Mortimore. *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants*. RFC 7523. 2015. URL: <https://www.rfc-editor.org/info/rfc7523>.
- [28] M. Jones, B. Campbell, C. Mortimore, and Y. Y. Goland. *Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants*. RFC 7521. 2015. URL: <https://www.rfc-editor.org/info/rfc7521>.

- [29] M. Jones, N. Sakimura, and J. Bradley. *OAuth 2.0 Authorization Server Metadata*. RFC 8414. 2018. URL: <https://www.rfc-editor.org/info/rfc8414>.
- [30] P. Kasselmann, D. Fett, and F. Skokan. *Cross-Device Flows: Security Best Current Practice*. Internet-Draft draft-ietf-oauth-cross-device-security-02. Work in Progress. Internet Engineering Task Force, 2023. 37 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-oauth-cross-device-security/02/>.
- [31] R. Küsters, G. Schmitz, and D. Fett. *The Web Infrastructure Model (WIM)*. Technical Report. Version 1.0. 2022. URL: https://www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf.
- [32] T. Lodderstedt and B. Campbell. *Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)*. OpenID Foundation. 2018. URL: <https://openid.net/specs/openid-financial-api-jarm.html>.
- [33] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett. *OAuth 2.0 Security Best Current Practice*. Internet-Draft. Work in Progress. Internet Engineering Task Force, 2021. 52 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/19/>.
- [34] T. Lodderstedt, B. Campbell, N. Sakimura, D. Tonge, and F. Skokan. *OAuth 2.0 Pushed Authorization Requests*. RFC 9126. 2021. URL: <https://www.rfc-editor.org/info/rfc9126>.
- [35] T. Lodderstedt and V. Dzhuvinov. *JWT Response for OAuth Token Introspection*. Internet-Draft draft-ietf-oauth-jwt-introspection-response-12. Work in Progress. Internet Engineering Task Force, 2021. 19 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-oauth-jwt-introspection-response/12/>.
- [36] V. Mladenov, C. Mainka, and J. Schwenk. “On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect”. In: *CoRR* abs/1508.04324v2 (2015). arXiv: 1508.04324v2 [cs.CR]. URL: <http://arxiv.org/abs/1508.04324v2>.
- [37] *Open Banking UK*. URL: <https://www.openbanking.org.uk/>.
- [38] A. Parecki. *OAuth 2.0 Simplified: Token Introspection Endpoint*. URL: <https://www.oauth.com/oauth2-servers/token-introspection-endpoint/>.
- [39] J. Reschke. *The ‘Basic’ HTTP Authentication Scheme*. RFC 7617. 2015. URL: <https://www.rfc-editor.org/info/rfc7617>.
- [40] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. 2008. URL: <https://www.rfc-editor.org/info/rfc5246>.
- [41] J. Richer. *OAuth 2.0 Token Introspection*. RFC 7662. 2015. URL: <https://www.rfc-editor.org/info/rfc7662>.
- [42] J. Richer, M. Jones, J. Bradley, M. Machulak, and P. Hunt. *OAuth 2.0 Dynamic Client Registration Protocol*. RFC 7591. 2015. URL: <https://www.rfc-editor.org/info/rfc7591>.
- [43] J. Richer, M. B. Jones, J. Bradley, and M. Machulak. *OAuth 2.0 Dynamic Client Registration Management Protocol*. RFC 7592. 2015. URL: <https://www.rfc-editor.org/info/rfc7592>.
- [44] N. Sakimura, J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. 2015. URL: <https://www.rfc-editor.org/info/rfc7636>.
- [45] N. Sakimura, J. Bradley, M. Jones, and E. Jay. *OpenID Connect Discovery 1.0 incorporating errata set 1*. OpenID Foundation. 2014. URL: http://openid.net/specs/openid-connect-discovery-1_0.html.
- [46] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Foundation. 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html.
- [47] N. Sakimura, J. Bradley, and M. Jones. *The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)*. RFC 9101. 2021. URL: <https://www.rfc-editor.org/info/rfc9101>.
- [48] K. M. zu Selhausen and D. Fett. *OAuth 2.0 Authorization Server Issuer Identification*. RFC 9207. 2022. URL: <https://www.rfc-editor.org/info/rfc9207>.
- [49] D. Tonge. *FAPI Client Initiated Backchannel Authentication Profile, Commit ac96a05*. OpenID Foundation. 2023. URL: https://bitbucket.org/openid/fapi/src/ac96a05126cd451e83dd253500a8777bcaa72c96/Financial_API_WD_CIBA.md.

APPENDIX A

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* [7] is modeled. We continue with the formal models of the FAPI 2.0 clients (Appendix A-J), the FAPI 2.0 ASs (Appendix A-K), and the FAPI 2.0 RSs (Appendix A-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 A-K. RS is the set of processes representing resource servers, described in Appendix A-L. C is the set of processes representing clients, described in Appendix A-J. Finally, B is the set of processes representing browsers, including their users. They are described in Appendix E-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 1. An *identity* i is a term of the form $\langle name, domain \rangle$ with $name \in \mathbb{S}$ and $domain \in \text{Doms}$. 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 $\text{governor} : \text{ID} \rightarrow \mathcal{W}$, $\langle name, domain \rangle \mapsto \text{dom}^{-1}(domain)$ and $\text{ID}^y := \text{governor}^{-1}(y)$.

C. Keys and Secrets

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

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

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 $\text{tlskey} : \text{Doms} \rightarrow K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $\text{tlskeys}^p = \langle \{ \langle d, \text{tlskey}(d) \rangle \mid d \in \text{dom}(p) \} \rangle$ (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 and RSs to sign HTTP messages. Let $\text{signkey} : \text{AS} \times \text{C} \times \text{RS} \rightarrow K_{\text{sign}}$ be an injective mapping that assigns a (different) signing key to every AS, client, and RS. Note that clients also sign other things, e.g., DPoP proofs, but the keys used there are not part of K_{sign} , but are taken from N (those keys are freshly chosen by a client when it registers with an AS).
- The set Passwords is the set of passwords (secrets) the browsers share with servers. These are the passwords the users use to log in. Let $\text{secretOfID} : \text{ID} \rightarrow \text{Passwords}$ be a bijective mapping that assigns a password to each identity.
- The set RScredentials is a set of secrets shared between authorization and resource servers. RSs use these to authenticate at ASs' token introspection endpoints. Let $\text{secretOfRS} : \text{Doms} \times \text{Doms} \rightarrow \text{RScredentials}$ be a partial mapping, assigning a secret to some of the RS-AS pairs (with the function arguments in that order).

D. Passwords

Definition 2. Let $\text{ownerOfSecret} : \text{Passwords} \rightarrow \text{B}$ be a mapping that assigns to each password a browser which *owns* this password. Similarly, we define $\text{ownerOfID} : \text{ID} \rightarrow \text{B}$ as $i \mapsto \text{ownerOfSecret}(\text{secretOfID}(i))$, 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 $b \in B$) are modeled as described in [Appendix E](#). Before defining the initial states of Web browsers, we introduce the following set (for some process p):

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

Definition 3 (Initial Web Browser State for FAPI). The initial state of a Web browser process $b \in B$ follows the description in [Definition 74](#), with the following additional constraints:

- $s_0^b.\text{ids} \equiv \langle \{i \mid b = \text{ownerOfID}(i)\} \rangle$
- $s_0^b.\text{secrets}$ contains an entry $\langle \langle d, S \rangle, \langle \text{Secrets}^{b,p} \rangle \rangle$ for each $p \in \text{AS} \cup \text{C} \cup \text{RS}$ and every domain $d \in \text{dom}(p)$ (and nothing else), i.e.,

$$s_0^b.\text{secrets} \equiv \left\langle \left\{ \left\langle \langle d, S \rangle, \langle \text{Secrets}^{b,p} \rangle \right\rangle \mid \exists p, d : p \in \text{AS} \cup \text{C} \cup \text{RS} \wedge d \in \text{dom}(p) \right\} \right\rangle$$

- $s_0^b.\text{keyMapping} \equiv \langle \{ \langle d, \text{pub}(\text{tlskey}(d)) \rangle \mid d \in \text{Doms} \} \rangle$

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

Definition 5. For each $rs \in \text{RS}$, let $\text{supportedAuthorizationServer}^{rs} \subseteq \text{AS}$ be a finite set of ASs. These are the ASs supported by the RS.

Definition 6. For each $rs \in \text{RS}$, let $\text{authorizationServerOfResource}^{rs} : \text{resourceURLPath}^{rs} \rightarrow \text{supportedAuthorizationServer}^{rs}$ 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. Modeling mTLS

OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens (mTLS) [7] provides a method for both client authentication and token binding. Note that both mechanisms may be used independently of each other.

OAuth 2.0 Mutual TLS Client Authentication makes use of *TLS client authentication*⁵, 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. 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

⁴A resource is managed by the RS if and only if $\text{resourceID} \in \text{resourceURLPath}^{rs}$.

⁵As noted in Section 7.2 of [7], this extension supports all TLS versions with certificate-based client authentication.

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.⁶

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

We model mTLS similarly to [16]. An overview of the mTLS model is shown in Figure 3. 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 [40, 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.

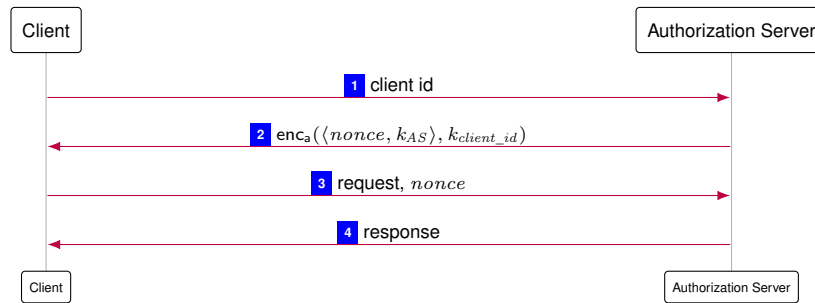


Figure 3. Overview of mTLS model

All messages are sent by the generic HTTPS server model (Appendix E-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 4 shows the complete messages, i.e., with the encryption used for transmitting the messages.

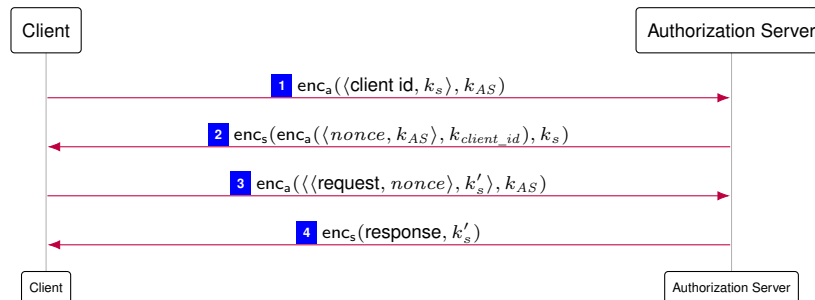


Figure 4. Detailed view on mTLS model

⁶The 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.

H. Additional HTTP Headers

In order to model FAPI 2.0, we extend the list of headers of [Definition 48](#) with the following headers:

- For DPoP, we add the header $\langle \text{DPoP}, p \rangle$ where $p \in \mathcal{T}_{\mathcal{N}}$ is (for honest senders) a DPoP proof (i.e., a signed JWT).
- The Authorization header can also take on values $\langle \text{Bearer}, t \rangle$ where $t \in \mathcal{T}_{\mathcal{N}}$ is usually a bearer token.
- We add the header $\langle \text{Accept}, s \rangle$ with $s \in \mathbb{S}$.
- For HTTP Message Signatures, we add the following headers
 - $\langle \text{Signature-Input}, inputs \rangle$ where $inputs$ is a dictionary of elements $label: t$ with $t \in \mathcal{T}_{\mathcal{N}}, label \in \mathbb{S}$. For honest senders, t is of the form $\langle s, p \rangle$ where s is a sequence of pairs, each containing a HTTP message component identifier and a possibly empty sequence of parameters; whereas p is a dictionary of signature parameters with their values. E.g.,

$$\left[\text{label1}: \left\langle \left\langle @\text{status}, \langle \rangle \right\rangle, \left\langle \text{content-digest}, \langle \text{req} \rangle \right\rangle, [\text{keyid}: \text{some_id}] \right\rangle \right]$$
 - $\langle \text{Signature}, sigs \rangle$ where $sigs$ is a dictionary of elements $label: t$ with $t \in \mathcal{T}_{\mathcal{N}}, label \in \mathbb{S}$. For honest senders, t is a signature.
 - $\langle \text{Content-Digest}, digest \rangle$ where $digest \in \mathcal{T}_{\mathcal{N}}$ is – for honest senders – a hash of the message body.

I. Helper Functions

The following helper function is used by processes when verifying HTTP message signatures.

Algorithm 1 Compare component values for HTTP message signatures.

```

1: function IS_COMPONENT_EQUAL( $m, request, signerSignatureBase, component$ )  $\rightarrow request$  may be empty ( $\diamond$ )
2:   let  $componentName := component.1$ 
3:   let  $componentParam := component.2$ 
4:   let  $knownComponents := \{ @\text{method}, @\text{target-uri}, @\text{status}, \text{authorization}, \text{content-digest}, \text{dpop} \}$ 
5:   if  $componentName \notin knownComponents$  then
6:     return  $\perp$ 
7:   let  $componentValue := \diamond$ 
8:   if  $componentParam \equiv \langle \rangle$  then  $\rightarrow$  Compare against component value from  $m$ 
9:     if  $componentName \equiv @\text{method}$  then
10:       let  $componentValue := m.method$ 
11:     if  $componentName \equiv @\text{target-uri}$  then
12:       let  $componentValue := \langle \text{URL}, m.protocol, m.host, m.path, m.parameters, \perp \rangle$ 
13:     if  $componentName \equiv @\text{status}$  then
14:       let  $componentValue := m.status$ 
15:     if  $componentName \equiv \text{authorization}$  then
16:       let  $componentValue := m.headers[\text{Authorization}]$ 
17:     if  $componentName \equiv \text{content-digest}$  then
18:       let  $componentValue := m.headers[\text{Content-Digest}]$ 
19:     if  $componentName \equiv \text{dpop}$  then
20:       let  $componentValue := m.headers[\text{DPoP}]$ 
21:   else if  $componentParam \equiv \langle \text{req} \rangle$  then  $\rightarrow$  Compare against component value from request
22:     if  $componentName \equiv @\text{method}$  then
23:       let  $componentValue := request.method$ 
24:     if  $componentName \equiv @\text{target-uri}$  then
25:       let  $componentValue := \langle \text{URL}, request.protocol, request.host, request.path, request.parameters, \perp \rangle$ 
26:     if  $componentName \equiv \text{content-digest}$  then
27:       let  $componentValue := request.headers[\text{Content-Digest}]$ 
28:     if  $componentName \equiv \text{dpop}$  then
29:       let  $componentValue := request.headers[\text{DPoP}]$ 
30:   else
31:     return  $\perp$   $\rightarrow$  Unsupported component parameter
32:   if  $componentValue \equiv signerSignatureBase[component]$  then
33:     return  $\top$ 
34:   else
35:     return  $\perp$ 

```

J. Clients

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

Definition 7. A state $s \in Z^c$ of a client c is a term of the form $\langle \text{DNSaddress}, \text{pendingDNS}, \text{pendingRequests}, \text{corrupt}, \text{keyMapping}, \text{tlskeys}, \text{sessions}, \text{oauthConfigCache}, \text{jwksCache}, \text{asAccounts}, \text{mtlsCache}, \text{pendingCIBAResponses}, \text{resourceASMapping}, \text{dpopNonces}, \text{jwk}, \text{rsSigKeys} \rangle$ with $\text{DNSaddress} \in \text{IPs}$, $\text{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$, $\text{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\text{keyMapping} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\text{tlskeys} \in [\text{Doms} \times K_{\text{TLS}}]$ (all former components as in Definition 77), $\text{sessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{oauthConfigCache} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\text{jwksCache} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\text{asAccounts} \in [\text{Doms} \times [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]]$, $\text{mtlsCache} \in \mathcal{T}_{\mathcal{N}}$, $\text{pendingCIBAResponses} \in \mathcal{T}_{\mathcal{N}}$, $\text{resourceASMapping} \in [\text{Doms} \times [\mathbb{S} \times \text{Doms}]]$, $\text{dpopNonces} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\text{jwk} \in K_{\text{sign}}$, and $\text{rsSigKeys} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$.

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

- $s_0^c.\text{DNSaddress} \in \text{IPs}$,
- $s_0^c.\text{pendingDNS} \equiv \langle \rangle$,
- $s_0^c.\text{pendingRequests} \equiv \langle \rangle$,
- $s_0^c.\text{corrupt} \equiv \perp$,
- $s_0^c.\text{keyMapping}$ being the same as the keymapping for browsers,
- $s_0^c.\text{tlskeys} \equiv \text{tlskeys}^c$ (see Appendix A-C),
- $s_0^c.\text{sessions} \equiv \langle \rangle$,
- $s_0^c.\text{oauthConfigCache} \equiv \langle \rangle$,
- $s_0^c.\text{jwksCache} \equiv \langle \rangle$,
- $s_0^c.\text{asAccounts} \equiv \langle \rangle$,
- $s_0^c.\text{mtlsCache} \equiv \langle \rangle$,
- $s_0^c.\text{pendingCIBAResponses} \equiv \langle \rangle$ (Upon receiving a CIBA start request, the client responds with a binding message and by setting a cookie. The client stores the necessary information in this field and continues the flow upon receiving a trigger message),
- $s_0^c.\text{resourceASMapping}[\text{domRS}][\text{resourceID}] \in \text{dom}(\text{authorizationServerOfResource}^{rs}(\text{resourceID}))$, $\forall rs \in \text{RS}$ and $\forall \text{domRS} \in \text{dom}(rs)$ and $\forall \text{resourceID} \in \text{resourceURLPath}^{rs}$ (a domain of the AS managing the resource stored at rs identified by resourceID),
- $s_0^c.\text{dpopNonces} \equiv \langle \rangle$,
- $s_0^c.\text{jwk} \equiv \text{signkey}(c)$ (used for HTTP message signing, see Appendix A-C), and
- $s_0^c.\text{rsSigKeys} \equiv rsk$ such that $rsk[\text{domRS}] = \text{pub}(\text{signkey}(rs))$ for all $\text{domRS} \in \text{dom}(rs)$ for all $rs \in \text{RS}$ (see [22, Sec. 5.6.2.2]).

We now specify the relation R^c : This relation is based on the model of generic HTTPS servers (see Appendix E-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 3–9. 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 33).

The script that is used by the client on its index page is specified in Algorithm 10. This script uses the $\text{GETURL}(\text{tree}, \text{docnonce})$ function 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 2 Relation of a Client R^c – Processing HTTPS Requests

```
1: 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  $c$ .
2:   if  $m.path \equiv /$  then → Serve index page (start flow).
3:     let  $m' := enc_s(\langle \text{HTTPResp}, m.nonce, 200, headers, \langle \text{script\_client\_index}, \langle \rangle \rangle, k \rangle)$  → Reply with script_client_index.
4:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
5:   else if  $m.path \equiv /startLogin \wedge m.method \equiv \text{POST}$  then → Start a new FAPI 2.0 flow (see script_client_index)
6:     if  $m.headers[\text{Origin}] \neq \langle m.host, S \rangle$  then
7:       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).
8:     let  $selectedAS := m.body$ 
9:     let  $sessionId := \nu_1$  → Session id is a freshly chosen nonce.
10:    let  $s'.sessions[sessionId] := [\text{startRequest}: [message: m, key: k, receiver: a, sender: f],$ 
       $\hookrightarrow \text{selected\_AS}: selectedAS, cibaFlow: \perp]$ 
11:    call PREPARE_AND_SEND_INITIAL_REQUEST( $sessionId, a, s'$ ) → Start authorization flow with the AS (Algorithm 8)
12:  else if  $m.path \equiv /redirect\_ep$  then → User is being redirected after authentication to the AS.
13:    let  $sessionId := m.headers[\text{Cookie}][\langle \_Host, sessionId \rangle]$ 
14:    if  $sessionId \notin s'.sessions$  then
15:      stop
16:    let  $session := s'.sessions[sessionId]$  → Retrieve session data.
17:    let  $selectedAS := session[selected\_AS]$ 
18:    if  $session[\text{requested\_signed\_authz\_response}] \equiv \top$  then
19:      if  $\text{checksig}(m.parameters[\text{response}], s'.jwksCache[selectedAS]) \neq \top$  then
20:        stop → Invalid or missing signature on authorization response, see JARM [32, Sec. 2.4]
21:      let  $authzResponse := \text{extractmsg}(m.parameters[\text{response}])$ 
22:      if  $authzResponse[\text{aud}] \neq session[\text{client\_id}]$  then
23:        stop → Wrong/missing audience value, see JARM [32, Sec. 2.4]
24:      let  $m.parameters := authzResponse$  → Remove signature (so we always store a “plain” message below)
25:    else
26:      let  $authzResponse := m.parameters$ 
27:      if  $code \notin authzResponse \vee iss \notin authzResponse$  then
28:        stop
29:      let  $code := authzResponse[\text{code}]$ 
30:      let  $issuer := authzResponse[\text{iss}]$ 
31:      if  $issuer \neq selectedAS$  then → Check issuer parameter (RFC 9207 [48]).
32:        stop
      → Store browser’s request for use in CHECK_ID_TOKEN (Algorithm 7) and PROCESS_HTTPS_RESPONSE (Algorithm 3)
33:      let  $s'.sessions[sessionId][\text{redirectEpRequest}] := [message: m, key: k, receiver: a, sender: f]$ 
34:      call SEND_TOKEN_REQUEST( $sessionId, code, a, s'$ ) → Retrieve a token from AS’s token endpoint.
35:    else if  $m.path \equiv /start\_ciba$  then → Start a CIBA flow. We assume that anyone can start the flow at a client by providing the
      identity of an end-user (which the client uses as a login_hint)
36:      let  $selectedAS := m.body[\text{authServ}]$ 
37:      let  $identity := m.body[\text{identity}]$ 
38:      let  $sessionId := \nu_6$  → Session id is a freshly chosen nonce.
39:      let  $bindingMessage := \nu_{\text{bindingMsg}}$ 
40:      let  $s'.sessions[sessionId] := [\text{selected\_AS}: selectedAS, \text{selected\_identity}: identity,$ 
       $\hookrightarrow \text{binding\_message}: bindingMessage, \text{start\_polling}: \perp, \text{cibaFlow}: \top]$ 
      → Store record for continuing the flow later upon receiving a trigger message
41:      let  $s'.pendingCIBARRequests := s'.pendingCIBARRequests + \langle \langle sessionId, a \rangle \rangle$ 
42:      let  $headers := [\text{Set-Cookie}: [\langle \_Host, sessionId \rangle: \langle sessionId, \top, \top, \top \rangle]]$ 
43:      let  $body := [\text{binding\_message}: bindingMessage]$ 
44:      let  $m' := enc_s(\langle \text{HTTPResp}, m.nonce, 200, headers, body \rangle, k)$ 
45:      stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
46:    else if  $m.path \equiv /ciba\_notif\_ep$  then → CIBA notification endpoint
47:      let  $receivedNotificationToken := m.headers[\text{Authorization}].2$ 
48:      let  $receivedAuthReqId := m.body[\text{auth\_req\_id}]$ 
49:      let  $sessionId$  such that  $sessionId \in s'.sessions$ 
       $\hookrightarrow \wedge s'.sessions[sessionId][\text{client\_notification\_token}] \equiv receivedNotificationToken$ 
       $\hookrightarrow \wedge s'.sessions[sessionId][\text{client\_notification\_token}] \neq \langle \rangle$ 
       $\hookrightarrow \wedge s'.sessions[sessionId][\text{auth\_req\_id}] \equiv receivedAuthReqId$  if possible; otherwise stop
50:      call SEND_CIBA_TOKEN_REQUEST( $sessionId, a, s'$ ) → Send a token request
      → Algorithm continues on next page.
```

```

51: else if  $m.path \equiv /ciba\_get\_ssid\_or\_resource$  then
    → When starting a CIBA flow, the client responds with a Set-Cookie header with a login session id. Once the user login at the
    client is finished (i.e., after the client checks the ID token) or once the client gets access to some resource, the initiator can send
    a request to this endpoint (with the login session id cookie) and get logged in at the client or get access to resources that an RS
    provided to the client.
52: let  $sessionId := m.headers[Cookie][(\_Host, sessionId)]$ 
53: if  $sessionId \notin s'.sessions$  then
54:   stop
55: let  $session := s'.sessions[sessionId]$  → Retrieve session data.
56: if  $session[cibaFlow] \equiv \perp$  then
    → This endpoint can only be used for CIBA flows. The authorization code flow model provides this functionality when
    receiving the responses by the AS or RS.
57:   stop
58: if  $serviceSessionId \notin session \wedge resource \notin session$  then → User authentication/authorization not finished yet
59:   stop
60: let  $headers := []$ 
61: let  $body := []$ 
62: if  $serviceSessionId \in session$  then
63:   let  $serviceSessionId := session[serviceSessionId]$ 
64:   let  $headers[Set-Cookie] := [serviceSessionId: \langle serviceSessionId, T, T, T \rangle]$ 
65: if  $resource \in session$  then
66:   let  $body := session[resource]$ 
67: let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, headers, body \rangle, k)$ 
68: stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
69: stop → Unknown endpoint or malformed request.

```

Algorithm 3 Relation of a Client R^c – Processing HTTPS Responses

```
1: function PROCESS_HTTPS_RESPONSE( $m$ ,  $reference$ ,  $request$ ,  $a$ ,  $f$ ,  $s'$ )
2:   if  $reference[responseTo] \equiv \text{MTLS}$  then  $\rightarrow$  Client received an mTLS nonce (see Appendix A-G)
3:     let  $m_{dec}, k'$  such that  $m_{dec} \equiv \text{dec}_a(m.\text{body}, k') \wedge \text{selectedAS} \in s'.\text{asAccounts} \wedge s'.\text{asAccounts}[\text{selectedAS}][\text{tls\_key}] \equiv k'$ 
4:        $\hookrightarrow$  if possible; otherwise stop
5:     let  $\text{mtlsNonce}, \text{serverPubKey}$  such that  $m_{dec} \equiv \langle \text{mtlsNonce}, \text{serverPubKey} \rangle$  if possible; otherwise stop
6:     if  $\text{serverPubKey} \equiv s'.\text{keyMapping}[request.\text{host}]$  then  $\rightarrow$  Verify sender of mTLS nonce
7:       let  $\text{clientId} := reference[\text{client\_id}]$   $\rightarrow$  Note: If  $\text{client\_id} \notin reference$ , then  $reference[\text{client\_id}] \equiv \langle \rangle$ 
8:       let  $\text{pubKey} := reference[\text{pub\_key}]$   $\rightarrow$  See note for client ID above
9:       let  $s'.\text{mtlsCache} := s'.\text{mtlsCache} + \langle \rangle (\text{request}.\text{host}, \text{clientId}, \text{pubKey}, \text{mtlsNonce})$ 
10:    stop  $\langle \rangle, s'$ 
11:  if  $reference[responseTo] \equiv \text{CLIENT\_MANAGEMENT}$  then
12:     $\rightarrow$  Process a client information response [42, 43]. According requests are initiated by trigger messages, see Algorithm 9 and Section V.B.
13:    let  $\text{selectedAS} := reference[\text{selected\_AS}]$ 
14:    let  $\text{clientId} := s'.\text{asAccounts}[\text{selectedAS}][\text{client\_id}]$   $\rightarrow$   $\text{client\_id}$  cannot be changed (see Sec. 2.2 of RFC 7592 [43])
15:    if  $m.\text{status} \equiv 204 \wedge \text{request}.\text{method} \equiv \text{DELETE}$  then  $\rightarrow$  Client was deleted at AS (see Sec. 2.3 of RFC 7592 [43])
16:      let  $s'.\text{asAccounts} := s'.\text{asAccounts} - \text{selectedAS}$ 
17:      stop  $\langle \rangle, s'$ 
18:    if  $m.\text{body}[\text{client\_type}] \notin \{\text{mTLS\_mTLS}, \text{pkjwt\_mTLS}, \text{mTLS\_DPoP}, \text{pkjwt\_DPoP}\}$  then
19:      stop  $\rightarrow$  Invalid client type
20:    let  $\text{clientType} := m.\text{body}[\text{client\_type}]$ 
21:    if  $m.\text{body}[\text{jwtks}] \neq reference[\text{request}][\text{jwtks}]$  then
22:      stop  $\rightarrow$  AS changed client's  $\text{jwtks}$  value: abort client metadata update
23:    let  $\text{regClientUri} := m.\text{body}[\text{reg\_client\_uri}]$ 
24:    let  $\text{regAt} := m.\text{body}[\text{reg\_at}]$ 
25:    let  $s'.\text{asAccounts}[\text{selectedAS}] := [\text{client\_id}: \text{clientId}, \text{client\_type}: \text{clientType}, \text{reg\_at}: \text{regAt},$ 
26:       $\hookrightarrow \text{reg\_client\_uri}: \text{regClientUri}, \text{sign\_key}: reference[\text{sigKey}],$ 
27:       $\hookrightarrow \text{tls\_key}: reference[\text{tlsKey}], \text{grant\_types}: m.\text{body}[\text{grant\_types}]]$ 
28:    if  $\text{backchannel\_token\_delivery\_mode} \in m.\text{body}$  then
29:      let  $s'.\text{asAccounts}[\text{selectedAS}][\text{backchannel\_token\_delivery\_mode}] := m.\text{body}[\text{backchannel\_token\_delivery\_mode}]$ 
30:      if  $m.\text{body}[\text{backchannel\_token\_delivery\_mode}] \in \{\text{ping}, \text{push}\}$  then
31:        if  $\text{backchannel\_client\_notification\_endpoint} \notin m.\text{body}$  then
32:          stop
33:        let  $\text{clientNotificationEP} := m.\text{body}[\text{backchannel\_client\_notification\_endpoint}]$ 
34:        let  $s'.\text{asAccounts}[\text{selectedAS}][\text{backchannel\_client\_notification\_endpoint}] := \text{clientNotificationEP}$ 
35:      stop  $\langle \rangle, s'$ 
36:    let  $\text{sessionId} := reference[\text{session}]$ 
37:    let  $\text{session} := s'.\text{sessions}[\text{sessionId}]$ 
38:    let  $\text{selectedAS} := \text{session}[\text{selected\_AS}]$ 
39:     $\rightarrow$  Note: PREPARE_AND_SEND_INITIAL_REQUEST issues CONFIG, and REGISTRATION requests as required – once these get a response, we continue the PAR preparation by calling PREPARE_AND_SEND_INITIAL_REQUEST again.
40:  if  $reference[responseTo] \equiv \text{CONFIG}$  then
41:    if  $m.\text{body}[\text{issuer}] \neq \text{selectedAS}$  then  $\rightarrow$  Verify issuer identifier according to Sec. 3.3 of RFC 8414 [29]
42:    stop
43:    let  $s'.\text{oauthConfigCache}[\text{selectedAS}] := m.\text{body}$ 
44:    call PREPARE_AND_SEND_INITIAL_REQUEST( $\text{sessionId}, a, s'$ )
45:  else if  $reference[responseTo] \equiv \text{REGISTRATION}$  then
46:    if  $m.\text{body}[\text{client\_type}] \notin \{\text{mTLS\_mTLS}, \text{pkjwt\_mTLS}, \text{mTLS\_DPoP}, \text{pkjwt\_DPoP}\}$  then
47:      stop  $\rightarrow$  Invalid client type
48:    let  $\text{clientType} := m.\text{body}[\text{client\_type}]$ 
49:    let  $\text{clientId} := m.\text{body}[\text{client\_id}]$ 
50:    let  $\text{regClientUri} := m.\text{body}[\text{reg\_client\_uri}]$   $\rightarrow$  DCM endpoint of AS Sec. 3 of RFC 7592 [43]
51:    let  $\text{regAt} := m.\text{body}[\text{reg\_at}]$   $\rightarrow$  DCM bearer token Sec. 3 of RFC 7592 [43]
52:    if  $m.\text{body}[\text{jwtks}] \neq reference[\text{request}][\text{jwtks}]$  then
53:      stop  $\rightarrow$  AS changed client's  $\text{jwtks}$  value: abort registration
54:     $\rightarrow$  Note: The  $\text{jwtks}$  value contains the client's keys for client authentication as well as token sender constraining. Since the client might use different keys for different ASs (and change the keys used with a given AS), it needs to keep track of which keys to use with each AS.
55:    let  $s'.\text{asAccounts}[\text{selectedAS}] := [\text{client\_id}: \text{clientId}, \text{client\_type}: \text{clientType}, \text{reg\_at}: \text{regAt},$ 
56:       $\hookrightarrow \text{reg\_client\_uri}: \text{regClientUri}, \text{sign\_key}: reference[\text{sigKey}],$ 
57:       $\hookrightarrow \text{tls\_key}: reference[\text{tlsKey}], \text{grant\_types}: m.\text{body}[\text{grant\_types}]]$ 
```

\rightarrow Algorithm continues on next page.

```

50:   if backchannel_token_delivery_mode ∈ m.body then
51:     let s'.asAccounts[selectedAS][backchannel_token_delivery_mode] := m.body[backchannel_token_delivery_mode]
52:     if m.body[backchannel_token_delivery_mode] ∈ {ping, push} then
53:       if backchannel_client_notification_endpoint ∉ m.body then
54:         stop
55:         let clientNotificationEP := m.body[backchannel_client_notification_endpoint]
56:         let s'.asAccounts[selectedAS][backchannel_client_notification_endpoint] := clientNotificationEP
57:     call PREPARE_AND_SEND_INITIAL_REQUEST(reference[session], a, s')
58:   else if reference[responseTo] ≡ PAR then
59:     if reference[response_mode] ≡ jwt then
60:       → Client requested a signed authorization response
61:       let s'.sessions[sessionId][requested_signed_authz_response] := ⊤
62:       let requestUri := m.body[request_uri]
63:       let s'.sessions[sessionId][request_uri] := requestUri
64:       let clientId := session[client_id]
65:       let request := session[startRequest]
66:       → In the following, we construct the response to the initial request by some browser
67:       let authEndpoint := s'.oauthConfigCache[selectedAS][auth_ep]
68:       → The authorization endpoint URL may include query components, which must be retained while also ensuring that no parameter
69:       appears more than once (Sec. 3.1 of RFC 6749 [23]). However, following Sec. 4 of RFC 9126 [34] and Sec. 5 of RFC 9101 [47]
70:       closely could introduce duplicates. We opted to overwrite client_id and request_uri parameters if present.
71:       let authEndpoint.parameters[client_id] := clientId
72:       let authEndpoint.parameters[request_uri] := requestUri
73:       let headers := [Location: authEndpoint]
74:       let headers[Set-Cookie] := [(_Host, sessionId): ⟨sessionId, ⊤, ⊤, ⊤⟩]
75:       let response := encs(⟨HTTPResp, request[message].nonce, 303, headers, ⟨⟩⟩, request[key])
76:       let leakAuthZReq ← {⊤, ⊥} → We assume that the authorization request, in particular request_uri and client_id, may
77:       leak to the attacker, see [13].
78:     if leakAuthZReq ≡ ⊤ then
79:       let leak := ⟨LEAK, authEndpoint⟩
80:       let leakAddress ← IPs
81:       stop ⟨⟨request[sender], request[receiver], response⟩, ⟨leakAddress, request[receiver], leak⟩⟩, s'
82:     else
83:       stop ⟨⟨request[sender], request[receiver], response⟩⟩, s'
84:   else if reference[responseTo] ≡ TOKEN then
85:     let useAccessTokenNow := ⊤
86:     if session[scope] ≡ openid then → Non-deterministically decide whether to use the AT or check the ID token (if requested)
87:       let useAccessTokenNow ← {⊤, ⊥}
88:     if useAccessTokenNow ≡ ⊤ then
89:       call USE_ACCESS_TOKEN(reference[session], m.body[access_token], request.host, a, s')
90:     let selectedAsTokenEp := s'.oauthConfigCache[selectedAS][token_ep]
91:     if request.host ≠ selectedAsTokenEp.host then
92:       stop → Verify sender of HTTPS response is the expected AS (see [46, Sec. 3.1.3.7])
93:     call CHECK_ID_TOKEN(reference[session], m.body[id_token], s')
94:   else if reference[responseTo] ≡ RESOURCE_USAGE then
95:     → Construct response to browser's request to the client's redirect endpoint (with the retrieved resource as payload)
96:     let expectSignedResponse ← {⊤, ⊥} → Choose whether to expect a signed resource response
97:     let s'.sessions[sessionId][expect_signed_resource_res] := expectSignedResponse
98:     if expectSignedResponse ≡ ⊤ then → Check whether client expects a signed response
99:       if hash(m.body) ≠ m.headers[Content-Digest] then
100:         stop → Content-digest is required by FAPI 2.0 Message Signing [22, Sec. 5.6.2.2]
101:         let coveredComponents := m.headers[Signature-Input][res]
102:         let rsDom := request.host → RS to which the resource request was sent
103:         let pubKey := s'.rsSigKeys[rsDom]
104:         let signerSignatureBase := extractmsg(m.headers[Signature][res])
105:         if @status ∉ coveredComponents.1 ∨ content-digest ∉ coveredComponents.1 ∨
106:            ↪ coveredComponents.2[tag] ≠ fapi-2-response then
107:           stop → See [22, Sec. 5.6.2.2], these components must be present
108:         if signerSignatureBase.2[tag] ≠ fapi-2-request ∨ keyid ∉ signerSignatureBase.2 then
109:           stop
110:         for component ∈ coveredComponents.1 do
111:           let isComponentEqual := IS_COMPONENT_EQUAL(m, request, signerSignatureBase, component)
112:           if isComponentEqual ≠ ⊤ then
113:             stop
114:     → Algorithm continues on next page.

```

```

106:   → If we make it here, the response signature base matches the actual response data.
107:   if  $pubKey \equiv \langle \rangle \vee \text{checksig}(m.\text{headers}[\text{Signature}][\text{res}], pubKey) \neq \top$  then
108:     stop → Invalid public key/message or signature does not verify
109:   let  $resource := m.\text{body}[\text{resource}]$ 
110:   let  $s'.\text{sessions}[\text{sessionId}][\text{resourceServer}] := resource$  → Store received resource
111:   let  $s'.\text{sessions}[\text{sessionId}][\text{resourceServer}] := request.\text{host}$  → Store the domain of the RS
112:   if  $session[\text{cibaFlow}] \equiv \perp$  then → Send the resource as a response to the redirection endpoint request.
113:     let  $request := session[\text{redirectEpRequest}]$  → Data on browser's request to client's redirect endpoint
114:     let  $m' := \text{enc}_s(\langle \text{HTTPResp}, request[\text{message}].\text{nonce}, 200, \langle \rangle, resource \rangle, request[\text{key}])$ 
115:     stop  $\langle \langle request[\text{sender}], request[\text{receiver}], m' \rangle \rangle, s'$ 
116:   else → Wait for the browser to send a request with the login session id, see Line 51 of Algorithm 2
117:     stop  $\langle \rangle, s'$ 
118:   else if  $reference[\text{responseTo}] \equiv \text{DPOP\_NONCE}$  then
119:     let  $dpopNonce := m.\text{body}[\text{nonce}]$ 
120:     let  $rsDomain := request.\text{host}$ 
121:     let  $s'.\text{dpopNonces}[\text{rsDomain}] := s'.\text{dpopNonces}[\text{rsDomain}] + \langle \rangle dpopNonce$ 
122:     stop  $\langle \rangle, s'$ 
123:   else if  $reference[\text{responseTo}] \equiv \text{CIBA\_AUTH\_REQ}$  then
124:     let  $authnReqId := m.\text{body}[\text{auth\_req\_id}]$ 
125:     let  $s'.\text{sessions}[\text{sessionId}][\text{auth\_req\_id}] := authnReqId$  → Store received request identifier
126:     → If the client has registered the poll delivery mode, it can start polling at the token endpoint
127:     if  $s'.\text{asAccounts}[\text{selectedAS}][\text{backchannel\_token\_delivery\_mode}] \equiv \text{poll}$  then
128:       let  $s'.\text{sessions}[\text{sessionId}][\text{start\_polling}] := \top$  → Client can start polling
129:       stop  $\langle \rangle, s'$ 
130:   stop

```

Algorithm 4 Relation of a Client R^c – Request to token endpoint.

```
1: function SEND_TOKEN_REQUEST(sessionId, code, a, s')
2:   let session := s'.sessions[sessionId]
3:   if code_verifier  $\notin$  session then
4:     stop
5:   let pkceVerifier := session[code_verifier]
6:   let selectedAS := session[selected_AS]
7:   let headers := []
8:   let body := [grant_type: authorization_code, code: code, redirect_uri: session[redirect_uri]]
9:   let body[code_verifier] := pkceVerifier → add PKCE Code Verifier (RFC 7636 [44], Section 4.5)
10:  let clientId := s'.asAccounts[selectedAS][client_id]
11:  let clientType := s'.asAccounts[selectedAS][client_type]
12:  let clientSignKey := s'.asAccounts[selectedAS][sign_key] → Used in private_key_jwt authentication and DPoP
13:  let oauthConfig := s'.oauthConfigCache[selectedAS]
14:  let tokenEndpoint := oauthConfig[token_ep]
    → Client Authentication:
15:  if clientType  $\in$  {mTLS_mTLS, mTLS_DPoP} then → mTLS client authentication
16:    let body[client_id] := clientId → RFC 8705 [7] mandates client_id when using mTLS authentication
17:    let mtlsNonce such that  $\langle$ tokenEndpoint.host, clientId,  $\langle$  $\rangle$ , mtlsNonce $\rangle \in$  s'.mtlsCache if possible; otherwise stop
18:    let authData := [TLS_AuthN: mtlsNonce]
19:    let s'.mtlsCache := s'.mtlsCache  $-^{\langle}$   $\langle$ tokenEndpoint.host, clientId,  $\langle$  $\rangle$ , mtlsNonce $\rangle$ 
20:  else if clientType  $\in$  {pkjwt_mTLS, pkjwt_DPoP} then → private_key_jwt client authentication
21:    let jwt := [iss: clientId, sub: clientId, aud: selectedAS]
22:    let jws := sig(jwt, clientSignKey)
23:    let authData := [client_assertion: jws]
24:  else
25:    stop → Invalid client type
    → Sender Constraining:
26:  if clientType  $\equiv$  mTLS_mTLS then → mTLS sender constraining (same nonce as for mTLS authN)
27:    let mtlsNonce := authData[TLS_AuthN]
28:    let body[TLS_binding] := mtlsNonce
29:  else if clientType  $\equiv$  pkjwt_mTLS then → mTLS sender constraining (fresh mTLS nonce)
30:    let mtlsNonce such that  $\langle$ tokenEndpoint.host, clientId,  $\langle$  $\rangle$ , mtlsNonce $\rangle \in$  s'.mtlsCache if possible; otherwise stop
31:    let s'.mtlsCache := s'.mtlsCache  $-^{\langle}$   $\langle$ tokenEndpoint.host, clientId,  $\langle$  $\rangle$ , mtlsNonce $\rangle$ 
32:    let body[TLS_binding] := mtlsNonce
33:  else → Sender constraining using DPoP
34:    let htu := tokenEndpoint
35:    let htu.parameters :=  $\langle$  $\rangle$  → [15, Sec. 4.2]: without query
36:    let htu.fragment :=  $\perp$  → [15, Sec. 4.2]: without fragment
37:    let dpopJwt := [headers: [jwk: pub(clientSignKey))]
38:    let dpopJwt[payload] := [htm: POST, htu: htu]
39:    let dpopProof := sig(dpopJwt, clientSignKey)
40:    let headers[DPoP] := dpopProof → add DPoP header; the dpopJwt can be extracted with the extractmsg() function
41:  let body := body  $+^{\langle}$  authData
42:  let message :=  $\langle$ HTTPReq,  $\nu_2$ , POST, tokenEndpoint.host, tokenEndpoint.path, tokenEndpoint.parameters, headers, body $\rangle$ 
43:  call HTTPS_SIMPLE_SEND([responseTo: TOKEN, session: sessionId], message, a, s')
```

Algorithm 5 Relation of a Client R^c – Request to token endpoint for CIBA flows.

```
1: function SEND_CIBA_TOKEN_REQUEST(sessionId, a, s')
2:   let session := s'.sessions[sessionId]
3:   let selectedAS := session[selected_AS]
4:   let authnReqId := session[auth_req_id]
5:   let headers := []
6:   let body := [grant_type: urn:openid:params:grant-type:ciba, auth_req_id: authnReqId]
7:   let clientId := s'.asAccounts[selectedAS][client_id]
8:   let clientType := s'.asAccounts[selectedAS][client_type]
9:   let clientSignKey := s'.asAccounts[selectedAS][sign_key] → Used in private_key_jwt authentication and DPoP
10:  let oauthConfig := s'.oauthConfigCache[selectedAS]
11:  let tokenEndpoint := oauthConfig[token_ep]
    → Client Authentication:
12:  if clientType ∈ {mTLS_mTLS, mTLS_DPoP} then → mTLS client authentication
13:    let body[client_id] := clientId → RFC 8705 [7] mandates client_id when using mTLS authentication
14:    let mtlsNonce such that ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
15:    let authData := [TLS_AuthN: mtlsNonce]
16:    let s'.mtlsCache := s'.mtlsCache - ⟨⟩ ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩
17:  else if clientType ∈ {pkjwt_mTLS, pkjwt_DPoP} then → private_key_jwt client authentication
18:    let jwt := [iss: clientId, sub: clientId, aud: selectedAS]
19:    let jws := sig(jwt, clientSignKey)
20:    let authData := [client_assertion: jws]
21:  else
22:    stop → Invalid client type
    → Sender Constraining:
23:  if clientType ≡ mTLS_mTLS then → mTLS sender constraining (same nonce as for mTLS authN)
24:    let mtlsNonce := authData[TLS_AuthN]
25:    let body[TLS_binding] := mtlsNonce
26:  else if clientType ≡ pkjwt_mTLS then → mTLS sender constraining (fresh mTLS nonce)
27:    let mtlsNonce such that ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
28:    let s'.mtlsCache := s'.mtlsCache - ⟨⟩ ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩
29:    let body[TLS_binding] := mtlsNonce
30:  else → Sender constraining using DPoP
31:    let htu := tokenEndpoint
32:    let htu.parameters := ⟨⟩ → [15, Sec. 4.2]: without query
33:    let htu.fragment := ⊥ → [15, Sec. 4.2]: without fragment
34:    let dpopJwt := [headers: [jwk: pub(clientSignKey)]]
35:    let dpopJwt[payload] := [htm: POST, htu: htu]
36:    let dpopProof := sig(dpopJwt, clientSignKey)
37:    let headers[DPoP] := dpopProof → add DPoP header; the dpopJwt can be extracted with the extractmsg() function
38:  let body := body + ⟨⟩ authData
39:  let message := ⟨HTTPReq,  $\nu_2$ , POST, tokenEndpoint.host, tokenEndpoint.path, tokenEndpoint.parameters, headers, body⟩
40:  call HTTPS_SIMPLE_SEND([responseTo: TOKEN, session: sessionId], message, a, s')
```

Algorithm 6 Relation of a Client R^c – Using the access token.

```
1: function USE_ACCESS_TOKEN(sessionId, token, tokenEPDomain, a, s')
2:   let session := s'.sessions[sessionId]
3:   let selectedAS := session[selected_AS]
4:   let rsDomain ← Doms → This domain may or may not belong to a “real” RS. If it belongs to the attacker, this request leaks the
   → Note: All paths except the mTLS and DPoP preparation endpoints are resource paths at the RS.
   access token (but no mTLS nonce, nor a DPoP proof for an honest server).
5:   let resourceID ←  $\mathbb{S}$  such that resourceID  $\notin$  {/mTLS-prepare, /DPoP-nonce}
6:   let url := ⟨URL, S, rsDomain, resourceID, ⟨⟩,  $\perp$ ⟩
7:   if s'.resourceASMapping[rsDomain][resourceID]  $\neq$  tokenEPDomain then
8:     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.
9:   let clientType := s'.asAccounts[selectedAS][client_type]
10:  let clientId := s'.asAccounts[selectedAS][client_id]
11:  let body := []
12:  if clientType ∈ {mTLS_mTLS, pkjwt_mTLS} then → mTLS sender constraining
13:    let mtlsNonce such that ⟨rsDomain, ⟨⟩, pubKey, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
14:    let body[TLS_binding] := mtlsNonce → This nonce is not necessarily associated with the same of the client’s keys as the
   access token. In such a case, the RS will reject this request and the client has to try
   again.
15:    let headers := [Authorization: [Bearer: token]] → FAPI 2.0 mandates to send access token in header
16:    let s'.mtlsCache := s'.mtlsCache  $-^{\langle \rangle}$  ⟨rsDomain, ⟨⟩, pubKey, mtlsNonce⟩
17:  else if clientType ∈ {mTLS_DPoP, pkjwt_DPoP} then → DPoP sender constraining
18:    let privKey := s'.asAccounts[selectedAS][sign_key] → get private signing key registered with selectedAS
19:    let dpopNonce such that dpopNonce ∈ s'.dpopNonces[rsDomain] if possible; otherwise stop
20:    let s'.dpopNonces[rsDomain] := s'.dpopNonces[rsDomain]  $-^{\langle \rangle}$  dpopNonce
21:    let htu := url
22:    let htu.parameters := ⟨⟩ → [15, Sec. 4.2]: without query
23:    let htu.fragment :=  $\perp$  → [15, Sec. 4.2]: without fragment
24:    let dpopJwt := [headers: [jwk: pub(privKey))]
25:    let dpopJwt[payload] := [htm: POST, htu: htu, ath: hash(token), nonce: dpopNonce]
26:    let dpopProof := sig(dpopJwt, privKey)
27:    let headers := [Authorization: [DPoP: token]] → See [15, Sec. 7.1]
28:    let headers[DPoP] := dpopProof → add DPoP header; the dpopJwt can be extracted with the extractmsg() function
29:  let signRequest ← { $\top$ ,  $\perp$ } → Choose whether to sent a signed resource request
30:  if signRequest  $\equiv$   $\top$  then
31:    let clientSignKey := s'.asAccounts[selectedAS][sign_key]
32:    let headers[Content-Digest] := hash(body) → See [22, Sec. 5.6.1.1 No. 8]
33:    let coveredComponents := {⟨@method, ⟨⟩⟩, ⟨@target-uri, ⟨⟩⟩, ⟨authorization, ⟨⟩⟩, ⟨content-digest, ⟨⟩⟩},
   ↪ [tag: fapi-2-request, keyid: pub(clientSignKey))] → See [22, Sec. 5.6.1.1]
34:    let signatureBase := [⟨@method, ⟨⟩⟩: POST, ⟨@target-uri, ⟨⟩⟩: url, ⟨authorization, ⟨⟩⟩: headers[Authorization],
   ↪ ⟨content-digest, ⟨⟩⟩: headers[Content-Digest]]
35:    if DPoP ∈ headers then
36:      let coveredComponents.1 := coveredComponents.1  $+^{\langle \rangle}$  ⟨dpop, ⟨⟩⟩ → See [22, Sec. 5.6.1.1 No. 7]
37:      let signatureBase[⟨dpop, ⟨⟩⟩] := headers[DPoP]
38:      let signatureBase := signatureBase  $+^{\langle \rangle}$  coveredComponents.2 → Signature parameters, cf. [2, Sec. 2.5]
39:      let headers[Signature] := [req: sig(signatureBase, clientSignKey)]
40:      let headers[Signature-Input] := [req: coveredComponents]
41:    let s'.sessions[sessionId][signed_resource_req] := signRequest
42:    let message := ⟨HTTPReq,  $\nu_3$ , POST, url.domain, url.path, ⟨⟩, headers, body⟩
43:    call HTTPS_SIMPLE_SEND([responseTo: RESOURCE_USAGE, session: sessionId], message, a, s')
```

Algorithm 7 Relation of a Client R^c – Check ID Token and log user in at c .

```
1: function CHECK_ID_TOKEN( $sessionId, idToken, s'$ ) → Check ID Token validity and create service session.
2:   let  $session := s'.sessions[sessionId]$  → Retrieve session data.
3:   let  $selectedAS := session[selected\_AS]$ 
4:   let  $oauthConfig := s'.oauthConfigCache[selectedAS]$  → Retrieve configuration for user-selected AS.
5:   let  $clientInfo := s'.asAccounts[selectedAS]$  → Retrieve client info used at that AS.
6:   let  $data := extractmsg(idToken)$  → Extract contents of signed ID Token.
   → The following ID token checks are mandated by [46, 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 85 of Algorithm 3).
7:   if  $data[iss] \neq selectedAS$  then
8:     stop → Check the issuer; note that previous checks ensure  $oauthConfig[issuer] \equiv selectedAS$ 
9:   if  $data[aud] \neq clientInfo[client\_id]$  then
10:    stop → Check the audience against own client id.
11:  if  $nonce \in session \wedge data[nonce] \neq session[nonce]$  then
12:    stop → If a nonce was used, check its value.
13:  let  $s'.sessions[sessionId][loggedInAs] := \langle selectedAS, data[sub] \rangle$  → User is now logged in. Store user identity and issuer of
   ID token.
14:  let  $s'.sessions[sessionId][serviceSessionId] := \nu_4$  → Choose a new service session id.
15:  if  $session[cibaFlow] \equiv \perp$  then → Send a response to the request to the redirection endpoint with the service session id.
16:    let  $request := session[redirectEpRequest]$  → 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 2).
17:    let  $headers[Set-Cookie] := [serviceSessionId: \langle \nu_4, \top, \top, \top \rangle]$  → Create a cookie containing the service session id, effec-
   tively logging the user identified by  $data[sub]$  in at this
   client.
18:    let  $m' := enc_s(\langle HTTPResp, request[message].nonce, 200, headers, ok \rangle, request[key])$ 
19:    stop  $\langle \langle request[sender], request[receiver], m' \rangle, s' \rangle$ 
20:  else → Wait for the browser to send a request with the login session id, see Line 51 of Algorithm 2
21:  stop  $\langle \rangle, s'$ 
```

Algorithm 8 Relation of a Client R^c – Prepare and send pushed authorization request or CIBA authentication request.

```
1: function PREPARE_AND_SEND_INITIAL_REQUEST( $sessionId, a, s'$ )
2:   let  $redirectUris := \{\langle URL, S, d, /redirect\_ep, \rangle, \perp\} \mid d \in \text{dom}(c)\}$   $\rightarrow$  Set of redirect URIs for all domains of  $c$ .
3:   let  $redirectUri \leftarrow redirectUris$   $\rightarrow$  Select a (potentially) different redirect URI for each authorization request
4:   let  $session := s'.sessions[sessionId]$ 
5:   let  $selectedAS := session[selected\_AS]$   $\rightarrow$  AS selected by the user at the beginning of the flow.
    $\rightarrow$  Check whether the client needs to fetch AS metadata first and do so if required.
6:   if  $selectedAS \notin s'.oauthConfigCache$  then
7:     let  $path \leftarrow \{/.well\_known/openid-configuration, /.well\_known/oauth-authorization-server\}$ 
8:     let  $message := \langle \text{HTTPReq}, \nu_5, \text{GET}, selectedAS, path, \rangle, \langle \rangle, \langle \rangle$ 
9:     call HTTPS_SIMPLE_SEND( $[responseTo: \text{CONFIG}, session: sessionId], message, a, s'$ )
10:  let  $oauthConfig := s'.oauthConfigCache[selectedAS]$ 
11:  if  $selectedAS \notin s'.asAccounts$  then  $\rightarrow c$  not yet registered with  $selectedAS$  – Dynamic Client Registration (see RFC 7591 [42])
12:    let  $url := oauthConfig[reg\_ep]$ 
13:    let  $signingKey := \nu_{cliSignK}$   $\rightarrow$  Generate signing key (pair) to use with  $selectedAS$ 
14:    let  $tlsKey := \nu_{cliTlsK}$   $\rightarrow$  Generate mTLS key (pair) to use with  $selectedAS$  (see also Appendix A-G)
15:    let  $jwtks := \{\langle use: sig, val: pub(signingKey) \rangle, \langle use: TLS, val: pub(tlsKey) \rangle\}$ 
16:    let  $regData := [\text{redirect\_uris}: redirectUris], jwtks: jwtks$ 
17:    let  $cibaDeliveryMode \leftarrow \{\text{poll}, \text{ping}, \text{push}\}$ 
18:    let  $regData[\text{backchannel\_token\_delivery\_mode}] := cibaDeliveryMode$ 
19:    if  $cibaDeliveryMode \equiv \text{ping} \vee cibaDeliveryMode \equiv \text{push}$  then
20:      let  $regData[\text{backchannel\_client\_notification\_endpoint}] \leftarrow \{\langle URL, S, d, /ciba\_notif\_ep, \rangle, \perp\} \mid d \in \text{dom}(c)\}$ 
21:    if  $cibaDeliveryMode \equiv \text{ping} \vee cibaDeliveryMode \equiv \text{poll}$  then
22:      let  $regData[\text{grant\_types}] := \langle \text{authorization\_code}, \text{urn:openid:params:grant-type:ciba} \rangle$ 
23:    else
24:      let  $regData[\text{grant\_types}] := \langle \text{authorization\_code} \rangle$ 
25:    let  $message := \langle \text{HTTPReq}, \nu_5, \text{POST}, url.host, url.path, url.parameters, \rangle, regData$ 
26:    call HTTPS_SIMPLE_SEND( $[responseTo: \text{REGISTRATION}, session: sessionId, sigKey: signingKey, tlsKey: tlsKey],$ 
    $\hookrightarrow message, a, s'$ )
    $\rightarrow$  Construct pushed authorization request or CIBA authentication request
27:  if  $session[cibaFlow] \equiv \top$  then
28:    let  $requestEndpoint := oauthConfig[\text{backchannel\_authentication\_endpoint}]$ 
29:  else
30:    let  $requestEndpoint := oauthConfig[par\_ep]$ 
31:  let  $clientId := s'.asAccounts[selectedAS][client\_id]$ 
32:  let  $clientType := s'.asAccounts[selectedAS][client\_type]$ 
33:  let  $clientSignKey := s'.asAccounts[selectedAS][sign\_key]$ 
34:  if  $clientType \in \{\text{mTLS\_mTLS}, \text{mTLS\_DPoP}\}$  then  $\rightarrow$  mTLS client authentication
35:    let  $mtlsNonce$  such that  $\langle requestEndpoint.host, clientId, \rangle, mtlsNonce \in s'.mtlsCache$  if possible; otherwise stop
36:    let  $authData := [\text{TLS\_AuthN}: mtlsNonce]$ 
37:    let  $s'.mtlsCache := s'.mtlsCache - \langle requestEndpoint.host, clientId, \rangle, mtlsNonce$ 
38:  else if  $clientType \in \{\text{pkjwt\_mTLS}, \text{pkjwt\_DPoP}\}$  then  $\rightarrow$  private_key_jwt client authentication
39:    let  $jwt := [\text{iss}: clientId, \text{sub}: clientId, \text{aud}: selectedAS]$ 
40:    let  $jws := sig(jwt, clientSignKey)$ 
41:    let  $authData := [\text{client\_assertion}: jws]$ 
42:  if  $session[cibaFlow] \equiv \top$  then
43:    let  $requestData := [\text{client\_id}: clientId, \text{scope}: \text{openid}, \text{login\_hint}: session[selected\_identity],$ 
    $\hookrightarrow \text{binding\_message}: session[\text{binding\_message}]$ 
44:    if  $cibaDeliveryMode \equiv \text{ping}$  then
45:      let  $requestData[\text{client\_notification\_token}] := \nu_{cibaNotifToken}$ 
46:  else
47:    let  $pkceVerifier := \nu_{pkce}$   $\rightarrow$  Fresh random value
48:    let  $pkceChallenge := \text{hash}(pkceVerifier)$ 
49:    let  $requestData := [\text{response\_type}: \text{code}, \text{code\_challenge\_method}: \text{S256}, \text{client\_id}: clientId,$ 
    $\hookrightarrow \text{redirect\_uri}: redirectUri, \text{code\_challenge}: pkceChallenge]$ 
50:    let  $useOidc \leftarrow \{\top, \perp\}$   $\rightarrow$  Use of OIDC is optional
51:    if  $useOidc \equiv \top$  then
52:      let  $requestData[\text{scope}] := \text{openid}$ 
53:    let  $s'.sessions[sessionId][code\_verifier] := pkceVerifier$   $\rightarrow$  Store PKCE randomness in state
 $\rightarrow$  Algorithm continues on next page.
```

```

54: let  $s'.sessions[sessionId] := s'.sessions[sessionId] + \langle \rangle requestData$ 
55: let  $requestData := requestData + \langle \rangle authData$ 
56: if  $session[cibaFlow] \equiv \perp$  then
57:   let  $requestSignedResponse \leftarrow \{\top, \perp\}$   $\rightarrow$  Choose whether to request a signed authorization response (see also Section V-C)
58:   if  $requestSignedResponse \equiv \top$  then
59:     let  $requestData[response\_mode] := \text{jwt}$   $\rightarrow$  Request signed authorization response (cf. JARM [32, Sec. 2.3] and [22, Sec. 5.4.2 No. 1])
     $\rightarrow$  Note: Following the recommendation in [46, Sec. 3.1.2.1], we do not set a response_mode for “regular” requests.
60:   let  $signPAR \leftarrow \{\top, \perp\}$   $\rightarrow$  Choose whether to use a signed authorization request (see also Section V-C)
61:   if  $signPAR \equiv \top$  then
62:     let  $requestData[aud] := selectedAS$   $\rightarrow$  See [22, Sec. 5.3.2 No. 2]
63:     let  $body := \text{sig}(requestData, clientSignKey)$   $\rightarrow$  Sign authorization request (FAPI 2.0 Message Signing)
64:   else
65:     let  $body := requestData$ 
66:   else
67:     let  $body := requestData$ 
68:   let  $authReq := \langle \text{HTTPReq}, \nu_{authReqNonce}, \text{POST}, requestEndpoint.host, requestEndpoint.path, requestEndpoint.parameters, \langle \rangle, body \rangle$ 
69:   if  $session[cibaFlow] \equiv \top$  then
70:     call HTTPS_SIMPLE_SEND([responseTo: CIBA_AUTH_REQ, session:  $sessionId$ ],  $authReq, a, s'$ )
71:   else
72:     call HTTPS_SIMPLE_SEND([responseTo: PAR, session:  $sessionId$ , response_mode:  $requestData[response\_mode]$ ]
     $\hookrightarrow$  ,  $authReq, a, s'$ )

```

Algorithm 9 Relation of a Client R^c – Handle trigger events.

```
1: function PROCESS_TRIGGER( $a, s'$ )
2:   let  $action \leftarrow \{MTLS\_PREPARE\_AS, MTLS\_PREPARE\_RS, MTLS\_PREPARE\_MISCONFIGURED\_TOKEN\_EP,$ 
    $\hookrightarrow GET\_DPOP\_NONCE, CHANGE\_CLIENT\_CONFIG, CIBA\_POLL\_TOKEN\_EP, CIBA\_START\_FLOW\}$ 
3:   switch  $action$  do
4:     case  $MTLS\_PREPARE\_AS$ 
5:       let  $server \leftarrow \text{Doms such that } server \in s'.asAccounts \text{ if possible; otherwise stop}$ 
6:       let  $asAcc := s'.asAccounts[server]$ 
7:       let  $clientId := asAcc[client\_id]$ 
8:       let  $body := [client\_id: clientId]$ 
9:       let  $message := \langle HTTPReq, \nu_{mTLS}, GET, server, /MTLS-prepare, \langle \rangle, \langle \rangle, body \rangle$ 
10:      call  $HTTPS\_SIMPLE\_SEND([responseTo: MTLS, client\_id: clientId], message, a, s')$ 
11:     case  $MTLS\_PREPARE\_RS$ 
12:        $\rightarrow$  Non-deterministically contact some RS to get an mTLS nonce for mTLS access token sender constraining (for an access
13:         token issued by  $selectedAS$ , i.e., that token is bound to the mTLS key registered with  $selectedAS$ ).
14:       let  $resourceServer \leftarrow \text{Doms} \rightarrow$  Note: This may or may not be a “real” RS.
15:       let  $selectedAS \leftarrow \text{Doms such that } selectedAS \in s'.asAccounts \text{ if possible; otherwise stop}$ 
16:       let  $mTlsPrivKey := s'.asAccounts[selectedAS][tls\_key]$ 
17:       let  $pubKey := \text{pub}(mTlsPrivKey)$ 
18:       let  $body := [pub\_key: pubKey]$ 
19:       let  $message := \langle HTTPReq, \nu_{mTLS}, GET, resourceServer, /MTLS-prepare, \langle \rangle, \langle \rangle, body \rangle$ 
20:       call  $HTTPS\_SIMPLE\_SEND([responseTo: MTLS, pub\_key: pubKey], message, a, s')$ 
21:     case  $MTLS\_PREPARE\_MISCONFIGURED\_TOKEN\_EP$ 
22:        $\rightarrow$  This case allows the client to retrieve mTLS nonces from attacker-controlled servers and subsequently make requests
23:         to such servers. Without this case, the model would not capture attacks in which the client talks to attacker-controlled
24:         endpoints protected by mTLS.
25:       let  $server \leftarrow \text{Doms such that } server \in s'.asAccounts \text{ if possible; otherwise stop}$ 
26:       let  $asAcc := s'.asAccounts[server]$ 
27:       let  $clientId := asAcc[client\_id]$ 
28:       let  $host \leftarrow \text{Doms} \rightarrow$  Non-deterministically choose the domain instead of sending to the correct AS
29:       let  $body := [client\_id: clientId]$ 
30:       let  $message := \langle HTTPReq, \nu_{mTLS}, GET, host, /MTLS-prepare, \langle \rangle, \langle \rangle, body \rangle$ 
31:       call  $HTTPS\_SIMPLE\_SEND([responseTo: MTLS, client\_id: clientId], message, a, s')$ 
32:     case  $GET\_DPOP\_NONCE$ 
33:        $\rightarrow$  Our client uses DPoP server-provided nonces at the RS. The RS model offers a special endpoint to retrieve nonces.
34:       let  $resourceServer \leftarrow \text{Doms} \rightarrow$  Note: This may or may not be a “real” RS.
35:       let  $message := \langle HTTPReq, \nu_{DPoPReq}, GET, resourceServer, /DPoP-nonce, \langle \rangle, \langle \rangle, \langle \rangle \rangle$ 
36:       call  $HTTPS\_SIMPLE\_SEND([responseTo: DPOP\_NONCE], message, a, s')$ 
37:     case  $CHANGE\_CLIENT\_CONFIG \rightarrow$  Use dynamic client management at AS (see RFC 7592 [43])
38:        $\rightarrow$  Randomly select one of the ASs this client is registered with.
39:       let  $selectedAS \leftarrow \text{Doms such that } selectedAS \in s'.asAccounts \text{ if possible; otherwise stop}$ 
40:       let  $regClientUri := s'.asAccounts[selectedAS][reg\_client\_uri] \rightarrow$  Client management URI
41:       let  $regAt := s'.asAccounts[selectedAS][reg\_at] \rightarrow$  Client management access token
42:       let  $authHeader := [Authorization: \langle Bearer, regAt \rangle]$ 
43:       let  $DCMAction \leftarrow \{UPDATE, DELETE\} \rightarrow$  Randomly select a client management action
44:       switch  $DCMAction$  do
45:         case  $DELETE \rightarrow$  Delete client at AS, see Sec. 2.3 of RFC 7592 [43]
46:           let  $message := \langle HTTPReq, \nu_{DELReq}, DELETE, regClientUri.host, regClientUri.path, \langle \rangle, authHeader, \langle \rangle \rangle$ 
47:           call  $HTTPS\_SIMPLE\_SEND([responseTo: CLIENT\_MANAGEMENT], message, a, s')$ 
48:         case  $UPDATE \rightarrow$  Update client configuration at AT, see Sec. 2.2 of RFC 7592 [43]
49:            $\rightarrow$  Generate fresh key pairs and update client keys at AS (see also Section V-B). Note that following our simplified
50:             model of mTLS, the client metadata includes a public key instead of a distinguished name for mTLS (or similar,
51:             see Sec. 2.1.2 of RFC 8705 [7]).
52:           let  $redirectUris := \{\langle URL, S, d, /redirect\_ep, \langle \rangle, \perp \rangle \mid d \in \text{dom}(c)\} \rightarrow$  Set of redirect URIs for all domains of  $c$ 
53:           let  $newSigningKey := \nu_{cliSignK}$ 
54:           let  $newTLSKey := \nu_{cliTlsK}$ 
55:           let  $newJwks := \langle [use: sig, val: \text{pub}(newSigningKey)], [use: TLS, val: \text{pub}(newTLSKey)] \rangle$ 
56:           let  $body := [client\_id: clientId, jwks: newJwks, redirect\_uris: \langle redirectUris \rangle]$ 
57:            $\rightarrow$  Algorithm continues on next page.
```

```

47:   → Continuing the UPDATE case:
48:   let cibaDeliveryMode ← {poll, ping, push}
49:   let body[backchannel_token_delivery_mode] := cibaDeliveryMode
50:   if cibaDeliveryMode ≡ ping ∨ cibaDeliveryMode ≡ push then
51:     let body[backchannel_client_notification_endpoint] ←
52:       ↪ {⟨URL, S, d, /ciba_notif_ep, ⟨⟩, ⊥⟩ | d ∈ dom(c)}
53:   if cibaDeliveryMode ≡ ping ∨ cibaDeliveryMode ≡ poll then
54:     let body[grant_types] := ⟨authorization_code, urn:openid:params:grant-type:ciba⟩
55:   else
56:     let body[grant_types] := ⟨authorization_code⟩
57:   let message := ⟨HTTPReq, νPUTReq, PUT, regClientUri.host, regClientUri.path, ⟨⟩, authHeader, body⟩
58:   call HTTPS_SIMPLE_SEND([responseTo: CLIENT_MANAGEMENT, selected_AS: selected_AS,
59:     ↪ sigKey: newSigningKey, tlsKey: newTLSKey], message, a, s')
60: case CIBA_POLL_TOKEN_EP → Poll Token Endpoint
61: let sessionId such that sessionId ∈ s'.sessions
62:   ↪ ∧ s'.asAccounts[s'.sessions[sessionId][selected_AS]][backchannel_token_delivery_mode] ≡ poll
63:   ↪ ∧ s'.sessions[sessionId][start_polling] ≡ ⊤ if possible; otherwise stop
64: call SEND_CIBA_TOKEN_REQUEST(sessionId, a, s') → Send a token request
65: case CIBA_START_FLOW → Start the flow by sending the CIBA authentication request
66: let sessionId, a such that ⟨sessionId, a⟩ ∈ ∅ s'.pendingCIBARequests if possible; otherwise stop
67: let s'.pendingCIBARequests := s'.pendingCIBARequests - ∅ ⟨sessionId, a⟩
68: call PREPARE_AND_SEND_INITIAL_REQUEST(sessionId, a, s') → Start a CIBA flow (see Algorithm 8)
69: stop

```

Algorithm 10 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.

- 1: let *switch* ← {auth, link} → Non-deterministically decide whether to start a login flow or to follow some link.
- 2: if *switch* ≡ auth then → Start login flow.
 - 3: let *url* := GETURL(*tree*, *docnonce*) → Retrieve URL of current document.
 - 4: let *id* ← *ids* → Retrieve one of user's identities.
 - 5: let *as* := *id*.domain → Extract domain of AS from chosen *id*.
 - 6: let *url'* := ⟨URL, S, *url*.host, /startLogin, ⟨⟩, ⊥⟩ → Assemble request URL.
 - 7: let *command* := ⟨FORM, *url'*, POST, *as*, ⊥⟩ → Post a form including the selected AS to the client.
 - 8: stop (*s*, *cookies*, *localStorage*, *sessionStorage*, *command*) → Finish script's run and instruct the browser to execute the command (i.e., to POST the form).
- 9: else → Follow (random) link to facilitate referrer-based attacks.
 - 10: let *protocol* ← {P, S} → Non-deterministically select protocol (HTTP or HTTPS).
 - 11: let *host* ← Doms → Non-det. select host.
 - 12: let *path* ← S → Non-det. select path.
 - 13: let *fragment* ← S → Non-det. select fragment part.
 - 14: let *parameters* ← [S × S] → Non-det. select parameters.
 - 15: let *url* := ⟨URL, *protocol*, *host*, *path*, *parameters*, *fragment*⟩ → Assemble request URL.
 - 16: let *command* := ⟨HREF, *url*, ⊥, ⊥⟩ → Follow link to the selected URL.
 - 17: stop (*s*, *cookies*, *localStorage*, *sessionStorage*, *command*) → Finish script's run and instruct the browser to execute the command (follow link).

K. Authorization Servers

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

Definition 8. A state $s \in Z^{as}$ of an authorization server as is a term of the form $\langle \text{DNSaddress}, \text{pendingDNS}, \text{pendingRequests}, \text{corrupt}, \text{keyMapping}, \text{tlskeys}, \text{jwk}, \text{pendingClientIds}, \text{clients}, \text{records}, \text{authorizationRequests}, \text{cibaAuthnRequests}, \text{mtlsRequests}, \text{cibaEndUserEndpoints}, \text{rsCredentials} \rangle$ with $\text{DNSaddress} \in \text{IPs}$, $\text{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$, $\text{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\text{keyMapping} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\text{tlskeys} \in [\text{Doms} \times K_{\text{TLS}}]$ (all former components as in Definition 77), $\text{jwk} \in K_{\text{sign}}$, $\text{pendingClientIds} \in \mathcal{T}_{\mathcal{N}}$, $\text{clients} \in [\mathcal{T}_{\mathcal{N}} \times [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]]$, $\text{records} \in \mathcal{T}_{\mathcal{N}}$, $\text{authorizationRequests} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\text{cibaAuthnRequests} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\text{mtlsRequests} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\text{cibaEndUserEndpoints} \in \mathcal{T}_{\mathcal{N}}$, and $\text{rsCredentials} \in \mathcal{T}_{\mathcal{N}}$.

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

- $s_0^{as}.\text{DNSaddress} \in \text{IPs}$,
- $s_0^{as}.\text{pendingDNS} \equiv \langle \rangle$,
- $s_0^{as}.\text{pendingRequests} \equiv \langle \rangle$,
- $s_0^{as}.\text{corrupt} \equiv \perp$,
- $s_0^{as}.\text{keyMapping}$ being the same as the keymapping for browsers,
- $s_0^{as}.\text{tlskeys} \equiv \text{tlskeys}^{as}$ (see Appendix A-C),
- $s_0^{as}.\text{jwk} \equiv \text{signkey}(as)$ (see Appendix A-C),
- $s_0^{as}.\text{pendingClientIds} \equiv \langle \rangle$,
- $s_0^{as}.\text{clients} \equiv \langle \rangle$,
- $s_0^{as}.\text{records} \equiv \langle \rangle$,
- $s_0^{as}.\text{authorizationRequests} \equiv \langle \rangle$,
- $s_0^{as}.\text{cibaAuthnRequests} \equiv \langle \rangle$,
- $s_0^{as}.\text{mtlsRequests} \equiv \langle \rangle$,
- $s_0^{as}.\text{cibaEndUserEndpoints} \equiv \text{userEp}$ where userEp is a dictionary and $\langle \text{identity}, \text{ep} \rangle \in \langle \rangle \text{userEp} \Leftrightarrow (\text{identity}.\text{domain} \in \text{dom}(as) \wedge \text{dom}^{-1}(\text{ep}.\text{host}) = \text{ownerOfID}(\text{identity}))$, i.e., userEp maps identities to a domain of the browser of the identity (note that the browser model can receive requests as a modeling artefact), and
- $s_0^{as}.\text{rsCredentials} \equiv \text{rsCreds}$ where rsCreds is a dictionary and $\langle \text{rsDom}, c \rangle \in \langle \rangle \text{rsCreds} \Leftrightarrow (\exists d \in \text{dom}(as), \text{rsDom} \in \text{Doms}: c \equiv \text{secretOfRS}(d, \text{rsDom}))$, i.e., rsCreds maps RS domains to the corresponding RS credentials..

We now specify the relation R^{as} : This relation is based on the model of generic HTTPS servers (see Appendix E-L). We specify algorithms that differ from or do not exist in the generic server model in Algorithms 11 to 12. Algorithm 16 shows the script `script_as_form` that is used by ASs.

Algorithm 11 Relation of AS R^{as} – Processing HTTPS Requests

```
1: function PROCESS_HTTPS_REQUEST( $m, k, a, f, s'$ )
2:   if  $m.path \equiv /.well-known/openid-configuration \vee$ 
    $\hookrightarrow m.path \equiv /.well-known/oauth-authorization-server$  then  $\rightarrow$  We model both OIDD, RFC 8414, and FAPI CIBA.
3:     let  $metaData := [issuer: m.host]$ 
4:     let  $metaData[auth\_ep] := \langle URL, S, m.host, /auth, \langle \rangle, \perp \rangle$ 
5:     let  $metaData[token\_ep] := \langle URL, S, m.host, /token, \langle \rangle, \perp \rangle$ 
6:     let  $metaData[par\_ep] := \langle URL, S, m.host, /par, \langle \rangle, \perp \rangle$ 
7:     let  $metaData[introspec\_ep] := \langle URL, S, m.host, /introspect, \langle \rangle, \perp \rangle$ 
8:     let  $metaData[jwks\_uri] := \langle URL, S, m.host, /jwks, \langle \rangle, \perp \rangle$ 
9:     let  $metaData[reg\_ep] := \langle URL, S, m.host, /reg, \langle \rangle, \perp \rangle$ 
    $\rightarrow$  No support for push mode, see Section 5.2.2 of FAPI-CIBA [49]
10:    let  $metaData[backchannel\_token\_delivery\_modes\_supported] := \langle poll, ping \rangle$ 
11:    let  $metaData[backchannel\_authentication\_endpoint] := \langle URL, S, m.host, /backchannel-authn, \langle \rangle, \perp \rangle$ 
12:    let  $metaData[grant\_types\_supported] := \langle authorization\_code, urn:openid:params:grant-type:ciba \rangle$ 
13:    let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle \rangle, metaData \rangle, k)$ 
14:    stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
15:  else if  $m.path \equiv /jwks$  then
16:    let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle \rangle, pub(s'.jwks) \rangle, k)$ 
17:    stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
18:  else if  $m.path \equiv /reg \wedge m.method \equiv POST$  then
19:    call REGISTER_CLIENT( $m, k, a, f, s'$ )  $\rightarrow$  See Algorithm 13
20:  else if  $m.path \equiv /manage \wedge m.method \equiv PUT$  then  $\rightarrow$  DCM: update client metadata (see Sec. 2.2 of RFC 7592 [43])
21:    let  $clientId := m.body[clientId]$ 
22:    if  $clientId \notin s'.clients$  then
23:      stop  $\rightarrow$  Unknown client
24:    let  $clientInfo := s'.clients[clientId]$ 
25:    let  $regAT := m.headers[Authorization][Bearer]$ 
26:    if  $regAT \neq clientInfo[reg\_at]$  then
27:      stop  $\rightarrow$  Wrong registration access token
28:    let  $redirectUri := m.body[redirect\_uris]$ 
29:    let  $jwks := m.body[jwks]$   $\rightarrow$  Contains public keys of client
30:    let  $pubSigKey$  such that  $[use: sig, val: pubSigKey] \in \langle \rangle jwks$  if possible; otherwise stop
31:    let  $mtlsPubKey$  such that  $[use: TLS, val: mtlsPubKey] \in \langle \rangle jwks$  if possible; otherwise stop
32:    let  $regUri := \langle URL, S, m.host, /manage, \langle \rangle, \perp \rangle$ 
33:    let  $clientType \leftarrow \{mTLS\_mTLS, mTLS\_DPoP, pkjwt\_mTLS, pkjwt\_DPoP\}$   $\rightarrow$  Non-deterministic choice of client type
34:    let  $clientInfo[client\_type] := clientType$ 
35:    let  $clientInfo[jwt\_key] := pubSigKey$ 
36:    let  $clientInfo[mtls\_key] := mtlsPubKey$ 
37:    let  $clientInfo[redirect\_uris] := redirectUri$ 
38:    let  $regResponse := [client\_id: clientId, jwks: jwks, client\_type: clientType, reg\_at: regAT, reg\_client\_uri: regUri]$ 
39:    let  $tokenDeliveryMode \leftarrow \{poll, ping\}$   $\rightarrow$  Non-deterministic choice of CIBA token delivery mode
40:    let  $grantTypes := \langle authorization\_code, urn:openid:params:grant-type:ciba \rangle$   $\rightarrow$  AS registers both types (in our model)
41:    let  $clientInfo[grant\_types] := grantTypes$ 
42:    let  $regResponse[grant\_types] := grantTypes$ 
43:    let  $clientInfo[backchannel\_token\_delivery\_mode] := tokenDeliveryMode$ 
44:    let  $regResponse[backchannel\_token\_delivery\_mode] := tokenDeliveryMode$ 
45:    if  $tokenDeliveryMode \equiv ping$  then
46:      if  $backchannel\_client\_notification\_endpoint \notin m.body$  then
47:        stop
48:      let  $clientNotificationEP := m.body[backchannel\_client\_notification\_endpoint]$ 
49:      let  $regResponse[backchannel\_client\_notification\_endpoint] := clientNotificationEP$ 
50:      let  $clientInfo[backchannel\_client\_notification\_endpoint] := clientNotificationEP$ 
51:    let  $s'.clients[clientId] := clientInfo$ 
52:    let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle \rangle, regResponse \rangle, k)$ 
53:    stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
54:  else if  $m.path \equiv /manage \wedge m.method \equiv DELETE$  then  $\rightarrow$  DCM: delete client (see Sec. 2.3 of RFC 7592 [43])
55:    let  $regAT := m.headers[Authorization][Bearer]$ 
56:    let  $clientId$  such that  $s'.clients[clientId][reg\_at] \equiv regAT$  if possible; otherwise stop
57:    let  $s'.clients[clientId][active] := \perp$   $\rightarrow$  Deactivate client account
58:    stop  $\langle \rangle, s'$ 
 $\rightarrow$  Algorithm continues on next page.
```

```

59: else if  $m.path \equiv /auth$  then  $\rightarrow$  Authorization endpoint: Reply with login page.
60:   if  $m.method \equiv GET$  then
61:     let  $data := m.parameters$ 
62:   else if  $m.method \equiv POST$  then
63:     let  $data := m.body$ 
64:   let  $requestUri := data[request\_uri]$ 
65:   if  $requestUri \equiv \langle \rangle$  then
66:     stop  $\rightarrow$  FAPI 2.0 mandates PAR, therefore a request URI is required
67:   let  $authzRecord := s'.authorizationRequests[requestUri]$ 
68:   let  $clientId := data[client\_id]$ 
69:   if  $authzRecord[client\_id] \neq clientId$  then  $\rightarrow$  Check binding of request URI to client
70:     stop
71:   if  $clientId \notin s'.clients \vee s'.clients[clientId][active] \neq \top$  then
72:     stop  $\rightarrow$  Unknown client
73:   let  $s'.authorizationRequests[requestUri][auth2\_reference] := \nu_5$ 
74:   let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle \langle ReferrerPolicy, origin \rangle \rangle, \langle \langle script\_as\_form, [auth2\_reference: \nu_5] \rangle \rangle \rangle, k)$ 
75:   stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
76: else if  $m.path \equiv /auth2 \wedge m.method \equiv POST \wedge m.headers[Origin] \equiv \langle m.host, S \rangle$  then  $\rightarrow$  Second step of authorization
77:   let  $identity := m.body[identity]$ 
78:   let  $password := m.body[password]$ 
79:   if  $identity.domain \notin dom(as)$  then
80:     stop  $\rightarrow$  This AS does not manage  $identity$ 
81:   if  $password \neq secretOfID(identity)$  then
82:     stop  $\rightarrow$  Invalid user credentials
83:   let  $auth2Reference := m.body[auth2\_reference]$ 
84:   let  $requestUri$  such that  $s'.authorizationRequests[requestUri][auth2\_reference] \equiv auth2Reference$ 
85:      $\hookrightarrow$  if possible; otherwise stop
86:   let  $authzRecord := s'.authorizationRequests[requestUri]$ 
87:   let  $authzRecord[subject] := identity$ 
88:   let  $authzRecord[issuer] := m.host$ 
89:   let  $authzRecord[code] := \nu_1$   $\rightarrow$  Generate a fresh, random authorization code
90:   let  $s'.records := s'.records + \langle \rangle authzRecord$ 
91:   let  $redirectUri := authzRecord[redirect\_uri]$ 
92:   let  $responseData := [code: authzRecord[code]]$ 
93:   if  $authzRecord[state] \neq \langle \rangle$  then
94:     let  $responseData[state] := authzRecord[state]$ 
95:   if  $authzRecord[sign\_authz\_response] \equiv \top$  then
96:     let  $responseData[iss] := authzRecord[issuer]$   $\rightarrow$  iss claim is part of JWT instead of a parameter, see [22, Sec. 5.4.1]
97:     let  $responseData[aud] := clientId$   $\rightarrow$  See JARM [32, Sec. 2.1]
98:     let  $responseData := [response: sig(responseData, s'.jwk)]$   $\rightarrow$  Sign authorization response using JARM [32, Sec. 2.3.1]
99:   else
100:     let  $redirectUri.parameters[iss] := authzRecord[issuer]$   $\rightarrow$  Overwrite iss parameter if present in  $redirectUri$ 
101:     let  $redirectUri.parameters := redirectUri.parameters \cup responseData$ 
102:     let  $m' := enc_s(\langle HTTPResp, m.nonce, 303, \langle \langle Location, redirectUri \rangle \rangle, \langle \rangle \rangle, k)$ 
103:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
104: else if  $m.path \equiv /par \wedge m.method \equiv POST$  then  $\rightarrow$  Pushed Authorization Request
105:   let  $requireSignedPAR \leftarrow \{\top, \perp\}$   $\rightarrow$  Choose whether to require a signed PAR (see also Section V-C)
106:    $\rightarrow$  Note: If the client signed the PAR, but the AS chooses not to require a signature, client authentication below will fail.
107:   if  $requireSignedPAR \equiv \top$  then
108:     let  $mBody := extractmsg(m.body)$   $\rightarrow$  Note: If  $m.body \not\sim sig(*, *)$  (or  $mac(*, *)$ ), then there is no processing step
109:     if  $checksig(m.body, s'.clients[mBody[client\_id]][jwt\_key]) \neq \top$  then
110:       stop  $\rightarrow$  Invalid signature
111:     if  $mBody[aud] \neq m.host$  then
112:       stop  $\rightarrow$  Wrong audience value in JWS, see [22, Sec. 5.3.1 No. 2]
113:   else
114:     let  $mBody := m.body$ 
115:     let  $m.body := mBody$   $\rightarrow$  In case of a signed PAR: Strip off the signature after verifying it
116:     if  $m.body[response\_type] \neq code \vee m.body[code\_challenge\_method] \neq S256$  then
117:       stop
118:     let  $authnResult := AUTHENTICATE\_CLIENT(m, s')$   $\rightarrow$  Stops in case of errors/failed authentication
119:     let  $clientId := authnResult.1$ 
120:     let  $s' := authnResult.2$ 
121:     let  $mtlsInfo := authnResult.3$ 
122:     if  $clientId \neq mBody[client\_id]$  then
123:       stop  $\rightarrow$  Key used in client authentication is not registered for  $m.body[client\_id]$ 

```

\rightarrow Algorithm continues on next page.

```

122:   let redirectUri := mBody[redirect_uri] → Clients are required to send redirect_uri with each request
123:   if redirectUri ≡ ⟨⟩ then
124:     stop
125:   if redirectUri.protocol ≠ S then
126:     stop
127:   let codeChallenge := mBody[code_challenge] → PKCE challenge
128:   if codeChallenge ≡ ⟨⟩ then
129:     stop → Missing PKCE challenge
130:   let requestUri := ν4 → Choose random URI
131:   let authzRecord := [client_id: clientId]
132:   let authzRecord[state] := mBody[state]
133:   let authzRecord[scope] := mBody[scope]
134:   if nonce ∈ mBody then
135:     let authzRecord[nonce] := mBody[nonce]
136:   let authzRecord[redirect_uri] := redirectUri
137:   let authzRecord[code_challenge] := codeChallenge
138:   let authzRecord[signed_par] := requireSignedPAR
139:   if response_mode ∈ mBody ∧ mBody[response_mode] ≡ jwt then → Check whether client requested a signed response
140:     let authzRecord[sign_authz_response] := ⊤
141:   let body := [request_uri: requestUri]
142:   let s'.authorizationRequests[requestUri] := authzRecord → Store data linked to requestUri
143:   let m' := encs((HTTPResp, m.nonce, 201, ⟨⟩, body), k)
144:   stop ⟨⟨f, a, m'⟩⟩, s'
145: else if m.path ≡ /token ∧ m.method ≡ POST then
146:   if m.body[grant_type] ≠ authorization_code ∧ m.body[grant_type] ≠ urn:openid:params:grant-type:ciba then
147:     stop
148:   let authnResult := AUTHENTICATE_CLIENT(m, s') → Stops in case of errors/failed authentication
149:   let clientId := authnResult.1
150:   let s' := authnResult.2
151:   let mtlInfo := authnResult.3
152:   if m.body[grant_type] ≡ authorization_code then
153:     let code := m.body[code]
154:     let codeVerifier := m.body[code_verifier]
155:     if code ≡ ⟨⟩ ∨ codeVerifier ≡ ⟨⟩ then
156:       stop → Missing code or code_verifier
157:     let record, ptr such that record ≡ s'.records.ptr ∧ record[code] ≡ code
158:       ↪ ∧ code ≠ ⊥ ∧ ptr ∈ ℕ if possible; otherwise stop
159:     if record[code_challenge] ≠ hash(codeVerifier) ∨ record[redirect_uri] ≠ m.body[redirect_uri] then
160:       stop → PKCE verification failed or URI mismatch
161:   else if m.body[grant_type] ≡ urn:openid:params:grant-type:ciba then
162:     let authReqId := m.body[auth_req_id]
163:     if authReqId ≡ ⟨⟩ then
164:       stop → Missing auth_req_id
165:     let record, ptr such that record ≡ s'.records.ptr ∧ record[auth_req_id] ≡ auth_req_id
166:       ↪ ∧ auth_req_id ≠ ⊥ ∧ ptr ∈ ℕ if possible; otherwise stop
167:   if record[client_id] ≠ clientId then
168:     stop
169:   let clientType := s'.clients[clientId][client_type]
170:   if clientType ≡ pkjwt_DPoP ∨ clientType ≡ mTLS_DPoP then → DPoP token binding
171:     let tokenType := DPoP
172:     let dpopProof := m.headers[DPoP]
173:     let dpopJwt := extractmsg(dpopProof)
174:     let verificationKey := dpopJwt[headers][jwk]
175:     if checksig(dpopProof, verificationKey) ≠ ⊤ ∨ verificationKey ≡ ⟨⟩ then
176:       stop → Invalid DPoP signature (or empty jwk header)
177:     let dpopClaims := dpopJwt[payload]
178:     let reqUri := ⟨URL, S, m.host, m.path, ⟨⟩, ⊥⟩
179:     if dpopClaims[htm] ≠ m.method ∨ dpopClaims[htu] ≠ reqUri then
180:       stop → DPoP claims do not match corresponding message
181:     let cnfContent := [jkt: hash(verificationKey)]
182:   else if clientType ≡ pkjwt_mTLS ∨ clientType ≡ mTLS_mTLS then → mTLS token binding
183:     let tokenType := Bearer
184:     let mtlNonce := m.body[TLS_binding]

```

→ Algorithm continues on next page.

```

183:   if  $clientType \equiv mTLS\_mTLS$  then  $\rightarrow$  Client used mTLS authentication, reuse data from authentication
184:     if  $mtlsNonce \neq mtlsInfo.1$  then
185:       stop  $\rightarrow$  Client tried to use different mTLS nonce for authentication and token binding
186:     else  $\rightarrow$  Client did not use mTLS authentication
187:       let  $mtlsInfo$  such that  $mtlsInfo \in s'.mtlsRequests[clientId] \wedge mtlsInfo.1 \equiv mtlsNonce$  if possible; otherwise stop

188:         let  $s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] - \langle \rangle mtlsInfo$ 
189:         let  $mTlsKey := mtlsInfo.2$   $\rightarrow$  mTLS public key of client
190:         let  $cnfContent := [x5t\#S256: hash(mTlsKey)]$ 
191:       else
192:         stop  $\rightarrow$  Client used neither DPoP nor mTLS
193:     if  $m.body[grant\_type] \equiv authorization\_code$  then
194:       let  $s'.records.ptr[code] := \perp$   $\rightarrow$  Invalidate code
195:     else
196:       let  $s'.records.ptr[auth\_req\_id] := \perp$   $\rightarrow$  Invalidate request id
197:     let  $atType \leftarrow \{JWT, opaque\}$   $\rightarrow$  The AS chooses randomly whether it issues a structured or an opaque access token
198:     if  $atType \equiv JWT$  then  $\rightarrow$  Structured access token
199:       let  $accessTokenContent := [cnf: cnfContent, sub: record[subject], client\_sig\_key: s'.clients[clientId][jwt\_key]]$ 
200:       let  $accessToken := sig(accessTokenContent, s'.jwk)$ 
201:     else  $\rightarrow$  Opaque access token
202:       let  $accessToken := \nu_2$   $\rightarrow$  Fresh random value
203:     let  $s'.records.ptr[access\_token] := accessToken$   $\rightarrow$  Store for token introspection
204:     let  $s'.records.ptr[cnf] := cnfContent$   $\rightarrow$  Store for token introspection
205:     let  $body := [access\_token: accessToken, token\_type: tokenType]$ 
206:     if  $record[scope] \equiv openid$  then  $\rightarrow$  Client requested ID token
207:       let  $idTokenBody := [iss: record[issuer]]$ 
208:       let  $idTokenBody[sub] := record[subject]$ 
209:       let  $idTokenBody[aud] := record[client\_id]$ 
210:       if  $nonce \in record$  then
211:         let  $idTokenBody[nonce] := record[nonce]$ 
212:       let  $idToken := sig(idTokenBody, s'.jwk)$ 
213:       let  $body[id\_token] := idToken$ 
214:     let  $m' := enc_s(\langle \langle HTTPResp, m.nonce, 200, \langle \rangle, body \rangle, k \rangle)$ 
215:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
216:   else if  $m.path \equiv /introspect \wedge m.method \equiv POST \wedge token \in m.body$  then
217:     let  $rsSecret$  such that  $\langle Basic, rsSecret \rangle \equiv m.headers[Authorization]$  if possible; otherwise stop
218:     let  $rsDom$  such that  $s'.rsCredentials[rsDom] \equiv rsSecret$  if possible; otherwise stop  $\rightarrow$  RS authentication at AS
219:     let  $token := m.body[token]$ 
220:     let  $record$  such that  $record \in s'.records \wedge record[access\_token] \equiv token$  if possible; otherwise let  $record := \diamond$ 
221:     if  $record \equiv \diamond \vee s'.clients[record[client\_id]][active] \neq \top$  then  $\rightarrow$  Unknown  $token$  or deactivated client
222:       let  $body := [active: \perp]$ 
223:     else  $\rightarrow$   $token$  was issued by this AS & client is active
224:       let  $clientId := record[client\_id]$ 
225:        $\rightarrow$   $cnf$  claim contains hash of token binding key, the signing key is the key used by the client to sign HTTP messages
226:       let  $body := [active: \top, cnf: record[cnf], sub: record[subject], client\_sig\_key: s'.clients[clientId][jwt\_key]]$ 
227:     if  $m.headers[Accept] \equiv app/token-introspection+jwt$  then  $\rightarrow$  Check whether RS requested a signed response
228:       let  $body := sig([token\_introspection: body, iss: m.host, aud: rsDom], s'.jwk)$ 
229:     let  $m' := enc_s(\langle \langle HTTPResp, m.nonce, 200, \langle \rangle, body \rangle, k \rangle)$ 
230:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
231:   else if  $m.path \equiv /MTLS-prepare$  then  $\rightarrow$  See Appendix A-G
232:     let  $clientId := m.body[client\_id]$ 
233:     if  $s'.clients[clientId][active] \neq \top$  then
234:       stop
235:     let  $mtlsNonce := \nu_3$ 
236:     let  $clientKey := s'.clients[clientId][mtls\_key]$ 
237:     if  $clientKey \equiv \langle \rangle \vee clientKey \equiv pub(\diamond)$  then
238:       stop  $\rightarrow$  Client has no mTLS key
239:     let  $s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] + \langle \rangle (mtlsNonce, clientKey)$ 
240:     let  $m' := enc_s(\langle \langle HTTPResp, m.nonce, 200, \langle \rangle, enc_a(\langle mtlsNonce, s'.keyMapping[m.host] \rangle, clientKey) \rangle, k \rangle)$ 
241:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
242:   else if  $m.path \equiv /backchannel-authn \wedge m.method \equiv POST$  then  $\rightarrow$  CIBA Authentication Request
243:     let  $authnResult := AUTHENTICATE\_CLIENT(m, s')$   $\rightarrow$  Stops in case of errors/failed authentication
244:     let  $clientId := authnResult.1$ 
245:     let  $s' := authnResult.2$ 
246:     let  $mtlsInfo := authnResult.3$ 

```

\rightarrow Algorithm continues on next page.

```

246:   if  $clientId \neq m.body[client\_id]$  then
247:     stop → Key used in client authentication is not registered for  $m.body[client\_id]$ 
248:   if  $openid \notin m.body[scope]$  then
249:     stop
250:   if  $urn:openid:params:grant-type:ciba \notin s'.clients[clientId][grant\_types]$  then
251:     stop → Client not registered as a CIBA client
252:   let  $authzRecord := [client\_id: clientId]$ 
253:   let  $authzRecord[scope] := m.body[scope]$ 
254:   let  $authzRecord[binding\_message] := m.body[binding\_message]$ 
255:   let  $authzRecord[selected\_identity] := m.body[login\_hint]$ 
256:   let  $deliveryMode := s'.clients[clientId][backchannel\_token\_delivery\_mode]$ 
257:   if  $deliveryMode \equiv ping$  then
258:     let  $authzRecord[client\_notification\_token] := m.body[client\_notification\_token]$ 
259:   if  $m.body[selected\_identity].domain \notin dom(as)$  then
260:     stop → This AS does not manage the requested identity
261:   let  $authzRecord[authenticateUser] := \top$  → Flag indicating whether the AS needs to obtain end-user consent/authorization
262:   let  $authnReqId := \nu_{authn\_req\_id}$ 
263:   let  $s'.cibaAuthnRequests[authnReqId] := authzRecord$  → Store data linked to  $authnReqId$ 
264:   let  $body := [auth\_req\_id: authnReqId]$ 
265:   let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle \rangle, body \rangle, k)$ 
266:   stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
267: else if  $m.path \equiv /ciba-auth$  then → Authorization endpoint for CIBA Flows: Reply with login page, include binding message.
268:   if  $m.method \neq POST$  then
269:     stop
270:   if  $ciba\_user\_nonce \notin m.body$  then
271:     stop
272:   let  $cibaUserNonce := m.body[ciba\_user\_nonce]$ 
273:   let  $authnReqId$  such that  $authnReqId \in s'.cibaAuthnRequests$ 
     ↪  $\wedge s'.cibaAuthnRequests[authnReqId][cibaUserAuthNNonce] \equiv cibaUserNonce$  if possible; otherwise stop
274:   let  $bindingMessage := s'.cibaAuthnRequests[authnReqId][binding\_message]$ 
275:   let  $clientId := s'.cibaAuthnRequests[authnReqId][client\_id]$ 
276:   let  $clientDom \leftarrow s'.clients[clientId][redirect\_uris]$ 
277:   let  $s'.cibaAuthnRequests[authnReqId][ciba\_auth2\_reference] := \nu_{ciba\_auth2\_ref}$ 
278:   let  $body := \langle script\_as\_ciba\_form, [ciba\_auth2\_reference: \nu_{ciba\_auth2\_ref}, binding\_message: bindingMessage]$ 
     ↪  $client\_domain: clientDom.host \rangle$ 
279:   let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle ReferrerPolicy, origin \rangle, body \rangle, k)$ 
280:   stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
281: else if  $m.path \equiv /ciba-auth2 \wedge m.method \equiv POST \wedge m.headers[Origin] \equiv \langle m.host, S \rangle$  then → Finish authorization (CIBA)
282:   let  $identity := m.body[identity]$ 
283:   let  $password := m.body[password]$ 
284:   if  $identity.domain \notin dom(as)$  then
285:     stop → This AS does not manage  $identity$ 
286:   if  $password \neq secretOfID(identity)$  then
287:     stop → Invalid user credentials
288:   let  $auth2Reference := m.body[ciba\_auth2\_reference]$ 
289:   let  $authnReqId$  such that  $s'.cibaAuthnRequests[authnReqId][ciba\_auth2\_reference] \equiv auth2Reference$ 
     ↪ if possible; otherwise stop
290:   if  $identity \neq s'.cibaAuthnRequests[authnReqId][selected\_identity]$  then
291:     stop → Identity does not match the identity initially chosen for this flow
292:   let  $s'.cibaAuthnRequests[authnReqId][authenticateUser] := \perp$  → The user is now authenticated.
293:   let  $authzRecord := s'.cibaAuthnRequests[authnReqId]$ 
294:   let  $authzRecord[subject] := identity$ 
295:   let  $authzRecord[issuer] := m.host$ 
296:   let  $authzRecord[auth\_req\_id] := authnReqId$ 
297:   let  $s'.records := s'.records + \langle \rangle authzRecord$  → Add the whole record to the records entry (the AS will issue an AT when
     receiving a token request with the corresponding  $auth\_req\_id$  value)
298:   let  $clientId := authzRecord[client\_id]$ 
299:   if  $s'.clients[clientId][backchannel\_token\_delivery\_mode] \equiv ping$  then
300:     let  $clientURL := s'.clients[clientId][backchannel\_client\_notification\_endpoint]$ 
301:     let  $body := [auth\_req\_id: authnReqId]$ 
302:     let  $headers := [Authorization: \langle Bearer, authzRecord[client\_notification\_token] \rangle]$ 
303:     let  $message := \langle HTTPReq, \nu_{ciba\_ping}, POST, clientURL.host, clientURL.path, \langle \rangle, headers, body \rangle$ 
304:     call HTTPS_SIMPLE_SEND( $[responseTo: CIBAPingCallback], message, a, s'$ )
305:   else
306:     stop  $\langle \rangle, s'$ 
307: stop → Request was malformed or sent to non-existing endpoint.

```

Algorithm 12 Relation of AS R^{as} – Client Authentication

```
1: function AUTHENTICATE_CLIENT( $m, s'$ )  $\rightarrow$  Check client authentication in message  $m$ . Stops the current processing step in case
   of errors or failed authentication.
2:   if  $client\_assertion \in m.body$  then  $\rightarrow$  private_key_jwt client authentication
3:     let  $jwt_s := m.body[client\_assertion]$ 
4:     let  $clientId, verificationKey$  such that  $verificationKey \equiv s'.clients[clientId][jwt\_key] \wedge$ 
        $\hookrightarrow checksig(jwt_s, verificationKey) \equiv \top$  if possible; otherwise stop
5:     if  $verificationKey \equiv \langle \rangle \vee verificationKey \equiv pub(\diamond)$  then
6:       stop  $\rightarrow$  Client has no jwt key
7:     let  $clientInfo := s'.clients[clientId]$ 
8:     let  $clientType := clientInfo[client\_type]$ 
9:     if  $clientType \neq pkjwt\_mTLS \wedge clientType \neq pkjwt\_DPoP$  then
10:      stop  $\rightarrow$  Client authentication type mismatch
11:     let  $jwt := extractmsg(jwt_s)$ 
12:     if  $jwt[iss] \neq clientId \vee jwt[sub] \neq clientId$  then
13:       stop
14:     if  $jwt[aud] \neq \langle URL, S, m.host, /token, \langle \rangle, \perp \rangle \wedge jwt[aud] \neq m.host$   $\rightarrow$  issuer in AS metadata is just the host part
        $\hookrightarrow \wedge jwt[aud] \neq \langle URL, S, m.host, /par, \langle \rangle, \perp \rangle$  then
15:       stop  $\rightarrow$  aud claim value is neither token, nor PAR endpoint nor AS issuer identifier
16:   else if  $TLS\_AuthN \in m.body$  then  $\rightarrow$  mTLS client authentication
17:     let  $clientId := m.body[client\_id]$   $\rightarrow$  RFC 8705 [7] mandates client_id when using mTLS authentication
18:     let  $mtlsNonce := m.body[TLS\_AuthN]$ 
19:     let  $mtlsInfo$  such that  $mtlsInfo \in s'.mtlsRequests[clientId] \wedge mtlsInfo.1 \equiv mtlsNonce$  if possible; otherwise stop
20:     let  $clientInfo := s'.clients[clientId]$ 
21:     let  $clientType := clientInfo[client\_type]$ 
22:     if  $clientType \neq mTLS\_mTLS \wedge clientType \neq mTLS\_DPoP$  then
23:       stop  $\rightarrow$  Client authentication type mismatch
24:     let  $s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] - \langle \rangle mtlsInfo$ 
25:   else
26:     stop  $\rightarrow$  Unsupported client (authentication) type
27:   if  $s'.clients[clientId][active] \neq \top$  then
28:     stop
29:   if  $clientType \equiv mTLS\_mTLS \vee clientType \equiv mTLS\_DPoP$  then
30:     return  $\langle clientId, s', mtlsInfo \rangle$ 
31:   else
32:     return  $\langle clientId, s', \perp \rangle$   $\rightarrow$  private_key_jwt client authentication, i.e., no mTLS info
```

Algorithm 13 Relation of AS R^{as} – Process a DCR request.

```
1: function REGISTER_CLIENT( $m, k, a, f, s'$ )  $\rightarrow m$  is the decrypted HTTP request containing a client registration request
2:   let  $clientId \leftarrow s'.pendingClientIds$   $\rightarrow$  Client ids are provided by the attacker (see also Algorithm 14).
3:   let  $s'.pendingClientIds := s'.pendingClientIds - \langle \rangle clientId$ 
    $\rightarrow$  Construct client information response (see Sec. 2 of RFC 7592 [43] and Sec. 3.2.1 of RFC 7591 [42])
4:   let  $redirectUri := m.body[redirect_uris]$ 
5:   let  $jwtks := m.body[jwks]$   $\rightarrow$  Contains public keys of client
6:   let  $pubSigKey$  such that  $[use: sig, val: pubSigKey] \in \langle \rangle jwtks$  if possible; otherwise stop
7:   let  $mtlsPubKey$  such that  $[use: TLS, val: mtlsPubKey] \in \langle \rangle jwtks$  if possible; otherwise stop
8:   let  $clientType \leftarrow \{mTLS\_mTLS, mTLS\_DPoP, pkjwt\_mTLS, pkjwt\_DPoP\}$   $\rightarrow$  Non-deterministic choice of client type
9:   let  $regAT := \nu_{regAT}$   $\rightarrow$  Registration access token (cf. Sec. 3 of RFC 7592 [43])
10:  let  $regUri := \langle URL, S, m.host, /manage, \langle \rangle, \perp \rangle$   $\rightarrow$  Registration client uri (cf. Sec. 3 of RFC 7592 [43])
11:  let  $tokenDeliveryMode \leftarrow \{poll, ping\}$   $\rightarrow$  Non-deterministic choice of CIBA token delivery mode
12:  let  $grantTypes := \langle authorization\_code, urn:openid:params:grant-type:ciba \rangle$   $\rightarrow$  In the model, the AS always registers
   both types
13:  let  $regResponse := [client\_id: clientId, jwtks: jwtks, client\_type: clientType, reg\_at: regAT, reg\_client\_uri: regUri,$ 
    $\hookrightarrow grant\_types: grantTypes]$ 
14:  let  $clientInfo := [client\_type: clientType, redirect\_uris: redirectUri, jwt\_key: pubSigKey,$ 
    $\hookrightarrow mtls\_key: mtlsPubKey, reg\_at: regAT, grant\_types: grantTypes]$ 
15:  let  $clientInfo[active] := \top$   $\rightarrow$  This flag indicates whether a client account is active
16:  let  $regResponse[backchannel\_token\_delivery\_mode] := tokenDeliveryMode$ 
17:  let  $clientInfo[backchannel\_token\_delivery\_mode] := tokenDeliveryMode$ 
18:  if  $tokenDeliveryMode \equiv ping$  then
19:    if  $backchannel\_client\_notification\_endpoint \notin m.body$  then
20:      stop
21:    let  $clientNotificationEP := m.body[backchannel\_client\_notification\_endpoint]$ 
22:    let  $regResponse[backchannel\_client\_notification\_endpoint] := clientNotificationEP$ 
23:    let  $clientInfo[backchannel\_client\_notification\_endpoint] := clientNotificationEP$ 
24:  let  $s'.clients[clientId] := clientInfo$ 
25:  let  $m' := enc_s(\langle HTTPResp, m.nonce, 201, \langle \rangle, regResponse \rangle, k)$ 
26:  stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
```

Algorithm 14 Relation of AS R^{as} – Processing other messages.

```
1: function PROCESS_OTHER( $m, a, f, s'$ )
2:   let  $clientId := m$   $\rightarrow m$  is a client id chosen by and sent by an attacker process (see also Line 2 of Algorithm 13)
3:   if  $clientId \in s'.clients \vee clientId \in s'.pendingClientIds$  then
4:     stop
5:   let  $s'.pendingClientIds := s'.pendingClientIds + \langle \rangle clientId$ 
6:   stop  $\langle \rangle, s'$ 
```

Algorithm 15 Relation of a AS R^{as} – Handle trigger events.

```
1: function PROCESS_TRIGGER( $a, s'$ )
2:   let  $action \leftarrow \{CIBA\_OBTAIN\_CONSENT\}$ 
3:   switch  $action$  do
4:     case CIBA_OBTAIN_CONSENT
    $\rightarrow$  Choose one of the CIBA authentication requests for which the AS did not ask the end-user yet
5:       let  $authnReqId$  such that  $authnReqId \in s'.cibaAuthnRequests$ 
    $\hookrightarrow \wedge s'.cibaAuthnRequests[authnReqId][authenticateUser] \equiv \top$  if possible; otherwise stop
6:       let  $selectedUser := s'.cibaAuthnRequests[authnReqId][selected\_identity]$ 
    $\rightarrow$  Get the endpoint of the end-user
7:       let  $userEp := s'.cibaEndUserEndpoints[selectedUser]$ 
8:       let  $cibaUserAuthNNonce := \nu_{cibaUserNonce}$   $\rightarrow$  In the model, we let the AS chose a nonce that it sends to the user's browser.
   The browser sends this nonce to an endpoint of the AS, which the AS uses to identify the authentication request.
9:       let  $s'.cibaAuthnRequests[authnReqId][cibaUserAuthNNonce] := cibaUserAuthNNonce$ 
10:      let  $cibaURL := \{ \langle URL, S, d, /ciba\_auth, \langle \rangle, \perp \rangle \mid d \in \text{dom}(as) \}$   $\rightarrow$  A URI of the AS at which the end-user can authenticate
   for CIBA flows.
11:      let  $body := [ciba\_user\_nonce: cibaUserAuthNNonce, ciba\_url: cibaURL]$ 
12:      let  $message := \langle HTTPReq, \nu_{ciba}, GET, userEp.host, /start-ciba-authentication, \langle \rangle, \langle \rangle, body \rangle$ 
13:      call HTTPS_SIMPLE_SEND( $[responseTo: CibaUserAuthNReq], message, a, s'$ )
14:   stop
```

Algorithm 16 Relation of *script_as_form*: A login page for the user.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets \rangle$

- 1: **let** $url := \text{GETURL}(tree, docnonce)$
- 2: **let** $url' := \langle \text{URL}, S, url.host, /auth2, \rangle, \perp \rangle$
- 3: **let** $formData := scriptstate$
- 4: **let** $identity \leftarrow ids$
- 5: **let** $secret \leftarrow secrets$
- 6: **let** $formData[identity] := identity$
- 7: **let** $formData[password] := secret$
- 8: **let** $command := \langle \text{FORM}, url', \text{POST}, formData, \perp \rangle$
- 9: **stop** $\langle s, cookies, localStorage, sessionStorage, command \rangle$

Algorithm 17 Relation of *script_as_ciba_form*: A login page for the user for CIBA flows.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets \rangle$

- 1: **let** $url := \text{GETURL}(tree, docnonce)$
- 2: **let** $url' := \langle \text{URL}, S, url.host, /ciba-auth2, \rangle, \perp \rangle$
- 3: **let** $formData := [\text{ciba_auth2_reference}: scriptstate[\text{ciba_auth2_reference}]]$
- 4: **let** $bindingMessage := scriptstate[\text{binding_message}]$
- 5: **let** $clientDomain := scriptstate[\text{client_domain}]$
- 6: **let** $identity \leftarrow ids$
- 7: **let** $secret \leftarrow secrets$
- 8: **let** $formData[identity] := identity$
- 9: **let** $formData[password] := secret$
- 10: **let** $command := \langle \text{CIBAFORM}, url', \text{POST}, formData, \perp, clientDomain, bindingMessage \rangle$
- 11: **stop** $\langle s, cookies, localStorage, sessionStorage, command \rangle$

L. Resource Servers

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

Definition 9. A state $s \in Z^{rs}$ of a resource server rs is a term of the form $\langle \text{DNSaddress}, \text{pendingDNS}, \text{pendingRequests}, \text{corrupt}, \text{keyMapping}, \text{tlskeys}, \text{mtlsRequests}, \text{pendingResponses}, \text{resourceNonces}, \text{ids}, \text{asInfo}, \text{resourceASMapping}, \text{dpopNonces}, \text{jwk} \rangle$ with $\text{DNSaddress} \in \text{IPs}$, $\text{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$, $\text{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\text{keyMapping} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\text{tlskeys} \in [\text{Doms} \times K_{\text{TLS}}]$ (all former components as in [Definition 77](#)), $\text{mtlsRequests} \in \mathcal{T}_{\mathcal{N}}$, $\text{pendingResponses} \in \mathcal{T}_{\mathcal{N}}$, $\text{resourceNonces} \in [\text{ID} \times \mathcal{T}_{\mathcal{N}}]$, $\text{ids} \subset^{\langle \rangle} \text{ID}$, $\text{asInfo} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\text{resourceASMapping} \in [\text{resourceURLPath}^{rs} \times \mathcal{T}_{\mathcal{N}}]$, $\text{dpopNonces} \in \mathcal{T}_{\mathcal{N}}$, and $\text{jwk} \in K_{\text{sign}}$.

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

- $s_0^{rs}.\text{DNSaddress} \in \text{IPs}$,
- $s_0^{rs}.\text{pendingDNS} \equiv \langle \rangle$,
- $s_0^{rs}.\text{pendingRequests} \equiv \langle \rangle$,
- $s_0^{rs}.\text{corrupt} \equiv \perp$,
- $s_0^{rs}.\text{keyMapping}$ being the same as the keymapping for browsers,
- $s_0^{rs}.\text{tlskeys} \equiv \text{tlskeys}^{rs}$ (see [Appendix A-C](#)),
- $s_0^{rs}.\text{mtlsRequests} \equiv \langle \rangle$,
- $s_0^{rs}.\text{pendingResponses} \equiv \langle \rangle$,
- $s_0^{rs}.\text{resourceNonces}$ being a dictionary where the RS stores the resource nonces for each identity and resource id pair, initialized as $s_0^{rs}.\text{resourceNonces}[\text{id}][\text{resourceID}] := \langle \rangle$, $\forall \text{id} \in^{\langle \rangle} s_0^{rs}.\text{ids}, \forall \text{resourceID} \in \mathbb{S}$,
- $s_0^{rs}.\text{ids} \subset^{\langle \rangle} \text{ID}$ such that $\forall \text{id} \in s_0^{rs}.\text{ids} : \text{governor}(\text{id}) \in \text{supportedAuthorizationServer}^{rs}$, i.e., the RS manages only resources of identities that are governed by one of the AS supported by the RS,
- $s_0^{rs}.\text{asInfo}$: for each domain of a supported AS $\text{dom}_{as} \in \text{supportedAuthorizationServerDoms}^{rs}$, let $s_0^{rs}.\text{asInfo}$ contain a dictionary entry with the following values:
 - $s_0^{rs}.\text{asInfo}[\text{dom}_{as}][\text{as_introspect_ep}] \equiv \langle \text{URL}, \text{S}, \text{dom}_{as}, / \text{introspect}, \langle \rangle, \perp \rangle$ (the URL of the introspection endpoint of the AS)
 - $s_0^{rs}.\text{asInfo}[\text{dom}_{as}][\text{as_key}] \equiv \text{pub}(\text{signkey}(\text{dom}^{-1}(\text{dom}_{as})))$ being the verification key for the AS
 - $s_0^{rs}.\text{asInfo}[\text{dom}_{as}][\text{rs_credentials}]$ being a sequence s.t. $\forall c: c \in^{\langle \rangle} s_0^{rs}.\text{asInfo}[\text{dom}_{as}][\text{rs_credentials}] \Leftrightarrow (\exists \text{rsDom} \in \text{dom}(rs): c \equiv \text{secretOfRS}(\text{dom}_{as}, \text{rsDom}))$, i.e., the secrets used by the RS for authenticating at the AS

Hence, setting up the ASs supported by this RS,

- $s_0^{rs}.\text{resourceASMapping} \in \text{dom}(\text{authorizationServerOfResource}^{rs}(\text{resourceID})) \forall \text{resourceID} \in \text{resourceURLPath}^{rs}$ (a domain of the AS managing the resource identified by resourceID),
- $s_0^{rs}.\text{dpopNonces} \equiv \langle \rangle$, and
- $s_0^{rs}.\text{jwk} \equiv \text{signkey}(rs)$ (used for HTTP message signing, see [Appendix A-C](#)).

The relation R^{rs} is again based on the generic HTTPS server model (see [Appendix E-L](#)), for which the algorithms used for processing HTTP requests and responses are defined in [Algorithm 18](#) and [Algorithm 19](#).

Algorithm 18 Relation of RS R^{rs} – Processing HTTPS Requests

```
1: function PROCESS_HTTPS_REQUEST( $m, k, a, f, s'$ )
2:   if  $m.path \equiv /MTLS-prepare$  then
3:     let  $mtlsNonce := \nu_1$ 
4:     let  $clientKey := m.body[pub\_key]$  → Certificate is not required to be checked [7, Section 4.2]
5:     let  $s'.mtlsRequests := s'.mtlsRequests + \langle \rangle (mtlsNonce, clientKey)$ 
6:     let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle \rangle, enc_a(\langle mtlsNonce, s'.keyMapping[m.host] \rangle, clientKey) \rangle, k)$ 
7:     stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
8:   else if  $m.path \equiv /DPoP-nonce$  then
9:     let  $freshDpopNonce := \nu_{dpop}$ 
10:    let  $s'.dpopNonces := s'.dpopNonces + \langle \rangle freshDpopNonce$ 
11:    let  $m' := enc_s(\langle HTTPResp, m.nonce, 200, \langle \rangle, [nonce: freshDpopNonce] \rangle, k)$ 
12:    stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
13:   else
14:     let  $expectSignedRequest \leftarrow \{\top, \perp\}$  → Decide whether to expect a signed resource request (see also Section V-C).
15:     let  $resourceID := m.path$ 
16:     let  $responsibleAS := s'.resourceASMapping[resourceID]$ 
17:     if  $responsibleAS \equiv \langle \rangle$  then
18:       stop → Resource is not managed by any of the supported ASs
19:     let  $asInfo := s'.asInfo[responsibleAS]$ 
20:     if  $Authorization \notin m.headers$  then
21:       stop → Expected AT in Authorization header as mandated by FAPI 2.0
22:     let  $authnScheme := m.headers[Authorization].1$ 
23:     let  $accessToken := m.headers[Authorization].2$ 
24:     if  $authnScheme \equiv Bearer$  then → mTLS sender constraining
25:       let  $mtlsNonce := m.body[TLS\_binding]$ 
26:       let  $mtlsInfo$  such that  $mtlsInfo \in \langle \rangle s'.mtlsRequests \wedge mtlsInfo.1 \equiv mtlsNonce$  if possible; otherwise stop
27:       let  $s'.mtlsRequests := s'.mtlsRequests - \langle \rangle mtlsInfo$ 
28:       let  $mtlsKey := mtlsInfo.2$ 
29:       let  $cnfValue := [x5t\#S256: hash(mTlsKey)]$ 
30:     else if  $authnScheme \equiv DPoP$  then → DPoP sender constraining
31:       let  $dpopProof := m.headers[DPoP]$ 
32:       let  $dpopJwt := extractmsg(dpopProof)$ 
33:       let  $verificationKey := dpopJwt[headers][jwk]$ 
34:       if  $checksig(dpopProof, verificationKey) \neq \top \vee verificationKey \equiv \langle \rangle$  then
35:         stop → Invalid DPoP signature (or empty jwk header)
36:       let  $dpopClaims := dpopJwt[payload]$ 
37:       let  $reqUri := \langle URL, S, m.host, m.path, \langle \rangle, \perp \rangle$ 
38:       if  $dpopClaims[htm] \neq m.method \vee dpopClaims[htu] \neq reqUri$  then
39:         stop → DPoP claims do not match corresponding message
40:       if  $dpopClaims[nonce] \notin s'.dpopNonces$  then
41:         stop → Invalid DPoP nonce
42:       if  $dpopClaims[ath] \neq hash(accessToken)$  then
43:         stop → Invalid access token hash
44:       let  $s'.dpopNonces := s'.dpopNonces - \langle \rangle dpopClaims[nonce]$ 
45:       let  $cnfValue := [jkt: hash(verificationKey)]$ 
46:     else
47:       stop → Wrong Authorization header value
```

→ Algorithm continues on next page.

```

48:  let resource :=  $\nu_4$  → Generate a fresh resource nonce
49:  let accessTokenContent such that accessTokenContent  $\equiv$  extractmsg(accessToken)
    ↪ if possible; otherwise let accessTokenContent :=  $\diamond$ 
50:  if accessTokenContent  $\equiv$   $\diamond$  then → Not a structured AT, do Token Introspection
51:    let requestSignedIntrospecResponse  $\leftarrow$   $\{\top, \perp\}$  → Whether to request a signed introspection response (cf. Section V-C)
    → Store values for the pending request (needed when the RS gets the introspection response)
52:    let requestId :=  $\nu_2$ 
53:    let s'.pendingResponses[requestId] := [expectedCNF: cnfValue, requestingClient: f,
    ↪ resourceID: resourceID, originalRequest: m, originalRequestKey: k, resource: resource,
    ↪ requestSignedIntrospecResponse: requestSignedIntrospecResponse]
54:    let url := asInfo[as_introspect_ep]
55:    let rsCred  $\leftarrow$  asInfo[rs_credentials] → Secret for authenticating at the AS (see also Sec. 2.1 of RFC 7662 [41])
56:    let headers := [Authorization: (Basic, rsCred)]
57:    if requestSignedIntrospecResponse  $\equiv$   $\top$  then
58:      let headers[Accept] := app/token-introspection+jwt → Request signed introspection response [35, Sec. 4]
59:      let body := [token: accessToken]
60:      let message :=  $\langle$ HTTPReq,  $\nu_3$ , POST, url.domain, url.path, url.parameters, headers, body $\rangle$ 
61:      call HTTPS_SIMPLE_SEND(responseTo: TOKENINTROSPECTION, requestId: requestId,
    ↪ expectSignedRequest: expectSignedRequest], message, a, s')
    → If we make it here, the access token is a structured token
62:    if cnfValue.1  $\neq$  accessTokenContent[cnf].1  $\vee$  cnfValue.2  $\neq$  accessTokenContent[cnf].2 then
63:      stop → AT is bound to a different key
64:    if checksig(accessToken, asInfo[as_key])  $\neq$   $\top$  then
65:      stop → Verification of AT signature failed
66:    if expectSignedRequest  $\equiv$   $\top$  then
67:      let verificationKey := accessTokenContent[client_sig_key] → AS includes the client's HTTP Message Signing key
    in structured AT, see also Section V-C.
68:      let hasValidSignature := VERIFY_REQUEST_SIGNATURE(m, verificationKey)
69:      if hasValidSignature  $\neq$   $\top$  then
70:        stop
71:      let id := accessTokenContent[sub]
72:      if id  $\notin$   $\langle$  $\diamond$ s'.ids $\rangle$  then
73:        stop → RS does not manage resources of this RO
    → Token binding successfully checked, the RS gives access to a resource of the identity
74:      let s'.resourceNonces[id][resourceID] := s'.resourceNonces[id][resourceID] + $\diamond$  resource
75:      let body := [resource: resource] → This will be the resource response message body
76:      let signResResponse  $\leftarrow$   $\{\top, \perp\}$  → Whether to sign the resource response (cf. Section V-C)
77:      if signResResponse  $\equiv$   $\top$  then
78:        let headers := SIGN_RESOURCE_RESPONSE(body, s')
79:      else
80:        let headers :=  $\langle$  $\rangle$ 
81:      let m' := encs( $\langle$ HTTPResp, m.nonce, 200, headers, body $\rangle$ , k)
    → Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
82:      let leakingMessage :=  $\langle$ HTTPReq,  $\nu_{RRleak}$ , POST, m.domain, m.path, m.parameters, m.headers,  $\langle$  $\rangle$  $\rangle$ 
83:      let leakAddress  $\leftarrow$  IPs
84:      stop  $\langle$  $\langle$ f, a, m' $\rangle$ ,  $\langle$ leakAddress, a,  $\langle$ LEAK, leakingMessage $\rangle$  $\rangle$  $\rangle$ , s'

```

Algorithm 19 Relation of a Resource Server R^{rs} – Processing HTTPS Responses

```
1: function PROCESS_HTTPS_RESPONSE( $m$ ,  $reference$ ,  $request$ ,  $key$ ,  $a$ ,  $f$ ,  $s'$ )
2:   if  $reference[responseTo] \equiv \text{TOKENINTROSPECTION}$  then
3:     let  $pendingRequestInfo := s'.pendingResponses[reference[requestId]]$ 
4:     let  $s'.pendingResponses := s'.pendingResponses - reference[requestId]$ 
5:     let  $clientAddress := pendingRequestInfo[requestingClient]$ 
6:     let  $expectedCNF := pendingRequestInfo[expectedCNF]$ 
7:     let  $origReq := pendingRequestInfo[originalRequest]$ 
8:     let  $originalRequestKey := pendingRequestInfo[originalRequestKey]$ 
9:     let  $resourceID := pendingRequestInfo[resourceID]$ 
10:    let  $resource := pendingRequestInfo[resource]$ 
11:    let  $responsibleAS := s'.resourceASMapping[resourceID]$ 
12:    if  $responsibleAS \equiv \langle \rangle$  then
13:      stop  $\rightarrow$  Resource is not managed by any of the supported ASs
14:    let  $asInfo := s'.asInfo[responsibleAS]$ 
15:    if  $pendingRequestInfo[requestSignedIntrospecResponse] \equiv \top$  then
16:      if  $checksig(m.body, asInfo[as\_key]) \neq \top$  then
17:        stop
18:      let  $response := extractmsg(m.body)$ 
19:      if  $response[iss] \neq responsibleAS \vee response[aud] \neq m.host \vee token\_introspection \notin response$  then
20:        stop
21:      let  $m.body := response[token\_introspection]$   $\rightarrow$  Remove signature for uniform handling of  $m$  below
22:    if  $m.body[active] \neq \top$  then
23:      stop  $\rightarrow$  Access token was invalid
24:    let  $responseCNF := m.body[cnf]$ 
25:    if  $responseCNF.1 \neq expectedCNF.1 \vee responseCNF.2 \neq expectedCNF.2$  then
26:      stop  $\rightarrow$  Access token was bound to a different key
27:    let  $id := m.body[sub]$ 
28:    if  $id \notin s'.ids$  then
29:      stop  $\rightarrow$  RS does not manage resources of this RO
30:       $\rightarrow$  Handle signed resource requests (i.e., HTTP Message Signatures)
31:    let  $expectSignedRequest := reference[expectSignedRequest]$ 
32:    if  $expectSignedRequest \equiv \top$  then
33:      let  $verificationKey := m.body[client\_sig\_key]$   $\rightarrow$  AS includes the client's HTTP Message Signing key in introspection
34:       $\rightarrow$  Now that  $rs$  knows the client's HTTP Message Signing key, it can verify the signature on the resource request.
35:      let  $hasValidSignature := \text{VERIFY\_REQUEST\_SIGNATURE}(origReq, verificationKey)$ 
36:      if  $hasValidSignature \neq \top$  then
37:        stop
38:       $\rightarrow$  Token binding etc. successfully checked, the RS now gives access to a resource of the identity
39:    let  $s'.resourceNonces[id][resourceID] := s'.resourceNonces[id][resourceID] + \langle \rangle resource$ 
40:    let  $body := [resource: resource]$   $\rightarrow$  This will be the resource response message body
41:    if  $reference[signResResponse] \equiv \top$  then
42:      let  $headers := \text{SIGN\_RESOURCE\_RESPONSE}(body, s')$ 
43:    else
44:      let  $headers := \langle \rangle$ 
45:    let  $m' := enc_s((\text{HTTPResp}, origReq.nonce, 200, headers, body), originalRequestKey)$ 
46:     $\rightarrow$  Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
47:    let  $leakingMessage := \langle \text{HTTPReq}, \nu_{RRleak}, \text{POST}, origReq.domain, origReq.path, origReq.parameters, origReq.headers, [] \rangle$ 
48:    let  $leakAddress \leftarrow \text{IPs}$ 
49:    stop  $\langle \langle f, a, m' \rangle, \langle leakAddress, a, \langle \text{LEAK}, leakingMessage \rangle \rangle \rangle, s'$ 
50:  stop  $\rightarrow$  Unknown response type
```

Algorithm 20 Relation of a Resource Server R^{rs} – Create the headers to sign a resource response

```
1: function SIGN_RESOURCE_RESPONSE(body, s')
2:   let headers := [Content-Digest: hash(body)] → See [22, Sec. 5.6.2.1 No. 6]
   → See [22, Sec. 5.6.2.1]. In our model, the RS never includes the request signature in a response signature (this would only add
   components to the signature, hence if anything, making the response “more secure” w.r.t. non-repudiation – however, we are able
   to prove non-repudiation even without this).
3:   let coveredComponents := {(@status, {}), (content-digest, {}), [tag: fapi-2-response, keyid: pub(s'.jwk)]}
4:   let signatureBase := [(@status, {}): 200, (content-digest, {}): headers[Content-Digest]]
5:   let signatureBase := signatureBase + {} coveredComponents.2 → Add signature parameters [2, Sec. 2.5]
6:   let headers[Signature] := [res: sig(signatureBase, s'.jwk)]
7:   let headers[Signature-Input] := [res: coveredComponents]
8:   return headers
```

Algorithm 21 Relation of a Resource Server R^{rs} – Verify the signature on a resource request

```
1: function VERIFY_REQUEST_SIGNATURE(m, verificationKey) → m is the resource request
2:   if Signature ∈ m.headers then
3:     if hash(m.body) ≠ m.headers[Content-Digest] then
4:       return ⊥ → Content-digest is required by FAPI 2.0 Message Signing [22, Sec. 5.6.1.2]
5:     let coveredComponents := m.headers[Signature-Input][req]
6:     let signerSignatureBase := extractmsg(m.headers[Signature][req])
7:     if @method ∉ coveredComponents.1 ∨ content-digest ∉ coveredComponents.1 ∨
       ↪ @target-uri ∉ coveredComponents.1 ∨ authorization ∉ coveredComponents.1 ∨
       ↪ coveredComponents.2[tag] ≠ fapi-2-request then
8:       return ⊥ → See [22, Sec. 5.6.1.2], these components must be present
9:     if signerSignatureBase.2[tag] ≠ fapi-2-request ∨ keyid ∉ signerSignatureBase.2 then
10:      stop
11:     for component ∈ coveredComponents.1 do
12:       let isComponentEqual := IS_COMPONENT_EQUAL(m, {}, signerSignatureBase, component)
13:       if isComponentEqual ≠ ⊤ then
14:         return ⊥
         → If we make it here, the request signature base matches the actual request data.
15:     if verificationKey ≡ {} ∨ checksig(m.headers[Signature][req], verificationKey) ≠ ⊤ then
16:       return ⊥ → Invalid public key/message or signature does not verify
17:     return ⊤ → If we make it here, the request signature is fully verified.
18:   else
19:     return ⊥ → Missing signature header
```

APPENDIX B

FAPI 2.0 WEB SYSTEM

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

- $\mathcal{W} = \text{Hon} \cup \text{Net}$ 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 $\text{Hon} := \text{B} \cup \text{C} \cup \text{AS} \cup \text{RS}$. DNS servers are subsumed by the network attacker and are therefore not modeled explicitly.
- \mathcal{S} contains the scripts shown in Table I, with string representations defined by the mapping script .
- E^0 contains only the trigger events.

$s \in \mathcal{S}$	$\text{script}(s)$
R^{att}	att_script
$\text{script_client_index}$	script_client_index
script_as_form	script_as_form

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

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

APPENDIX C

FORMAL SECURITY PROPERTIES

In this section, we present our formal security properties for FAPI 2.0 ecosystems (within our model, i.e., FAPI 2.0 Websystems). However, in order to do so, we first need some definitions.

Notion of an AT being bound to key, AS, Client Id, and identity. The following definition captures that an access token was issued by an authorization server as , bound to a key k , and a client id $clientId$, and is associated with an identity id . This definition is needed in the subsequent definitions.

Definition 10 (Access Token bound to Key, Authorization Server, Client Id, and Identity). Let $k \in \mathcal{T}_{\mathcal{K}}$ be a term, $as \in \text{AS}$ an authorization server, $clientId$ a client identifier, and $id \in \text{ID}$ an identity. We say that a term t is an *access token bound to k , as , $clientId$, and id* in state S of the configuration (S, E, N) of a run ρ of a FAPI web system \mathcal{FAPI} , if there exists an entry $rec \in {}^\diamond S(as).\text{records}$ such that

$$rec[\text{access_token}] \equiv t \wedge \tag{1}$$

$$rec[\text{subject}] \equiv id \wedge \tag{2}$$

$$rec[\text{client_id}] \equiv clientId \wedge \tag{3}$$

$$((rec[\text{cnf}] \equiv [\text{jkt} : \text{hash}(k)]) \vee \tag{4}$$

$$(rec[\text{cnf}] \equiv [\text{x5t}\#\text{S256} : \text{hash}(k)])) \tag{5}$$

In a bit more detail:

- (1) captures that the AS as created the access token.
- (2) captures that the access token is associated with identity id (i.e., this identity authenticated previously at the authorization endpoint of the AS, and when the AT is redeemed at a RS, the RS will provide access to resources of this identity).
- (3) captures that t was created for a client with client identifier $clientId$ at as .
- (4) and (5) capture that the access token is bound to a key. If (4) holds, then we say that the access token is bound via DPoP, otherwise, the token is bound via mTLS.

Notion of a client id being issued to a client by an authorization server. With this definition, we capture that an authorization server as issued a client identifier $clientId$ to a client c (as part of Dynamic Client Registration) in a processing step.

Definition 11 (Client Identifier Issued to Client by AS). We say that a *client identifier $clientId$ has been issued to c by as in processing step P in a run ρ* (of a FAPI websystem \mathcal{FAPI}), if all of the following hold true:

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

$$(ii) e_{\text{in}}^P = \langle x^P, y^P, m^P \rangle, \text{ with } m^P = \text{enc}_a(\langle \text{regReq}, k \rangle, pk_{\text{as}}), \text{ where } \text{regReq} \text{ matches } \langle \text{HTTPReq}, n, \text{POST}, d_{\text{as}}, *, *, *, \text{regData} \rangle \text{ (Definition 37).}$$

- (iii) There is a processing step $Q = (S^q, E^q, N^q) \xrightarrow[c \rightarrow E_{\text{out}}^Q]{e_{\text{in}}^Q \rightarrow c} (S^{q+1}, E^{q+1}, N^{q+1})$ prior to P in ρ such that there is an event $\langle x, y, m^P \rangle \in E_{\text{out}}^Q$, i.e., c emits m^P in Q (Definition 78).
- (iv) $E_{\text{out}}^P = \langle \langle y^P, x^P, \text{resp} \rangle \rangle$, with $\text{resp} = \text{enc}_s(\langle \text{HTTPResp}, n, 201, \langle \rangle, \text{regResp} \rangle, k)$ (i.e., a response to the request in m^P), where $\text{regResp}[\text{client_id}] = \text{clientId}$.

Lemma 1. If a client identifier clientId has been issued to c by an honest $\text{as} \in \text{AS}$ in processing step $P = (S, E, N) \rightarrow (S', E', N')$ in a run ρ , then all of the following hold true:

- (I) Process as finished P by executing Line 26 of Algorithm 13.
- (II) We have $\text{clientId} \in S'(\text{as}).\text{clients}$.
- (III) Condition (ii) in Definition 11 is implied by condition (iv).

PROOF. (I). An honest AS only outputs an HTTPS response with code 201 (as it does in P by Definition 11) in two places: In Line 144 of Algorithm 11, the response body is a dictionary with only one key, namely `request_uri`, i.e., does not contain a key `client_id` (see (iv) in Definition 11). The second place is Line 26 of Algorithm 13, where the response body is a dictionary which indeed contains a key `client_id` (Line 13 of Algorithm 13). Hence, we have that as must have finished P by executing Line 26 of Algorithm 13.

(II). To reach Line 26 of Algorithm 13, as must have executed Line 24 of Algorithm 13, which immediately gives us (together with Line 26) $\text{clientId} \in S'(\text{as}).\text{clients}$.

(III). An honest AS only outputs an event as described in condition (iv) of Definition 11 in Line 26 of Algorithm 13 (cf. (I) above). Hence, as must have executed Algorithm 13 in P . Algorithm 13, in turn, is only called in Lines 18f. of Algorithm 11, and only if the method field of the first argument to Algorithm 11 is POST. Furthermore, Algorithm 11 is only ever called in Line 9 of Algorithm 39, where the input event must match $\text{enc}_a(\langle \langle \text{HTTPReq}, *, *, *, *, *, *, * \rangle, * \rangle, *)$ (see Line 8 of Algorithm 39). Therefore, the format of the input event e_{in}^P must be as described in condition (ii).

However, we still have to prove that the nonce n from condition (ii) is indeed the same as in condition (iv), and the same for the addresses y^P and x^P .

As for the nonce n in condition (iv), it is set by the AS in Line 25 of Algorithm 13 to the nonce of the input message, which in turn is the first argument to Algorithm 13, which originates from Lines 18f. of Algorithm 11, i.e., with Line 9 of Algorithm 39, this is the nonce of the input event (and hence, the same n as in condition (ii)).

For x^P and y^P , the same argumentation as for n applies (except that the values are the third, resp. fourth argument to Algorithm 13). ■

Notion of an HTTPS response to an HTTPS request. With this definition, we capture that some process p sent an HTTPS request to some process p' in processing step R , and p' responds to this request with an HTTPS response in processing step Q .

Definition 12 (HTTPS Response to HTTPS Request Sent by p to p'). Let $p, p' \in C \cup \text{AS} \cup B \cup \text{RS}$, and ρ some run of a FAPI websystem \mathcal{FAPI} with network attacker. We say that resp is an HTTPS response to an HTTPS Request req sent by p to p' , if all of the following are true:

- (i) $\text{resp} \in \text{HTTPSResponses}$, i.e., $\text{resp} \sim \text{enc}_s(\langle \text{HTTPResp}, n, *, *, * \rangle, k)$
- (ii) $\text{req} \in \text{HTTPSRequests}$, i.e., $\text{req} \sim \text{enc}_a(\langle \langle \text{HTTPReq}, n', *, *, *, *, *, * \rangle, k' \rangle, \text{pubKey})$
- (iii) $\exists d_{p'} \in \text{dom}(p')$ such that $\text{tlskey}(d_{p'}) \equiv \text{pubKey}$
- (iv) $k \equiv k'$
- (v) $n \equiv n'$
- (vi) There is a processing step $Q = (S^q, E^q, N^q) \xrightarrow[p' \rightarrow E_{\text{out}}^Q]{} (S^{q'}, E^{q'}, N^{q'})$ in ρ , such that there is an event $\langle x, y, \text{resp} \rangle \in E_{\text{out}}^Q$.
- (vii) Prior to Q , there is a processing step $R = (S^r, E^r, N^r) \xrightarrow[p \rightarrow E_{\text{out}}^R]{} (S^{r'}, E^{r'}, N^{r'})$ in ρ , such that there is an event $\langle x', y', \text{req} \rangle \in E_{\text{out}}^R$.

A. Authorization, Authentication and Session Integrity Properties

In the following, we describe and define our formal security properties for authorization, authentication, and session integrity for both authentication and authorization. We expect these properties to hold for all possible configurations of a FAPI 2.0 ecosystem, including dynamic client registration, dynamic client management, the FAPI 2.0 Message Signing profiles in any combination (including not using FAPI 2.0 Message Signing at all), and parallel FAPI-CIBA flows.

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). In a bit more detail, our 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 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 [24] on how our model manages resources). (ii) The token at is bound to some key k , as , and the user identity id (see Definition 10). (iii) If k is the key of an honest client, then the attacker cannot derive the resource.

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.

Definition 13 (Authorization Property). We say that a FAPI web system $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$ is *secure w.r.t. authorization* iff for every run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$, every RS $rs \in \text{RS}$ that is honest in S^n , every identity $id \in \langle \rangle s_0^{rs}.\text{ids}$ with $b = \text{ownerOfID}(id)$ being an honest browser in S^n , every processing step $Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \rightarrow E_{\text{out}}^Q]{e_{\text{in}}^Q \rightarrow rs} (S^{Q'}, E^{Q'}, N^{Q'})$ in ρ , every $\text{resourceID} \in \mathbb{S}$ with $as = \text{authorizationServerOfResource}^{rs}(\text{resourceID})$ being honest in S^Q , it holds true that:⁷

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

- (i) \exists a processing step $P = s_i \xrightarrow[rs \rightarrow E_{\text{out}}^P]{e_{\text{in}}^P \rightarrow rs} s_{i+1}$ such that
 - (i.a) either $P = Q$, or P is prior to Q in ρ , and
 - (i.b) e_{in}^P is an event $\langle x, y, \text{enc}_a(\langle m_{\text{req}}, k_1 \rangle, k_2) \rangle$ 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 Authorization header, i.e., $t \equiv m_{\text{req}}.\text{headers}[\text{Authorization}].2$, and
 - (i.c) r is a fresh nonce generated in P at the resource endpoint of rs in Line 48 of Algorithm 18
- (ii) t is bound to a key $k \in \mathcal{T}_{\mathcal{N}_C}$, as , a client identifier clientId , and id in S^Q (see Definition 10).
- (iii) If there exists a client $c \in \mathcal{C}$ such that clientId has been issued to c by as in a processing step R prior to P in ρ , and if c is honest in S^n , then r is not derivable from the attackers knowledge in S^n , i.e., $r \notin d_\emptyset(S^n(\text{attacker}))$.

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 `sessions` state subterm, and associates with it the identity that is logged in to the session (the identity is taken from an id token). 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.

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 14 (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 $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$ iff there exists some session id x and a domain $d \in \text{dom}(\text{governor}(id))$ such that $S(c).\text{sessions}[x][\text{loggedInAs}] \equiv \langle d, id \rangle$ and $S(c).\text{sessions}[x][\text{serviceSessionId}] \equiv n$.

Definition 15 (Authentication Property). We say that a FAPI 2.0 web system with network attacker $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$ is *secure w.r.t. authentication* iff for every run ρ of $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$, every configuration (S, E, N) in ρ , every $c \in \mathcal{C}$ that is honest in S , every identity $id \in \text{ID}$ with $as = \text{governor}(id)$ being an honest AS (in S) and with $b = \text{ownerOfID}(id)$ 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., $n \notin d_\emptyset(S(\text{attacker}))$).

⁷`authorizationServerOfResource` ^{rs} is a mapping from resource ids to the authorization server that manages the respective resource, see Definition 6.

3) 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.

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

Definition 16 (User is logged in). For a run ρ of a FAPI web system with a network attacker $\mathcal{F}API$ 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

$$Q = (S, E, N) \xrightarrow{c \rightarrow E_{out}} (S', E', N')$$

and some event $\langle y, y', m \rangle \in E_{out}$ 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 $\langle \text{Set-Cookie}, [\text{serviceSessionId}: \langle ssid, \top, \top, \top \rangle] \rangle$ for some nonce $ssid$ such that $S(c).sessions[lsid][\text{serviceSessionId}] \equiv ssid$ and $S(c).sessions[lsid][\text{loggedInAs}] \equiv \langle d, id \rangle$ with $d \in \text{dom}(as)$. We then write $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$.

Definition 17 (User started authorization code login flow). For a run ρ of a FAPI web system with a network attacker $\mathcal{F}API$ 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 Line 8 of Algorithm 10, 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 $\langle \text{Set-Cookie}, [_Host, \text{sessionId}: \langle lsid, \top, \top, \top \rangle] \rangle$. We then write $\text{started}_\rho^Q(b, c, lsid)$.

Definition 18 (User started CIBA login flow). For a run ρ of a FAPI web system with a network attacker $\mathcal{F}API$ we say that the user of the browser b started a CIBA login session identified by a nonce $lsid$ at the client c in a processing step Q in ρ if in that processing step, (1) the browser b emits an HTTPS request with a payload matching $\langle \text{HTTPReq}, *, *, clientDom, /start_ciba, *, \langle \rangle, body \rangle$, with $clientDom \in \text{dom}(c)$, and (2) c (in some later processing step) 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 $\langle \text{Set-Cookie}, [_Host, \text{sessionId}: \langle lsid, \top, \top, \top \rangle] \rangle$. We then write $\text{startedCIBA}_\rho^Q(b, c, lsid)$.

Definition 19 (User authenticated at an AS for authorization code flow). For a run ρ of a FAPI web system with a network attacker $\mathcal{F}API$ 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 $Q = (S, E, N) \rightarrow (S', E', N')$ 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 16, selected the identity id , and (2) we have that

- the $scriptstate$ of that document, when triggered in Q , contains a nonce $auth2Reference$ such that $scriptstate[\text{auth2_reference}] \equiv auth2Reference$, and
- there is a nonce $requestUri$ such that $S(as).authorizationRequests[requestUri][\text{auth2_reference}] \equiv auth2Reference$, and
- $S(c).sessions[lsid][\text{request_uri}] \equiv requestUri$.

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

Definition 20 (User authenticated at an AS for CIBA flow). For a run ρ of a FAPI web system with a network attacker $\mathcal{F}API$ 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 $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ in which the browser b was triggered, selected a document loaded from an origin of as , executed the script $script_as_ciba_form$ in that document, and in that script, (1) in Line 6 of Algorithm 17, selected the identity id , and (2) we have that

- the $scriptstate$ of that document, when triggered in Q , contains a nonce $auth2Reference$ such that $scriptstate[\text{ciba_auth2_reference}] \equiv auth2Reference$, and
- there is a nonce $authnReqId$ such that $S(as).cibaAuthnRequests[authnReqId][\text{ciba_auth2_reference}] \equiv auth2Reference$, and
- $S(c).sessions[lsid][\text{auth_req_id}] \equiv authnReqId$.

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

Definition 21 (Resource Access). For a run ρ of a FAPI web system with a network attacker $\mathcal{F}API$ we say that a browser $b \in \mathbb{B}$ gets access to a resource r 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 $Q = (S, E, N) \xrightarrow{c \rightarrow E_{out}} (S', E', N')$ in ρ if

- $S(c).sessions[lsid][\text{cibaFlow}] \equiv \perp$ and c executes Line 114 of Algorithm 3 in Q , or

- $\text{resource} \in S(c).\text{sessions}[lsid]$ and c executes Line 68 of Algorithm 2 in Q ,

includes the resource r in the body of the HTTPS response that is sent out there (i.e., $\exists \langle x', y', m \rangle \in {}^\diamond E_{\text{out}}$ such that $m \sim \text{enc}_s(\langle \text{HTTPResp}, *, *, *, r \rangle, *)$), and it holds true that

- (i) $r \in {}^\diamond S'(rs).\text{resourceNonces}[u][\text{resourceId}]$ and $as = \text{authorizationServerOfResource}^{rs}(\text{resourceID})$ (for some value $\text{resourceId} \in \mathcal{T}_{\mathcal{N}}$), and
- (ii) $\langle \langle _Host, \text{sessionId} \rangle, \langle \text{lsid}, y, z, z' \rangle \rangle \in {}^\diamond S'(b).\text{cookies}[d]$ for $d \in \text{dom}(c)$, $y, z, z' \in \mathcal{T}_{\mathcal{N}}$, and
- (iii) $S'(c).\text{sessions}[lsid][\text{resourceServer}] \in \text{dom}(rs)$, and
- (iv) the request to which the client is responding in Q contains a Cookie header with the cookie $\langle _Host, \text{sessionId} \rangle$ with the value $lsid$.

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

Definition 22 (Client Leaked Authorization Request). Let \mathcal{FAPI} be an FAPI web system with a network attacker. For a run ρ of \mathcal{FAPI} with a processing step Q , a client $c \in \mathcal{C}$, a browser b , an authorization server $as \in \text{AS}$, an identity id , a login session id $lsid$, and $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$, we say that c leaked the authorization request for $lsid$, if there is a processing step $Q' = (S, E, N) \xrightarrow{c \rightarrow E_{\text{out}}} (S', E', N')$ in ρ prior to Q such that in Q' , c executes Line 75 of Algorithm 3 and there is a nonce requestUri and an event $\langle x, y, m \rangle \in E_{\text{out}}$ with $m.1 \equiv \text{LEAK}$ and $m.2.\text{parameters}[\text{request_uri}] \equiv \text{requestUri}$ such that $S'(c).\text{sessions}[lsid][\text{request_uri}] \equiv \text{requestUri}$.

4) Session Integrity Property for Authentication

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

Definition 23 (Session Integrity for Authentication for Authorization Code Flows). Let \mathcal{FAPI} be an FAPI web system with a network attacker. We say that \mathcal{FAPI} is secure w.r.t. session integrity for authentication for authorization code flows iff for every run ρ of \mathcal{FAPI} , every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , every browser b that is honest in S , every $as \in \text{AS}$, every identity id , every client $c \in \mathcal{C}$ that is honest in S , every nonce $lsid$ with $S(c).\text{sessions}[lsid][\text{cibaFlow}] \equiv \perp$, and $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 22), we have that (1) there exists a processing step Q' in ρ (before Q) such that $\text{started}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S , then there exists a processing step Q'' in ρ (before Q) such that $\text{authenticated}_\rho^{Q''}(b, c, id, as, lsid)$.

For the session integrity properties of CIBA flows, we need the following assumption on how browsers (and hence, the users modeled as part of the browsers) select an identity owned by them (i.e., owned by the browser) when initiating CIBA flows (note that in our model, from the point of view of a client, a CIBA flow is started by an HTTPS request to the client's `/start-ciba` endpoint with an identity and an AS identifier, i.e., domain – the identity is then used as a login hint to initiate a CIBA flow at the selected AS).

Note that if the initiating client in a real-world protocol flow with CIBA – for whatever reason – sends the “wrong” login hint, then the AS will ask the “wrong” user to authenticate and authorize the request. While an honest user might decline such a request, an attacker (aiming to break session integrity) would happily authorize such a request. As the client has no way of knowing who really authenticated at the AS, it cannot distinguish this case from an honest flow. I.e., the assumption that the client selects the “correct” login hint is necessary – otherwise, session integrity is easily broken, and there is no evidence that FAPI-CIBA aims to protect in these cases on a protocol level.

Assumption 1 (Honest Browsers Always Select Owned Identity for CIBA). Let \mathcal{FAPI} be an FAPI web system with a network attacker. In every run ρ of \mathcal{FAPI} , every processing step $Q = (S, E, N) \xrightarrow{b \rightarrow E_{\text{out}}} (S', E', N')$ in ρ , every browser b that is honest in S , every event $\langle x, y, m \rangle \in E_{\text{out}}$, we have that if $m \sim \text{enc}_a(\langle \langle \text{HTTPReq}, *, *, *, /start-ciba, *, *, body \rangle, *, * \rangle, *)$, then $body \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\text{identity} \in body$, and $\text{ownerOfID}(body[\text{identity}]) \equiv b$.

Definition 24 (Session Integrity for Authentication for FAPI-CIBA). Let \mathcal{FAPI} be an FAPI web system with a network attacker. We say that \mathcal{FAPI} is secure w.r.t. session integrity for authentication for CIBA flows iff for every run ρ of \mathcal{FAPI} , every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , every browser b that is honest in S and behaves according to Assumption 1, every $as \in \text{AS}$, every identity id , every client $c \in \mathcal{C}$ that is honest in S , every nonce $lsid$ with $S(c).\text{sessions}[lsid][\text{cibaFlow}] \equiv \top$, and $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$ we have that (1) there exists a processing step Q' in ρ (before Q) such that $\text{startedCIBA}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S , then there exists a processing step Q'' in ρ (before Q) such that $\text{authenticatedCIBA}_\rho^{Q''}(b, c, id, as, lsid)$.

By *session integrity for authentication* we denote the conjunction of both session integrity for authentication for authorization code flows and FAPI-CIBA (Definition 23 and Definition 24).

5) 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 for Authorization Code Flows). Let $\mathcal{F}API$ be a FAPI web system with a network attacker. We say that $\mathcal{F}API$ is secure w.r.t. session integrity for authorization for authorization code flows iff for every run ρ of $\mathcal{F}API$, every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , every browser b that is honest in S , every $as \in AS$, every identity u , every client $c \in C$ that is honest in S , every $rs \in RS$ that is honest in S , every nonce r , every nonce $lsid$ with $S(c).sessions[lsid][cibaFlow] \equiv \perp$, we have that if $accessesResource_{\rho}^Q(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 22), then (1) there exists a processing step Q' in ρ (before Q) such that $started_{\rho}^{Q'}(b, c, lsid)$, and (2) if as is honest in S , then there exists a processing step Q'' in ρ (before Q) such that $authenticated_{\rho}^{Q''}(b, c, u, as, lsid)$.

Definition 26 (Session Integrity for Authorization for FAPI-CIBA). Let $\mathcal{F}API$ be a FAPI web system with a network attacker. We say that $\mathcal{F}API$ is secure w.r.t. session integrity for FAPI-CIBA iff for every run ρ of $\mathcal{F}API$, every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , every browser b that is honest in S and behaves according to Assumption 1, every $as \in AS$, every identity u , every client $c \in C$ that is honest in S , every $rs \in RS$ that is honest in S , every nonce r , every nonce $lsid$ with $S(c).sessions[lsid][cibaFlow] \equiv \top$, we have that if $accessesResource_{\rho}^Q(b, r, u, c, rs, as, lsid)$, then (1) there exists a processing step Q' in ρ (before Q) such that $startedCIBA_{\rho}^{Q'}(b, c, lsid)$, and (2) if as is honest in S , then there exists a processing step Q'' in ρ (before Q) such that $authenticatedCIBA_{\rho}^{Q''}(b, c, u, as, lsid)$.

By session integrity for authorization we denote the conjunction of both session integrity for authorization for authorization code flows and FAPI-CIBA (Definition 25 and Definition 26).

By session integrity we denote the conjunction of both session integrity for authorization and authentication.

B. Non-Repudiation Properties

Our non-repudiation properties capture that if some honest party accepts a message it expected to be signed, then – if the used signing key belongs to an honest party – that honest party actually signed the message in question.

1) Signed Authorization Requests

Definition 27 (Non-Repudiation for Signed Authorization Requests). Let $\mathcal{F}API$ be a FAPI web system with a network attacker. We say that $\mathcal{F}API$ is secure w.r.t. non-repudiation for signed authorization requests iff for every run ρ of $\mathcal{F}API$, every configuration (S^n, E^n, N^n) in ρ , every process $as \in AS$ that is honest in S^n , every request uri $requestUri$, we have that if $S^n(as).authorizationRequests[requestUri][signed_par] \equiv \top$, then all of the following hold true:

- (I) There exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \rightarrow as} (S', E', N')$ with (S, E, N) prior to (S^n, E^n, N^n) in ρ , such that $requestUri \notin S(as).authorizationRequests$ and $requestUri \in S'(as).authorizationRequests$.
- (II) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $enc_a(\langle \langle HTTPReq, \cdot, POST, selectedAS, /par, \cdot, \langle \rangle, body \rangle, \cdot \rangle, \cdot)$, where $body$ is of the form $sig(par, signKey)$ and $selectedAS \in \text{dom}(as)$.
- (III) If there is a process $c \in C$ which is honest in S^n , and a configuration (S^i, E^i, N^i) in ρ with $S^i(c).asAccounts[selectedAS][sign_key] \equiv signKey$, then there is a processing step $P = (S^j, E^j, N^j) \xrightarrow{c \rightarrow E_{out}} (S^{j+1}, E^{j+1}, N^{j+1})$ in ρ prior to Q during which c signed par (as contained in e_{in}) in Line 63 of Algorithm 8.

Informally, (I) captures that as accepted a PAR in processing step Q and issued $requestUri$ to identify that PAR. With (II), we require such a PAR to have a valid signature for some key $signKey$ on it. Finally, (III) captures that if the signature is valid for a key which an honest client registered with as , then it was indeed that exact client which signed the PAR.

2) Signed Authorization Responses

Definition 28 (Non-Repudiation for Signed Authorization Responses). Let $\mathcal{F}API$ be a FAPI web system with a network attacker. We say that $\mathcal{F}API$ is secure w.r.t. non-repudiation for signed authorization responses iff for every run ρ of $\mathcal{F}API$, every configuration (S^n, E^n, N^n) in ρ , every session id $sessionId$, every process $c \in C$ that is honest in S^n , we have that if

- (1) there exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \rightarrow c} (S', E', N')$ with (S, E, N) prior to (S^n, E^n, N^n) in ρ such that $redirectEpRequest \notin S(c).sessions[sessionId]$ and $redirectEpRequest \in S'(c).sessions[sessionId]$, and

- (2) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $\text{enc}_a(\langle \langle \text{HTTPReq}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle, / \text{redirect_ep}, \text{parameters}, \text{headers}, \cdot, \cdot, \cdot \rangle, \cdot)$, and
- (3) $S^n(c).\text{sessions}[\text{sessionId}][\text{requested_signed_authz_response}] \equiv \top$,

then all of the following hold true:

- (I) The term parameters from (2) above is a dictionary with at least a key response with value $\text{sig}(\text{authzResponse}, \text{signKey})$, with authzResponse being a dictionary with at least the keys iss and code .
- (II) If there is an $as \in \text{AS}$ with $S^n(as).\text{jwt} \equiv \text{signKey}$, and as honest in S^n , then there is a processing step $P = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ prior to Q in ρ , and as signed authzResponse (as contained in e_{in}) during P in Line 97 of Algorithm 11.

Informally, (1) captures that c accepted an authorization response in some processing step Q , (2) and (3) capture that c expected this response to be signed. Given these conditions, (I) captures that the response was indeed signed, and (II) ensures that if the key used to signed the response belongs to an honest AS, then this AS indeed signed the authorization response.

3) Signed Introspection Responses

Definition 29 (Non-Repudiation for Signed Introspection Responses). Let \mathcal{FAPI} be a FAPI web system with a network attacker. We say that \mathcal{FAPI} is secure w.r.t. non-repudiation for signed introspection responses iff for every run ρ of \mathcal{FAPI} , every configuration (S^n, E^n, N^n) in ρ , every process $rs \in \text{RS}$ that is honest in S^n , every request id requestId , we have that if there exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \rightarrow rs} (S', E', N')$ in ρ such that $S(rs).\text{pendingResponses}[\text{requestId}][\text{requestSignedIntrospecResponse}] \equiv \top$, and $\text{requestId} \notin S'(rs).\text{pendingResponses}$, and (S, E, N) prior to (S^n, E^n, N^n) in ρ , then all of the following hold true:

- (I) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $\text{enc}_s(\langle \langle \text{HTTPResp}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle, \text{body} \rangle, \cdot)$, where body is of the form $\text{sig}(\text{introspecResponse}, \text{signKey})$.
- (II) If there is an $as \in \text{AS}$ with $S^n(as).\text{jwt} \equiv \text{signKey}$, and as honest in S^n , then there is a processing step $P = (S^i, E^i, N^i) \xrightarrow[as \rightarrow E_{out}]{\quad} (S^{i+1}, E^{i+1}, N^{i+1})$ prior to Q in ρ , and as signed introspecResponse (as contained in e_{in} above) during P in Line 227 of Algorithm 11.

Informally, the precondition about Q captures that rs accepted an introspection response during Q , and expected that response to be signed (which rs would have indicated the corresponding introspection request by setting the Accept header to an appropriate value). The postconditions then capture that the introspection response was indeed signed and that – if the used signing key belongs to an honest AS – that honest AS indeed signed the introspection response.

4) Signed Resource Requests

Definition 30 (Non-Repudiation for Signed Resource Requests). Let \mathcal{FAPI} be a FAPI web system with a network attacker. We say that \mathcal{FAPI} is secure w.r.t. non-repudiation for signed resource requests iff for every run ρ of \mathcal{FAPI} , every configuration (S^n, E^n, N^n) in ρ , every process $rs \in \text{RS}$ that is honest in S^n , we have that if

- (1) there exists a processing step $Q = (S, E, N) \xrightarrow[rs \rightarrow E_{out}]{\quad} (S', E', N')$ in ρ such that $E_{out} = \langle \langle x, y, \text{resRes} \rangle, \text{leakedRequest} \rangle$, with (S, E, N) prior to (S^n, E^n, N^n) , and
- (2) during Q , either Line 69 of Algorithm 18 or Line 33 of Algorithm 19 was executed,

then all of the following hold true:

- (I) resRes is of the form $\text{enc}_s(\langle \langle \text{HTTPResp}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle, \text{body} \rangle, \cdot)$ with $\text{body} \equiv [\text{resource: resource}]$.
- (II) There exists a processing step $R = s^r \xrightarrow{e_{in} \rightarrow rs} s^{r'}$ prior or equal to Q in ρ such that $e_{in} = \langle y, x, \text{resReq} \rangle$, and rs generated resource during R in Line 48 of Algorithm 18.
- (III) resReq is of the form $\text{enc}_a(\langle \langle \langle \text{HTTPReq}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle, \text{method}, \text{host}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle, \cdot, \cdot \rangle, \cdot)$ with $\text{Signature} \in \text{headers}$, $\text{Signature-Input} \in \text{headers}$, and $\text{headers}[\text{Signature}]$ being a dictionary with at least a key req with value $\text{sig}(\text{signatureBase}, \text{clientSignKey})$.
- (IV) $\text{headers}[\text{Signature-Input}][\text{req}]$ is a sequence $\langle \text{coveredComponents}, \text{metadata} \rangle$ (there may be additional sequence elements after those two), where metadata is a dictionary with at least a key tag with value fapi-2-request , and coveredComponents is a sequence with at least the following elements: $\langle @method, \langle \rangle \rangle$, $\langle @target-uri, \langle \rangle \rangle$, $\langle authorization, \langle \rangle \rangle$, and $\langle content-digest, \langle \rangle \rangle$.
- (V) signatureBase is of the form $[\langle @method, \langle \rangle \rangle: \text{method}, \langle @target-uri, \langle \rangle \rangle: \langle \text{URL}, \text{S}, \text{host}, \text{path}, \text{parameters}, \perp \rangle, \langle authorization, \langle \rangle \rangle: \text{headers}[\text{Authorization}], \langle content-digest, \langle \rangle \rangle: \text{hash}(\text{body})] + \langle \rangle [\text{tag: fapi-2-request}, \text{keyid: keyId}]$ for some keyId ; however, the dictionaries may contain additional elements.

- (VI) If there is a client $c \in \mathcal{C}$ which is honest in S^n , a domain $selectedAS$, and an index $j \leq n$ such that $S^j(c).asAccounts[selectedAS][sign_key] \equiv clientSignKey$, then there is a processing step $P = (S^i, E^i, N^i) \xrightarrow{c \rightarrow E'_{out}} (S^{i+1}, E^{i+1}, N^{i+1})$ prior to R in ρ , and c signed $signatureBase$ (as contained in e_{in} above) during P in Line 39 of Algorithm 6.

This property considers three processing steps. During P , the client c emits a signed resource request. During R , rs receives that signed resource request (and expected a signed resource request), and generates $resource$. Then, during Q (which is equal to R for structured ATs), rs sends out the resource response containing $resource$.

In a bit more detail, (1) captures that rs outputs two events during Q , which together with (2) implies that the first of these events is a resource response. In addition, (2) also captures that rs expected the resource request which lead to the response sent in Q to be signed. As for the postconditions, the first few capture the message structures of resource request and response, whereas (VI) says that if there is an honest client c , and a key $clientSignKey$ such that at some point, $clientSignKey$ belonged to c , then – if the resource request was signed with $clientSignKey$ – c must have signed the resource request.

5) Signed Resource Responses

Definition 31 (Non-Repudiation for Signed Resource Responses). Let $\mathcal{F}API$ be a FAPI web system with a network attacker. We say that $\mathcal{F}API$ is secure w.r.t. non-repudiation for signed resource responses iff for every run $\rho = ((S^0, E^0, N^0), \dots)$ of $\mathcal{F}API$, every configuration (S^n, E^n, N^n) in ρ , every client $c \in \mathcal{C}$ which is honest in S^n , every nonce $resource$, and every session id $sessionId \in S^n(c).sessions$ such that

- (1) $S^n(c).sessions[sessionId][expect_signed_resource_res] \equiv \top$, and
- (2) $S^n(c).sessions[sessionId][resource] \equiv resource$,

all of the following hold true:

- (I) There exists a processing step $P = (S, E, N) \xrightarrow{e_{in} \rightarrow c} (S', E', N')$ in ρ with (S, E, N) prior to (S', E', N') where $e_{in} = \langle x, y, m \rangle$, with m having the form $enc_s(\langle HTTPResp, \cdot, status, headers, body \rangle, \cdot)$, where $body \equiv [resource: resource]$, and $S(c) \neq S'(c)$.
- (II) $headers[Signature-Input]$ is a dictionary with at least a key res such that $headers[Signature-Input][res]$ is a sequence with at least two elements. For those first two elements, $components$, and $metadata$, we have $\langle @status, \rangle$, $\langle content-digest, \rangle \in \langle \rangle components$, and $metadata$ is a dictionary with at least the key tag such that $metadata[tag] \equiv fapi-2-response$.
- (III) $headers[Signature]$ is a dictionary with at least a key res such that $headers[Signature][res] \equiv sig(signatureBase, rsSigKey)$.
In addition, $signatureBase$ is of the form $[\langle @status, \rangle: status, \langle content-digest, \rangle: hash(body)] + \langle tag: fapi-2-response, keyid: keyId' \rangle$ for some $keyId'$; however, the dictionaries may contain additional elements.
- (IV) There exists a domain $rsDom \in S^n(c).rsSigKeys$ such that $S^n(c).rsSigKeys[rsDom] \equiv pub(rsSigKey)$.
- (V) If process $rs := dom^{-1}(rsDom)$ is honest in S^n , then there is a processing step $Q = s \xrightarrow{rs \rightarrow E_{out}} s'$, and rs signed the resource response contained in m during Q in Line 6 of Algorithm 20.

C. Security Properties for CIBA

We expect FAPI-CIBA to meet the same security properties as the FAPI 2.0 Security Profile with authorization code flow (i.e., authorization, authentication, and the two session integrity variants).

D. Main Security Theorem

As described in Section II-C, the protocol aims to fulfill authorization, authentication, session integrity for authentication & authorization (for authorization code, and CIBA flows), as well as the five session integrity properties. Thus, our overall security theorem is the conjunction of all those properties as defined above.

Theorem 1. Every FAPI 2.0 web system with network attacker $\mathcal{F}API$ fulfills all of the following properties:

- Authorization (Definition 13),
- Authentication (Definition 15),
- Session integrity for authentication in authorization code flows (Definition 15),
- Session integrity for authentication in CIBA flows (Definition 24),
- Session integrity for authorization in authorization code flows (Definition 25),
- Session integrity for authorization in CIBA flows (Definition 26),

- Non-repudiation for signed authorization requests (Definition 27),
- Non-repudiation for signed authorization responses (Definition 28),
- Non-repudiation for signed introspection responses (Definition 29),
- Non-repudiation for signed resource requests (Definition 30), and
- Non-repudiation for signed resource responses (Definition 31).

We highlight that we prove this theorem for the powerful attacker laid out in the FAPI 2.0 Attacker Model within a faithful formal model of FAPI 2.0 with Dynamic Client Registration, Dynamic Client Management, FAPI-CIBA, and FAPI 2.0 Message Signing. 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,⁸ 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 – note that even though FAPI 2.0 and the other modeled extensions may not use all of these Web features, they might still be (ab)used by an attacker. Thus, our analysis excludes attacks that arise from these features of the Web infrastructure. We prove Theorem 1 in Appendix D.

APPENDIX D PROOFS

A. Helper Lemmas

Lemma 2 (Host of HTTP Request). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ and every process $p \in C \cup AS \cup RS$ that is honest in S it holds true that if the generic HTTPS server calls `PROCESS_HTTPS_REQUEST`(m_{dec}, k, a, f, s) in Algorithm 39, then $m_{dec}.host \in \text{dom}(p)$, for all values of k, a, f and s .

PROOF. `PROCESS_HTTPS_REQUEST` is called only in Line 9 of Algorithm 39. The input message m is an asymmetrically 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

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

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

The first implication holds true due to $S(p).tlskeys \equiv s_0^p.tlskeys \equiv tlskeys^p$, as this sequence is never changed by any honest process $p \in C \cup AS \cup RS$ and due to the definitions of the initial states of clients, authorization servers, and resource servers (Definition 7, Definition 8, Definition 9). ■

Lemma 3 (Generic Server – Correctness of Reference and Request). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every processing step $P = (S^P, E^P, N^P) \rightarrow (S^{P'}, E^{P'}, N^{P'})$ in ρ , every $p \in C \cup AS \cup RS$ being honest in S^P , 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 $p \in C \cup AS \cup RS$ be honest in S^P . p calls the `PROCESS_HTTPS_RESPONSE` function only in the generic HTTPS server algorithm in Line 26 of Algorithm 39. The values *reference* and *request* are taken from $S^P(p).pendingRequests$ in Line 19 of Algorithm 39. Thus, p added these values to `pendingRequests` in a previous processing step $O = (S^O, E^O, N^O) \rightarrow (S^{O'}, E^{O'}, N^{O'})$ by executing Line 15 of Algorithm 39, 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 7, 8, and 9). In O , the process p takes both values from $S^O(p).pendingDNS$ in Line 13 and Line 14 of Algorithm 39. 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 34, 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. ■

⁸For example, FAPI 2.0 excludes code 307 redirects, as they would cause attacks similar to [20].

Lemma 4 (Client’s Signing Keys Do Not Leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAP}I$ with a network attacker, every configuration (S^i, E^i, N^i) in ρ , every client $c \in \mathcal{C}$ that is honest in S^i , every issuer identifier $issuer$, and every process $p \neq c$, we have $\forall j \leq i. issuer \in S^j(c).asAccounts \Rightarrow S^j(c).asAccounts[issuer][sign_key] \notin d_\emptyset(S^i(p))$.

PROOF. We start by proving that an honest client will only store nonces freshly chosen by that client in $asAccounts[issuer][sign_key]$, and that whenever a client updates this value, it completely “forgets” about the “old” value:

There are only two places in which a client stores a value in $asAccounts[issuer][sign_key]$ (during some processing step P): In Line 49 of Algorithm 3, when processing a DCR response, and in Line 23 of Algorithm 3, when processing a DCM response. In both cases, this value is taken from the *reference* parameter as given to PROCESS_HTTPS_RESPONSE (Algorithm 3). Hence, by Lemma 3, there must be a processing step Q prior to P , in which c called HTTPS_SIMPLE_SEND with a corresponding *reference* value. During P , the value of $reference[signKey]$ is then used to set $asAccounts[issuer][sign_key]$. Hence, we need to track where that value comes from.

In the case of Line 49 of Algorithm 3, the *reference* value passed to HTTPS_SIMPLE_SEND during Q must contain a key *responseTo* with value REGISTRATION (see Line 40 of Algorithm 3). Such a *reference* value is only used in a call of HTTPS_SIMPLE_SEND in Line 26 of Algorithm 8. There, the value of $reference[signKey]$ is a fresh nonce $\nu_{cliSignK}$ (see Line 13 of Algorithm 8), which is not stored anywhere else and is only sent out as $pub(\nu_{cliSignK})$ during Q . In addition, the *reference* with $\nu_{cliSignK}$ in it is not used anywhere until P : A client only accesses its *pendingRequests* state subterm in Line 39 of Algorithm 39, and if an entry of *pendingRequests* is used at all, it is immediately removed from the *pendingRequests* subterm (see Line 25 of Algorithm 39), i.e., cannot be read or used again in a later processing step.

In a very similar way, in case of Line 23 of Algorithm 3, the *reference* must contain a key *responseTo* with value CLIENT_MANAGEMENT (see Line 40 of Algorithm 3), which can only happen in Line 56 of Algorithm 9. There, the value for $reference[signKey]$ is once again a fresh nonce (see Line 43 of Algorithm 9), which is also only stored in the aforementioned *reference* and only sent out after applying $pub(\cdot)$. With the same argumentation as above, we conclude that this nonce is not used anywhere until P .

In both cases, the client does not store the “old” value of $asAccounts[issuer][sign_key]$ anywhere before setting/overwriting it (in processing step P).

As we will now show, a client only ever sends out the current value of $asAccounts[issuer][sign_key]$ in one of two ways: 1) wrapped in a $pub(\cdot)$, i.e., as a public key, from which the original value cannot be recovered (see the equational theory in Figure 5), or 2) as a signing key in a signature, where once again, the equational theory does not allow an extraction of the original value.

Specifically, a client c only uses the (current) value of $asAccounts[issuer][sign_key]$ in the following places:

Line 12 of Algorithm 4 The value *clientSignKey* is only used in Lines 22 and 39 of Algorithm 4 to create a term $sig(\cdot, clientSignKey)$, and in Line 37 of Algorithm 4 to create a term $pub(clientSignKey)$.

Line 9 of Algorithm 5 The value *clientSignKey* is only used in Lines 19 and 36 of Algorithm 5 as a signing key, and in Line 34 of Algorithm 5 to create a public key.

Line 18 of Algorithm 6 The value *privKey* is used only in Line 24 of Algorithm 6 to create a public key, and in Line 26 of Algorithm 6 as a signing key.

Line 33 of Algorithm 8 The value *clientSignKey* is used only in Lines 40 and 63 of Algorithm 8 as a signing key.

Note that in all of these places, the value of $asAccounts[issuer][sign_key]$ is only used as signature key or to create a public key, and both constructors do not allow an extraction of the contained key (see the equational theory in Figure 5).

Hence, we conclude that since a client only ever sends out the current value of $asAccounts[issuer][sign_key]$ in a term from which that value cannot be derived, and since the values in $asAccounts[issuer][sign_key]$ are nonces chosen by the client and not used for anything but creating signatures and public keys, i.e., these nonces are not stored or sent out in any other way, no other process can derive a value stored in $asAccounts[issuer][sign_key]$ currently (i.e., in (S^i, E^i, N^i)), or in the past. ■

Lemma 5 (Client’s TLS Key Does Not Leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAP}I$ with a network attacker, every configuration (S, E, N) in ρ , every client $c \in \mathcal{C}$ that is honest in S , every domain $d_c \in \text{dom}(c)$, and every process p with $p \neq c$, all of the following hold true:

- 1) $tlskey(d_c) \notin d_\emptyset(S(p))$
- 2) $\langle d_c, tlskey(d_c) \rangle \in^\diamond S^0(c).tlskeys$
- 3) $\langle d_c, tlskey(d_c) \rangle \in^\diamond S(c).tlskeys$

PROOF. With Definition 7, $\langle d_c, tlskey(c) \rangle \in^\diamond S^0(c).tlskeys$ is equivalent to $\langle d_c, tlskey(d_c) \rangle \in^\diamond tlskeys^c$. This, in turn, follows immediately from the definition of $tlskeys^c$ in Appendix A-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 $\langle d_c, tlskey(d_c) \rangle \in^\diamond S(c).tlskeys$.

The only place in which an honest client accesses any value in its `tlskeys` state subterm is Line 7 of Algorithm 39, where the value is only used to decrypt a message. Hence, the value read from the `tlskeys` state subterm cannot leak.

By definition of `tlskey`, $tlskeys^p$ in Appendix A-C and the initial states of authorization servers (Definition 8), clients (Definition 7), browsers (Definition 3), and resource servers (Definition 9), we have that no other process initially knows $tlskey(d_c)$.

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

Lemma 6 (Client’s mTLS Keys Do Not Leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S^i, E^i, N^i) in ρ , every client $c \in \mathcal{C}$ that is honest in S , every issuer identifier $issuer$, and every process p with $p \neq c$, we have $\forall j \leq i. issuer \in S^j(c).asAccounts \Rightarrow S^j(c).asAccounts[issuer][tls_key] \notin d_\emptyset(S^i(p))$

PROOF. The only places in which an honest client accesses values stored in its state under $asAccounts[issuer][tls_key]$ are:

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

Line 14 of Algorithm 9 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.

Hence, an honest client does not leak any value stored in its state under $asAccounts[issuer][tls_key]$. Note that the above also implies that values, once they are stored under $asAccounts[issuer][tls_key]$, are never “copied” to anywhere else in an honest client’s state.

The only places in which an honest client stores any value in its state under $asAccounts[issuer][tls_key]$ during some processing step P are Line 49 of Algorithm 3 and Line 23 of Algorithm 3. In both cases, the value is taken from the *reference* parameter as given to `PROCESS_HTTPS_RESPONSE` (Algorithm 3). Hence, by Lemma 3, there must be a processing step Q prior to P , in which the client called `HTTPS_SIMPLE_SEND` with a corresponding *reference*. We will now show that in both cases, 1) the value is a nonce freshly generated in Q , and 2) not stored anywhere in the client’s state except in the `pendingRequests` state subterm between Q and P , and $asAccounts[issuer][tls_key]$ after P . In addition, we will show that the value is not sent out in a derivable way during, and between Q and P .

Line 49 of Algorithm 3 Here, the value of $reference[responseTo]$ must be `REGISTRATION` (see Line 40 of Algorithm 3).

Hence, such a *reference* must have been used in a call to `HTTPS_SIMPLE_SEND` during Q (see again Lemma 3). The only place in which such a *reference* is used, is Line 26 of Algorithm 8. There, the value of $reference[tlsKey]$, which is stored to $asAccounts[issuer][tls_key]$ during P , is a fresh nonce $\nu_{cliTlsK}$ (see Line 14 of Algorithm 8). I.e., $\nu_{cliTlsK}$ is not derivable by any process prior to Q . Furthermore, $\nu_{cliTlsK}$ is only used in two places (during processing step Q):

Line 15 of Algorithm 8 Here, $\nu_{cliTlsK}$ is only used to create a public key, from which the original value cannot be derived (see above).

Line 26 of Algorithm 8 Here, $\nu_{cliTlsK}$ is passed to `HTTPS_SIMPLE_SEND` as part of the first argument, i.e., the aforementioned *reference* (see Algorithm 34). This *reference* is stored in the client’s `pendingDNS` state subterm (Line 2 of Algorithm 34) and not used anywhere else. Values stored in the client’s `pendingDNS` state subterm are only accessed in Lines 10ff. of Algorithm 39, where they are removed from `pendingDNS` and stored in another state subterm `pendingRequests`. This subterm, in turn, is only accessed in Lines 19ff. of Algorithm 39, where the value is removed from `pendingRequests` and passed to `PROCESS_HTTPS_RESPONSE`. I.e., any value passed to `HTTPS_SIMPLE_SEND` as part of the first argument, *reference*, including $\nu_{cliTlsK}$, is no longer in the client’s state once *reference* is passed to `PROCESS_HTTPS_RESPONSE`. Hence, we established that the use of $\nu_{cliTlsK}$ in Line 26 of Algorithm 8 (during Q) did not lead to this value being sent out until P , and it is also not being stored in the client’s state outside of $asAccounts[issuer][tls_key]$ after P . Note that the values we tracked through the client’s state are also not used in any message sent by the client between Q and up to and including P .

Line 23 of Algorithm 3 This case is very similar to the previous one, except that the value of $reference[responseTo]$ must be `CLIENT_MANAGEMENT` (see Line 10 of Algorithm 3), and the relevant nonce is generated in Line 44 of Algorithm 9 (instead of Line 14 of Algorithm 8).

So we conclude that $\forall j \leq i. issuer \in S^j(c).asAccounts \Rightarrow S^j(c).asAccounts[issuer][tls_key] \notin d_\emptyset(S^i(p))$. ■

Lemma 7 (Code used in Token Request was received at Redirection Endpoint). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every processing step

$$P = (S, E, N) \xrightarrow[c \rightarrow E_{out}^P]{e_{in}^P \rightarrow c} (S', E', N')$$

in ρ with $c \in \mathcal{C}$ being honest in S , it holds true that if Algorithm 3 (`PROCESS_HTTPS_RESPONSE`) is called in P with *reference* being the second and *request* being the third input argument, and if $reference[responseTo] \equiv \text{TOKEN}$

and $\text{code} \in \langle \rangle \text{ request.body}$, then there is a previous configuration $(S^{L'}, E^{L'}, N^{L'})$ such that $\text{request.body}[\text{code}] \equiv S^{L'}(c).\text{sessions}[\text{reference}[\text{session}]][\text{redirectEpRequest}][\text{message}].\text{parameters}[\text{code}]$.

PROOF. As shown in Lemma 3, there exists a processing step $L = (S^L, E^L, N^L) \rightarrow (S^{L'}, E^{L'}, N^{L'})$ prior to P in which c called `HTTPS_SIMPLE_SEND` with the same reference and request values. The only lines in which a client calls `HTTPS_SIMPLE_SEND` with $\text{reference}[\text{responseTo}] \equiv \text{TOKEN}$ are Line 43 of Algorithm 4 (`SEND_TOKEN_REQUEST`) and Line 40 of Algorithm 5 (`SEND_CIBA_TOKEN_REQUEST`).

The requests send in Line 40 of Algorithm 5 do not contain a code value in their body, see Lines 6, 13, 25, 29, 15, 20 and Line 38 of Algorithm 5, i.e., request was sent in Line 43 of Algorithm 4.

The code included in the request is the input parameter of `SEND_TOKEN_REQUEST` (see Lines 8, 41, and 42 of Algorithm 4). `SEND_TOKEN_REQUEST` is called only in Line 34 of Algorithm 2, i.e., at the redirection endpoint (`/redirect_ep`) of the client, and the code is contained in the parameters of the redirection request that the client stores into $S^{L'}(c).\text{sessions}[\text{sessionId}][\text{redirectionEpRequest}][\text{message}]$ in Line 33 of Algorithm 2, with $\text{sessionId} \equiv \text{reference}[\text{session}]$ (see also Lines 24, 26, and Line 29 of Algorithm 2). ■

Lemma 8 (Authorization Server’s Signing Key Does Not Leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$ with a network attacker, every configuration $Q = (S, E, N)$ in ρ , every authorization server $as \in \text{AS}$ that is honest in S , every term t with $\text{checksig}(t, \text{pub}(\text{signkey}(as))) \equiv \top$, and every process p with $p \neq as$, all of the following hold true:

- $\text{signkey}(as) \notin d_\emptyset(S(p))$
- $\text{signkey}(as) \equiv s_0^{as}.\text{jwt}$
- $\text{signkey}(as) \equiv S(as).\text{jwt}$
- if t is *known* (Definition 83) to p in Q , then t was *created* (Definition 81) by as in a processing step s_e prior to Q in ρ

PROOF. $\text{signkey}(as) \equiv s_0^{as}.\text{jwt}$ immediately follows from Definition 8. $\text{signkey}(as) \equiv S(as).\text{jwt}$ follows from Definition 8 and by induction over the processing steps: state subterm `jwt` of an honest authorization server is never changed.

By Definitions 7, 8, 9, 66, and Appendix A-C, we have that no process (except as) initially knows $\text{signkey}(as)$, i.e., $\text{signkey}(as) \notin d_\emptyset(S^0(p))$.

The only places in which an honest authorization server accesses the `jwt` state subterm are:

Lines 15f. of Algorithm 11 Here, the value of the `jwt` state subterm is only used in a `pub(·)` term constructor as private key from which a public key is derived, i.e., cannot be extracted from the resulting public key (see Figure 5). Thus, it does not matter where that term are stored or sent to.

Lines 97, 200, 212, and 227 of Algorithm 11 Here, the value of the `jwt` state subterm is only used in a `sig(·, ·)` term constructor as signature key, i.e., cannot be extracted from the resulting term (see Figure 5). Thus, it does not matter where that term are stored or sent to.

We conclude that these usages of the `jwt` state subterm do not leak $\text{signkey}(as)$ to any other process, in particular p , and hence, $\text{signkey}(as) \notin d_\emptyset(S(p))$.

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

By Definitions 7, 8, 9, 3, and Appendix A-C, we have that no process (including as) initially knows such a term t , i.e., $t \notin d_\emptyset(S^0(p))$. Together with Definition 57 and Definition 81, 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 76), e must have been *emitted* by some process in a processing step s_e prior to Q' in ρ . Definition 57 and Definition 78 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 9 (mTLS Nonce Created by AS does not Leak). For every run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$ with a network attacker, every configuration $s = (S, E, N)$ in ρ , every authorization server $as \in \text{AS}$ that is honest in S^n , every client $c \in \text{C}$ that is honest in S^n and has been issued client id clientId by as in some processing step $R = s^r \rightarrow s^{r+1}$ with s^r prior to s in ρ , every $i \in \mathbb{N}$ with $0 < i \leq |S(as).\text{mtlsRequests}[\text{clientId}]|$, and every process p with $as \neq p \neq c$ it holds true that $\text{mtlsNonce} := S(as).\text{mtlsRequests}[\text{clientId}].i.1$ is not derivable by p , i.e., $\text{mtlsNonce} \notin d_\emptyset(S^n(p))$.

PROOF.

(A) **Initial state.** Initially, the `mtlsRequests` subterm of the authorization server’s state is empty: $S^0(as).\text{mtlsRequests} \equiv \langle \rangle$ (Definition 8).

- (B) **c 's mTLS key at as is a public key & only c knows the private key.** An authorization server only adds values to the `mtlsRequests` subterm in Line 238 of Algorithm 11, where the mTLS nonce is chosen as a fresh nonce (Line 234 of Algorithm 11). Let $P = (S^j, E^j, N^j) \rightarrow (S^{j+1}, E^{j+1}, N^{j+1})$ be the processing step in which the nonce is chosen (note that (S^j, E^j, N^j) is prior to (S, E, N) in ρ). Note that for an AS to even reach Line 234 of Algorithm 11 during P , we need $clientId \in S^j(as).clients$ (otherwise, the check in Line 232 of Algorithm 11 would fail). Since an honest authorization server never removes entries from its `clients` state subterm, $clientId \in S^j(as).clients$ implies $clientId \in S^l(as).clients$ for all $j \leq l \leq n$. Hence, we can apply Lemma 17, i.e., we have $\exists k_{\text{mtls}} \in \mathcal{N}$ such that $S^l(as).clients[clientId][\text{mtls_key}] \equiv \text{pub}(k_{\text{mtls}})$, and for all processes $p \neq c$, we have $k_{\text{mtls}} \notin d_\emptyset(S^n(p))$.
- (C) **Only c can decrypt mTLS nonce.** During P , the authorization server sends out the fresh mTLS nonce in Line 240 of Algorithm 11, asymmetrically encrypted with the public key $clientKey \equiv S^j(as).clients[clientId][\text{mtls_key}]$ (Line 235 of Algorithm 11). We will refer to this ciphertext $\text{enc}_a(\langle \text{mtlsNonce}, x \rangle, clientKey)$ as $mtlsResp$. From (B), we have $\exists k_{\text{mtls}} \in \mathcal{N}$. $clientKey \equiv \text{pub}(k_{\text{mtls}})$, and for all processes $p \neq c$, we have $k_{\text{mtls}} \notin d_\emptyset(S^n(p))$. Therefore, only c can decrypt $mtlsResp$, or, put more formally, only c can derive $mtlsNonce$ from $mtlsResp$ (see Figure 5).
- (D) **as does not leak mTLS nonce stored in its state.** An authorization server only accesses any values stored in its `mtlsRequests` state subterm in the following places. For each of them, we will prove that no contents of `mtlsRequests` are included in an output event, or stored elsewhere in the authorization server's state. Hence, the authorization server does not leak $mtlsNonce$ from its state (an authorization server might also receive $mtlsNonce$ as part of an input event – this case is covered later).

Line 187 of Algorithm 11 The value taken from the `mtlsRequests` state subterm is only used in the subsequent line to remove a record from the authorization server's `mtlsRequests` state subterm.

Line 19 of Algorithm 12 Here, a single record $mtlsInfo$ is taken from the `mtlsRequests` state subterm, which is used in Line 24 of Algorithm 12 to delete a record from the `mtlsRequests` state subterm, and (possibly) returned as the third element of the return value of Algorithm 12 in Line 30 of Algorithm 12.

Hence, we now have to look at the places where Algorithm 12 is called, and how the third element of its return value is used (which is always stored in a variable $mtlsInfo$ right after Algorithm 12 returns):

Line 116 of Algorithm 11 Here, $mtlsInfo$ is not used at all.

Line 148 of Algorithm 11 Here, $mtlsInfo$ is used only in Line 184 of Algorithm 11, to compare against another value, i.e., $mtlsInfo$ (nor its contents) are not included in any event or stored in the authorization server's state.

Line 242 of Algorithm 11 Here, $mtlsInfo$ is not used at all.

- (E) **c does not leak $mtlsNonce$ upon receiving it.** Recall (C): The encrypted nonce sent out during P can only be decrypted by c . Furthermore, c decrypts such messages only in Line 3 of Algorithm 3 – the only other place where a message is decrypted asymmetrically by c is in the generic HTTPS server (Line 7 of Algorithm 39), where the process would stop due to the requirement that the decrypted message must begin with `HTTPReq`.

We also note that the ciphertext $mtlsResp$ created by the authorization server containing the nonce also contains a public TLS key of as (Lemma 2 and Line 239 of Algorithm 11).

After decrypting the mTLS nonce and public TLS key of as in Line 3 of Algorithm 3, the client stores the sequence $\langle request.host, clientId, pubKey, mtlsNonce \rangle$ into the `mtlsCache` subterm of its state in Line 8 of Algorithm 3, where $clientId, pubKey \in \mathcal{T}_{\mathcal{N}}$ and, in particular,

- $request.host$ is a domain of as (see Line 5 of Algorithm 3).
- $mtlsNonce$ 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 3 without creating an event and without storing $mtlsNonce$ in any other place.

- (F) **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 4 In this algorithm, the client accesses the `mtlsCache` subterm only in Line 17 and Line 30.

In both cases, the sequence containing the nonce is removed from the `mtlsCache` subterm (Lines 19 and 31), 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 43 contains the retrieved mTLS nonces only in the body, under the dictionary key `TLS_AuthN` (Line 18, Line 41) or `TLS_binding` (Line 28, Line 32, Line 41).

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 17, 30).

Case 2: Algorithm 6 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 6). 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 43. The request is sent to the same domain that is stored in the sequence containing the mTLS nonce.

Case 3: Algorithm 8 Here, Line 35 is the last line in which the client accesses the mtlCache state subterm. As in the previous cases, the client removes the corresponding sequence from the mtlCache subterm (Line 37). The client creates an HTTPS request which contains the mTLS nonce in the body under the key TLS_AuthN (Lines 36, 55, and 68). Again, the request is sent to the same domain that is stored in the sequence containing the mTLS nonce (see Line 68).

In all cases, the HTTPS request is sent to the domain stored in the first entry of the sequence containing the mTLS nonce (stored in the mtlCache subterm). Let $req_{c \rightarrow as}$ be the HTTP request that the client sends by calling HTTPS_SIMPLE_SEND.

HTTPS_SIMPLE_SEND stores the request $req_{c \rightarrow as}$ (which contains the mTLS nonce) in the pendingDNS state subterm of c , see Line 2 of Algorithm 34 and then stops with the DNS request (which does not contain the nonce) in Line 3 of Algorithm 34. Thus, after finishing this processing step, the client stores the mTLS nonce only in its pendingDNS state subterm.

The client accesses the pendingDNS state subterm only within the else case in Line 10 of Algorithm 39, 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 39, or creates a new pendingRequests entry containing the request $req_{c \rightarrow as}$ (and thus, also the mTLS nonce) in Line 15 of Algorithm 39. In this case, the client removes the request from the pendingDNS state subterm in Line 17 of Algorithm 39, i.e., regarding the client state, the mTLS nonce is now only contained in the newly created pendingRequests entry. The client finishes the processing step by encrypting $req_{c \rightarrow as}$ 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 39, and (E).

(G) as **does not leak mTLS nonce contained in request.**

As the client encrypts $req_{c \rightarrow as}$ asymmetrically with a key of as , it follows that only as can decrypt the HTTPS request (Lemma 46).

The authorization server only decrypts terms in the generic HTTPS server algorithms. More specifically, this request is decrypted (only) in Line 7 of Algorithm 39, as this is the only place where an authorization server decrypts a message asymmetrically, and then used as a function argument for PROCESS_HTTPS_REQUEST which is modeled in Algorithm 11. In Algorithm 11, none of the endpoints except for the PAR (Line 103) and token endpoints (Line 145) 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 12), which removes an entry from the mtlRequests 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 180–190), 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.

(H) c **does not leak mTLS nonce in request after getting the response.** When the client receives the HTTPS response to $req_{c \rightarrow as}$, 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 (see Lines 19ff. of Algorithm 39). Algorithm 3 (PROCESS_HTTPS_RESPONSE) does not store a nonce contained in the body of the request, i.e., the third argument, 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 same authorization server (i.e., only that authorization server can decrypt the client's message). As an honest authorization server never sends out such a nonce received in a request, and neither the client or authorization server leak the mTLS nonce as stored in their states in between, we conclude that the nonce never leaks to any other process, in particular not to p . ■

Lemma 10 (Resource Server's Signing Key Does Not Leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration $Q = (S, E, N)$ in ρ , every resource server $rs \in RS$ that is honest in S , every term t with $\text{checksig}(t, \text{pub}(\text{signkey}(rs))) \equiv \top$, and every process p with $p \neq rs$, all of the following hold true:

- $\text{signkey}(rs) \notin d_\emptyset(S(p))$
- $\text{signkey}(rs) \equiv s_0^{rs}.\text{jwt}$
- $\text{signkey}(rs) \equiv S(rs).\text{jwt}$
- if t is known (Definition 83) to p in Q , then t was created (Definition 81) by rs in a processing step s_e prior to Q in ρ

PROOF. $\text{signkey}(rs) \equiv s_0^{rs}.\text{jwt}$ immediately follows from Definition 9. $\text{signkey}(rs) \equiv S(rs).\text{jwt}$ follows from Definition 9 and by induction over the processing steps: state subterm jwt of an honest resource server is never changed.

By Definitions 7, 8, 9, 66, and Appendix A-C, we have that no process (except rs) initially knows $\text{signkey}(rs)$, i.e., $\text{signkey}(rs) \notin d_\emptyset(S^0(p))$.

The only place in which an honest resource server accesses the `jwt` state subterm is Line 6 of Algorithm 20. There, the value of the `jwt` state subterm is only used in a `sig(·, ·)` term constructor as signature key, i.e., cannot be extracted from the resulting term (see Figure 5). Thus, it does not matter where that term are stored or sent to. We conclude that this usage of the `jwt` state subterm does not leak `signkey(rs)` to any other process, in particular p , and hence, `signkey(rs)` $\notin d_0(S(p))$.

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

By Definitions 7, 8, 9, 3, and Appendix A-C, we have that no process (including rs) initially knows such a term t , i.e., $t \notin d_0(S^0(p))$. Together with Definition 57 and Definition 81, 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 76), e must have been emitted by some process in a processing step s_e prior to Q' in ρ . Definition 57 and Definition 78 imply that p (or any other process $\neq rs$) cannot have emitted e in s_e (i.e., cannot have created t in s_e).

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

Lemma 11 (mTLS Nonce created by RS does not Leak). For every run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every resource server $rs \in RS$ that is honest in S^n , every client $c \in C$ that is honest in S^n , every nonce $k_{\text{mTLS}} \in \mathcal{N}$, every $i \in \mathbb{N}$ with $0 \leq i \leq |S(rs).\text{mTLSRequests}|$ and with $S(rs).\text{mTLSRequests}.i.2 \equiv \text{pub}(k_{\text{mTLS}})$, every process p_1 with $p_1 \neq c$, and every process p_2 with $rs \neq p_2 \neq c$ it holds true that if $k_{\text{mTLS}} \notin d_0(S^n(p_1))$, then $\text{mTLSNonce} := S(rs).\text{mTLSRequests}.i.1$ does not leak to p_2 , i.e., $\text{mTLSNonce} \notin d_0(S^n(p_2))$.

PROOF. This proof is similar to the proof of Lemma 9:

Initially, the `mTLSRequests` subterm of the resource server’s state is empty, i.e., $S^0(rs).\text{mTLSRequests} \equiv \langle \rangle$ (Definition 9). A resource server only adds values to the `mTLSRequests` subterm in Line 5 of Algorithm 18, where the mTLS nonce (the first value of the sequence that is added to `mTLSRequests`) is a fresh nonce (Line 3 of Algorithm 18).

Let $(S^i, E^i, N^i) \rightarrow (S^{i'}, E^{i'}, N^{i'})$ be the processing step in which the nonce is chosen (note that (S^i, E^i, N^i) is prior to (S, E, N) in ρ). In the same processing step, the resource server sends out the nonce in Line 7 of Algorithm 18, asymmetrically encrypted with the public key `pub(kmTLS)` (precondition of the lemma, see also Line 5 and Line 6 of Algorithm 18; note that the RS never modifies the values stored in `mTLSRequests`, it only deletes entries in Line 27 of Algorithm 18). The `mTLSNonce` saved in `mTLSRequests` is not sent in any other place.

The encrypted nonce can only be decrypted by c , as only c can derive the private key k_{mTLS} (precondition of the lemma). c decrypts messages only in Line 3 of Algorithm 3. (The only other place where a message is decrypted asymmetrically by c is in the generic HTTPS server (Line 7 of Algorithm 39), 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 2).

After decrypting the mTLS nonce and public TLS key of rs in Line 3 of Algorithm 3, the client stores the sequence $\langle \text{request.host}, \text{clientId}, \text{pubKey}, \text{mTLSNonce} \rangle$ into the `mTLSCache` subterm of its state (Line 8 of Algorithm 3), where $\text{clientId}, \text{pubKey} \in \mathcal{T}_{\mathcal{N}}$ and, in particular,

- `request.host` is a domain of rs (see Line 5, Algorithm 3)
- `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 3 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 4

In this algorithm, the client accesses the `mTLSCache` subterm only in Line 17 and Line 30.

In both cases, the sequence containing the nonce is removed from the `mTLSCache` subterm (Lines 19 and 31), 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 43 contains the retrieved mTLS nonces only in the body, under the dictionary key `TLS_AuthN` (Line 18, Line 41) or `TLS_binding` (Line 28, Line 32, Line 41).

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 17, 30).

Note that messages created by Algorithm 4 do not contain an Authorization header.

Case 2: Algorithm 5 This case is similar to the previous case.

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 40 contains the retrieved mTLS nonces only in the body, under the dictionary key TLS_AuthN (Line 15, Line 38) or TLS_binding (Line 25, Line 29, Line 38).

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).

Note that messages created by Algorithm 5 do not contain an Authorization header.

Case 3: Algorithm 6

Here, the client accesses the mtlsCache state subterm only in Line 13, and removes the sequence with the mTLS nonce from the mtlsCache subterm (Line 16 of Algorithm 6). 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 43. The request is sent to the same domain that is stored in the sequence containing the mTLS nonce.

The client might sign this request (Lines 30-40 of Algorithm 6). Regarding the mTLS nonce, the client stores the hash of the body in the Content-Digest header (Line 32 of Algorithm 6). The signature stored in the Signature header covers the Content-Digest header, see Lines 32, 34, and 39 of Algorithm 6.

Case 4: Algorithm 8

Here, Line 35 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 37).

The client creates the term *requestData*, which contains the mTLS nonce in the body under the key TLS_AuthN (Lines 36, 55), and creates an HTTP request in Line 68 of Algorithm 8, with the body set to *requestData* (Line 65 and Line 67 of Algorithm 8), or set to the signed *requestData* value (the client might add more values to *requestData* in Line 59 and Line 62 of Algorithm 8).

Again, the request is sent to the same domain that is stored in the sequence containing the mTLS nonce (see Line 68).

Note that messages created by Algorithm 8 do not contain an Authorization header.

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 $req_{c \rightarrow rs}$ be the request that the client sends by calling HTTPS_SIMPLE_SEND.

HTTPS_SIMPLE_SEND stores the request $req_{c \rightarrow rs}$ (which contains the mTLS nonce) in the pendingDNS state subterm of *c*, see Line 2 of Algorithm 34, and then stops with the DNS request (which does not contain the nonce) in Line 3 of Algorithm 34. Thus, after finishing this processing step, the client stores the mTLS nonce only in its pendingDNS state subterm.

The client accesses the pendingDNS state subterm only within the else case in Line 10 of Algorithm 39, 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 39, or creates a new pendingRequests entry containing the request $req_{c \rightarrow rs}$ (and thus, also the mTLS nonce) in Line 15 of Algorithm 39. In this case, the client removes the request from the pendingDNS state subterm in Line 17 of Algorithm 39, 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 $req_{c \rightarrow rs}$ 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 39.

***rs* does not leak mTLS nonce contained in request.** As the HTTP request $req_{c \rightarrow rs}$ is encrypted asymmetrically 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 39, as this is the only place where a resource server decrypts a message asymmetrically, and then used as a function argument for PROCESS_HTTPS_REQUEST which is modeled in Algorithm 18.

In Algorithm 18, the /MTLS-prepare and /DPoP-nonce endpoints (Line 2 and Line 8 of Algorithm 18) do not read, store, or send out a value stored in the body of the request under the TLS_AuthN or TLS_binding key or within a signature (these endpoints do not call the extractmsg() function).

For the last endpoint starting at Line 13 of Algorithm 18, we now consider all possible mTLS nonces in $req_{c \rightarrow rs}$:

- $req_{c \rightarrow rs}.body[TLS_AuthN]$ (created in Algorithm 4, 5, or 8): Requests created by these algorithms do not contain an Authorization header (see above), thus, the RS would stop in Line 20 of Algorithm 18 without changing its state and without emitting messages.
- $req_{c \rightarrow rs}.body[TLS_binding]$ (created in Algorithm 4, 5, or 6): The RS accesses values stored in the body of the request under the TLS_binding key only in Line 25 of Algorithm 18. We distinguish the following cases:
 - Opaque access token: If Line 50 of Algorithm 18 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.
 - Structured access token: If Lines 62ff. of Algorithm 18 are executed, then the RS responds in the same processing step. The RS does not use the TLS_binding value, and uses the request $req_{c \rightarrow rs}$ (containing the nonce) only in Line 68 of

Algorithm 18, where it calls the VERIFY_REQUEST_SIGNATURE function (Algorithm 21), which returns a boolean value (without modifying the state of the RS or emitting messages).

- $req_{c \rightarrow rs}$.headers (created in Algorithm 6): The Content-Digest and Signature headers might contain the mTLS nonce. However, the Content-Digest header contains only the hashed request body, and the signature in the Signature headers covers the Content-Digest header. The RS leaks the headers (Line 82 of Algorithm 18 and Line 43 of Algorithm 19), but the original mTLS nonce value cannot be derived from the hash values.
- $req_{c \rightarrow rs}$.body (if the body is the signature created in Algorithm 8): As in the first case, the request does not contain an Authorization header, thus, the RS would stop in Line 20 of Algorithm 18 without changing its state and without emitting messages.

c does not leak mTLS nonce in request after getting the response. When receiving the HTTPS response to $req_{c \rightarrow rs}$, 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 3 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 12 (JWS client assertion created by client does not leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system \mathcal{FAP} with a network attacker, every configuration (S^i, E^i, N^i) in ρ , every authorization server $as \in AS$ that is honest in S^i , every client $c \in C$ that is honest in S^i and has been issued client identifier $clientId$ by as (in some processing step $s \rightarrow s'$ with s prior to (S^i, E^i, N^i) in ρ), every domain $issuer \in \text{dom}(as)$, every index $j \leq i$, every term $clientSignKey := S^j(c).asAccounts[issuer][sign_key]$, every term t with

- $\text{checksig}(t, \text{pub}(clientSignKey)) \equiv \top$,
- $\text{extractmsg}(t)[iss] \equiv clientId$,
- $\text{extractmsg}(t)[sub] \equiv clientId$, and
- $\text{extractmsg}(t)[aud].host \in \text{dom}(as)$ or $\text{extractmsg}(t)[aud] \in \text{dom}(as)$

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

PROOF. We can immediately apply Lemma 4, which gives us $clientSignKey \notin d_\emptyset(S^i(p))$ for all processes $p \neq c$.

Thus, only c can derive a term t such that $\text{checksig}(t, \text{pub}(clientSignKey)) \equiv \top$ (see Figure 5). In other words, for t to be known to any process (including c and as), c must have signed a dictionary with the corresponding *iss*, *sub*, and *aud* values.

An honest client signs dictionaries with both an *aud*, and an *iss* dictionary key only in the following locations:

Line 22 of Algorithm 4 The signature created in Line 22 of Algorithm 4 is added to the body of an HTTP request (Lines 23, 41, and 42 of Algorithm 4). 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 $\text{extractmsg}(t)[aud]$ (i.e., *selectedAS* in the context of Algorithm 4). From Lemma 21, 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 43, using *responseTo*: TOKEN in the first function argument). This party is as by the preconditions of this lemma, i.e., only as can decrypt the corresponding ciphertext and extract t .

Line 19 of Algorithm 5 This case is very similar to the first one, except for differing line numbers; the signature is added to an HTTP request (Lines 20, 38, and 39 of Algorithm 5), which is then passed to HTTPS_SIMPLE_SEND, and hence encrypted for as .

Line 40 of Algorithm 8 As in the first case, the signature created in Line 40 of Algorithm 8 is added to the body of an HTTP request (Lines 41, 55, and 68 of Algorithm 8). 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 $\text{extractmsg}(t)[aud]$ belongs. Analogous to the first case, we can apply Lemma 21 to conclude that this party must be an honest AS (and the request is stored by c with *responseTo*: PAR by HTTPS_SIMPLE_SEND).

On the client side, this leaves t being stored in the *pendingRequests* state subterm, which is only accessed when processing HTTPS responses. When the client receives such an HTTPS response, the generic HTTPS server decrypts the message and calls PROCESS_HTTPS_RESPONSE (Lines 19ff. of Algorithm 39). The original request (containing t) is used as the third function argument in that call. However, the instantiation of PROCESS_HTTPS_RESPONSE for clients (Algorithm 3) does not access the body of the original request when processing TOKEN or PAR responses and hence cannot leak t in any way.

This leaves us with as , which can decrypt the aforementioned requests containing t : when processing an HTTPS request in Algorithm 11, the authorization server does not store the client assertion and does not create a network message containing the client assertion: the signatures created in Line 22 of Algorithm 4, Line 19 of Algorithm 5, and Line 40 of Algorithm 8 are contained in the request under a key *client_assertion*, which an AS only accesses in Line 3 of Algorithm 12, where the

value is only used to verify the signature (Line 4 of Algorithm 12), and to extract the signed term (Line 11 of Algorithm 12). Note that in all three cases from above, the path element of the generated HTTP request is either `/par` or `/token` and during processing of requests to those endpoints, an AS also does not store the whole request or request body (see Algorithm 11). In other words, when *as* processes a request containing *t*, it does neither leak *t*, nor does it store *t* in its state (and hence, also cannot leak *t* at a later time).

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

Lemma 13 (Client Authentication). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every authorization server $as \in AS$, every client $c \in C$, every processing step Q in ρ

$$Q = (S, E, N) \xrightarrow[as \rightarrow E_{out}]{e_{in} \rightarrow as} (S', E', N')$$

with *c* and *as* being honest in S' and every client identifier *clientId* issued to *c* by *as* (during some processing step $s^{cid} \rightarrow s^{cid'}$), if $e_{in} \equiv \langle x, y, \text{enc}_a(\langle m, k \rangle, k') \rangle$ (for some x, y, k, k') such that

- $m \in \text{HTTPRequests}$ and
- $\text{client_id} \in m.\text{body} \Rightarrow m.\text{body}[\text{client_id}] \equiv \text{clientId}$ and
- $\text{client_id} \in \text{extractmsg}(m.\text{body}) \Rightarrow \text{extractmsg}(m.\text{body})[\text{client_id}] \equiv \text{clientId}$ and
- $\text{client_assertion} \in m.\text{body} \Rightarrow \text{extractmsg}(m.\text{body}[\text{client_assertion}])[\text{iss}] \equiv \text{clientId}$ and
- $\text{client_assertion} \in \text{extractmsg}(m.\text{body}) \Rightarrow \text{extractmsg}(\text{extractmsg}(m.\text{body})[\text{client_assertion}])[\text{iss}] \equiv \text{clientId}$ and
- $m.\text{path} \equiv \text{/par} \vee m.\text{path} \equiv \text{/token} \vee m.\text{path} \equiv \text{/backchannel-authn}$ and
- E_{out} is not empty,

then *c* created *m* (Definition 81).

PROOF. Since *m* may nor may not contain a signed body, and we sometimes need to refer to the body without a possible signature, we define

$$m' := \begin{cases} \langle \text{HTTPReq}, m.\text{nonce}, m.\text{method}, m.\text{host}, m.\text{path}, m.\text{parameters}, m.\text{headers}, \text{extractmsg}(m.\text{body}) \rangle & \text{if } m.\text{body} \\ & \sim \text{sig}(*, *) \\ m & \text{otherwise} \end{cases}$$

(A) *as* **does not create** *m*. An authorization server only emits HTTP(S) requests in two places:

Line 304 of Algorithm 11 The request body in this case is a dictionary with only one key, `auth_req_id`; i.e., it contains neither a key `TLS_AuthN`, nor a key `client_assertion`. We will come back to this later.

Line 13 of Algorithm 15 In this case, the path component of the emitted request is `/start-ciba-authentication`, i.e., neither `/par`, nor `/token`, nor `/backchannel-authn`.

(B) *as* **executes** `PROCESS_HTTPS_REQUEST` **during** Q . Processing of e_{in} during Q begins with Algorithm 39. Since we have E_{out} not empty, Q cannot finish at one of the parameterless **stops** in Algorithm 39. We also have $m \neq \text{CORRUPT}$ and *as* is honest, i.e., $S(as).\text{corrupt} \equiv \perp$, and therefore, Q does not stop in Line 6 of Algorithm 39.

The **stop** in Line 18 of Algorithm 39 cannot be reached, since $m \notin \text{DNSResponses}$ (see Appendix E-B5).

All other **stops** within Algorithm 39 are parameterless, hence, execution during Q must reach one of the function calls in Algorithm 39:

Line 9 of Algorithm 39 (`PROCESS_HTTPS_REQUEST`) As the third element within e_{in} has the correct structure, this function call can be reached.

Line 24 of Algorithm 39 (`PROCESS_OTHER`) The instantiation of `PROCESS_OTHER` for authorization servers (Algorithm 14) does not output any events, which contradicts precondition E_{out} not empty.

Line 26 of Algorithm 39 (`PROCESS_HTTPS_RESPONSE`) Since $m \notin \text{HTTPResponses}$ (Definition 48), this function call cannot be reached (due to the check in Line 23 of Algorithm 39).

Line 28 of Algorithm 39 (`PROCESS_TRIGGER`) Since $m \neq \text{TRIGGER}$, this function call cannot be reached.

Line 30 of Algorithm 39 (`PROCESS_OTHER`) The instantiation of `PROCESS_OTHER` for authorization servers (Algorithm 14) does not output any events, which contradicts precondition E_{out} not empty.

We conclude that *as* must execute `PROCESS_HTTPS_REQUEST` during Q .

(C) *as* **executes** **Line 144 or Line 215 or Line 266 of Algorithm 11** **during** Q . When processing e_{in} during Q , the generic HTTPS server calls `PROCESS_HTTPS_REQUEST`, i.e., Algorithm 11, in Line 9 of Algorithm 39 (see (B)).

If $m.\text{path} \equiv \text{/par}$ (with *m* from this lemma's preconditions), then the `PAR` endpoint starting in Line 103 of Algorithm 11 is executed. No **stop** within that endpoint except for the last (unconditional) **stop** in Line 144 of Algorithm 11 emits an event.

Analogously, if $m.path \equiv /token$, then the token endpoint starting in Line 145 of Algorithm 11 is executed and the (unconditional) **stop** in Line 215 of Algorithm 11 was reached, as no other **stop** within the token endpoint emits events. If $m.path \equiv /backchannel-authn$, then the backchannel authentication endpoint starting in Line 241 of Algorithm 11 is executed and the (unconditional) **stop** in Line 266 of Algorithm 11 was reached, as no other **stop** within this endpoint emits events.

- (D) **HTTP request contains values that only c and as know.** The precondition E_{out} not empty implies that the checks done in the AUTHENTICATE_CLIENT function (Algorithm 12), called in Line 116 of Algorithm 11 (PAR endpoint), or Line 148 of Algorithm 11 (token endpoint), or Line 242 of Algorithm 11 (backchannel authentication endpoint) did not lead to a **stop**.

In the case of the PAR endpoint, if as expects a signed PAR (Line 105 of Algorithm 11), the PAR signature is removed from $m.body$ (Line 106 of Algorithm 11), resulting in m' . Note that if Algorithm 12 is called with a signed PAR where the signature has not been removed, that algorithm stops without emitting any events in Line 26 of Algorithm 12 – hence, this cannot be the case in Q .

So in all three endpoints, Algorithm 12 is called with the HTTP request m' and $S(as)$ as input arguments. As Line 26 of Algorithm 12 is not executed (because E_{out} is not empty), it follows that $client_assertion \in m'.body$ or $TLS_AuthN \in m'.body$.

Case 1: $client_assertion \in m'.body$. As $extractmsg(m'.body[client_assertion])[iss] \equiv clientId$ (lemma precondition), and the check in Line 12 of Algorithm 12 succeeds (otherwise, E_{out} would be empty), the verification key used in Line 4 of Algorithm 12 during Q must have been $S(as).clients[clientId][jwt_key]$.

By applying Lemma 17, we get $\exists k_{jwt} \in \mathcal{K}$ such that $S(as).clients[clientId][jwt_key] \equiv pub(k_{jwt})$, and $k_{jwt} \notin d_\emptyset(S^m(p))$ for any process $p \neq c$. Hence, we have $m.body[client_assertion] \sim sig(*, pub(k_{jwt}))$.

A term $t \sim sig(*, pub(k_{jwt}))$ is not part of any processes' initial state (Definition 8, Definition 7, Definition 3, Definition 9). This, together with Figure 5, gives us that if any process can derive t in S – which is true for as – then t must originate from c (see also proof of Lemma 12). As shown in the proof of Lemma 12, a client only creates signed terms with aud and iss keys in the signed value in a few locations; and in each of those, the key used to sign such a term is taken from the client's $asAccounts$ state subterm, under some issuer, under key $sign_key$.

Now, let $cli_assertion := extractmsg(m'.body[client_assertion])$. Since the check in Line 12 of Algorithm 12 did not result in a parameterless **stop**, we have $cli_assertion[iss] \equiv clientId$, and $cli_assertion[sub] \equiv clientId$. Furthermore, $cli_assertion[aud].host \in dom(as)$ or $cli_assertion[aud] \in dom(as)$ (Line 14 of Algorithm 12 and the host of the request is a domain of the authorization server as shown in Lemma 2).

With this, we can apply Lemma 12.

Thus, for all processes p such that $as \neq p \neq c$, it holds true that $m'.body[client_assertion] \notin d_\emptyset(S'(p))$, i.e., only c and as can derive $m'.body[client_assertion]$. As authorization servers do not create HTTP(S) requests with a key $client_assertion$ (see (A)), it follows that m' – and hence m – was created by c .

Case 2: $TLS_AuthN \in m.body$. From Lines 17–19 of Algorithm 12 it follows that

$$\exists i \in \mathbb{N}. S(as).mtlsRequests[m.body[client_id]].i.1 \equiv m.body[TLS_AuthN]$$

Note that $client_id \in m.body$ as otherwise, the **stop** in Line 23 of Algorithm 12 will be executed.

Now, we can apply Lemma 9 with ρ' (ρ' being the trace prefix of ρ up to and including (S', E', N')).

Thus, for all processes p such that $as \neq p \neq c$, it holds true that $m'.body[TLS_AuthN] \notin d_\emptyset(S'(p))$, i.e., only c and as can derive $m'.body[TLS_AuthN]$. As authorization servers do not create HTTP(S) requests with a key TLS_AuthN in the request body (see (A)), we conclude that m' – and thus m – was created by c . ■

Lemma 14 (DPoP proof secrecy (RS)). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every resource server $rs \in RS$ that is honest in S , every client $c \in C$ that is honest in S , every nonce $signKey \in \mathcal{K}$, every process $p_1 \neq c$, every process p_2 with $rs \neq p_2 \neq c$, and every term t with

- $checksig(t, pub(signKey)) \equiv \top$
- $extractmsg(t)[payload][htu].host \in dom(rs)$,
- $ath \in {}^\diamond extractmsg(t)[payload]$,
- $extractmsg(t)[payload][nonce] \in S(rs).dpopNonces$

it holds true that if $signKey \notin d_\emptyset(S^m(p_1))$, then $t \notin d_\emptyset(S(p_2))$.

PROOF. As only c can derive the key $signKey$, it follows that only c can create such 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 three locations:

- In Line 39 of Algorithm 4, where the payload dictionary does not contain an `ath` value (see Line 38 of Algorithm 4)
- In Line 36 of Algorithm 5, where the payload dictionary does not contain an `ath` value (see Line 35 of Algorithm 5)
- In Line 26 of Algorithm 6.

The client sends the term t created in Line 26 of Algorithm 6 to `extractmsg(t)[payload][htu].host` via `HTTPS_SIMPLE_SEND` (using `responseTo: RESOURCE_USAGE` in the first function argument), see Lines 21, 25, 42, and 43 of Algorithm 6. 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 6, and potentially as part of the `Signature` header, see Line 37 and Line 39 of Algorithm 6. The client also adds an `Authorization` header containing a dictionary with a DPoP dictionary key, see Lines 27 and 42 of Algorithm 6.

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 `extractmsg(t)[payload][htu].host`, i.e., for rs , which can only be decrypted by rs (Lemma 46).

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 3) does not access the headers of the request when processing `RESOURCE_USAGE` responses.

When processing the HTTPS request created by the client in Algorithm 18, the resource server does not access the request headers (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 18). For all other path values (Line 13 of Algorithm 18), 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 18 of Algorithm 18). Otherwise, the resource server will eventually invalidate the nonce value stored in the DPoP proof in Line 44 of Algorithm 18 (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 20 and 30 of Algorithm 18). The **stops** 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 18. Thus, the nonce is not contained in `dpopNonces` after Line 44 of Algorithm 18 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 `extractmsg(t)[payload][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 15 (Registration Access Tokens Stored at AS Never Change). For

- every run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of $\mathcal{F}API$ with a network attacker,
- every authorization server $as \in AS$ that is honest in S^n ,
- every client $c \in C$ that is honest in S^n ,
- every client identifier $clientId \in \mathcal{T}_{\mathcal{N}}$ that has been issued to c by as in some processing step $R = (S^r, E^r, N^r) \rightarrow (S^{r'}, E^{r'}, N^{r'})$ in ρ (according to Definition 11),

it holds true that $S^n(as).clients[clientId][reg_at] \equiv S^{r'}(as).clients[clientId][reg_at]$.

PROOF. An honest AS modifies its `clients` state subterm only in Line 51 and Line 57 of Algorithm 11 (i.e., the `/manage` endpoint) and Line 24 of Algorithm 13 (the `/reg` endpoint). Let $P = (S, E, N) \rightarrow (S', E', N')$ be a processing step after R in which the AS modifies its `clients` state subterm. We show that the AS never modifies the `reg_at` value of the corresponding `clientId` dictionary, i.e., $S(as).clients[clientId][reg_at] \equiv S'(as).clients[clientId][reg_at]$.

Case 1: Line 51 of Algorithm 11. In this case, the AS stores the value `clientInfo`, which is equal to $S(as).clients[clientId]$ (see Line 24 of Algorithm 11) with some modified values (Lines 34-37, Line 41, Line 43, and Line 50 of Algorithm 11), however, without changing the `reg_at` value.

Case 2: Line 57 of Algorithm 11. Here, the AS only modifies the `active` entry of the dictionary (Line 57 of Algorithm 11). All other values, in particular, the `reg_at` value, stay the same.

Case 3: Line 24 of Algorithm 13. In this case, the AS does not change an existing client entry: Let $clientId'$ be the key of the entry modified by the AS in Line 24 of Algorithm 13. $clientId'$ is taken from $S(as).pendingClientIds$ in Line 2 of Algorithm 13. However, as $clientId \in S(as).clients$, it follows that $clientId \notin S(as).pendingClientIds$. ■

Lemma 16 (Secrecy of Registration Access Tokens Stored at AS). For

- every run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of $\mathcal{F}API$ with a network attacker,
 - every authorization server $as \in AS$ that is honest in S^n ,
 - every client $c \in C$ that is honest in S^n ,
 - every client identifier $clientId \in \mathcal{T}_{\mathcal{N}}$ that has been issued to c by as in some processing step $R = (S^r, E^r, N^r) \rightarrow (S^{r'}, E^{r'}, N^{r'})$ in ρ (according to Definition 11),
- it holds true that $S^{r'}(as).clients[clientId][reg_at] \notin d_{\emptyset}(S^n(attack))$.

PROOF. Note that an honest AS modifies its `clients` state subterm only in Line 51 and Line 57 of Algorithm 11 (i.e., the `/manage` endpoint), and Line 24 of Algorithm 13 (the `/reg` endpoint).

Creating the Registration Access Token: Initially, the `clients` state subterm of as is empty (Definition 8). As the AS issues $clientId$ in R , it follows that for all configurations (S', E', N') prior to $(S^{r'}, E^{r'}, N^{r'})$, $clientId \notin S'(as).clients$, as the AS takes $clientId$ from $S^r(as).pendingClientIds$ (Line 2 of Algorithm 13). Initially, `pendingClientIds` is empty (Definition 8), and the AS adds values to this state subterm only in Line 5 of Algorithm 14, after ensuring that the value that is being stored is not part of the `clients` and `pendingClientIds` state subterms (Line 3 of Algorithm 14). In R , the AS executes Line 26 of Algorithm 13 (Lemma 1), thus, it must have executed Line 24 of Algorithm 13 (i.e., the `/reg` endpoint, as Algorithm 13 is only called in Line 19 of Algorithm 11).

Let $\langle x, y, enc_a(\langle regReq, k \rangle, pk_{as}) \rangle$ be the input event that as processes in R . The request $enc_a(\langle regReq, k \rangle, pk_{as})$ was created by c (see Definition 11).

In Line 24 of Algorithm 13, the AS creates the entry for $clientId$. This dictionary entry contains the key `reg_at` (see Line 14 of Algorithm 13) with the value being a fresh nonce (see Line 9 of Algorithm 13). In addition to storing the registration access token into its state, the AS includes the value into the response (see Line 13 and Line 25 of Algorithm 13). Note that this response also contains a URL `reg_client_uri` with the domain being the host of the registration request and the path `/manage` (Lines 10 and 13 of Algorithm 13).

Thus, in the configuration $(S^{r'}, E^{r'}, N^{r'})$, the registration access token is only stored in $S^{r'}(as).clients[clientId]$ and only contained in the response to c .

Processing the Registration Response: The registration response is an HTTPS response encrypted with k , and only c can decrypt it. The client created the registration request in Line 26 of Algorithm 8, as this is the only place where a client creates POST requests containing `jwt`s in the body. Let $T = (S^t, E^t, N^t) \rightarrow (S^{t'}, E^{t'}, N^{t'})$ be the processing step in which the client processes the response. When calling the `HTTPS_SIMPLE_SEND` function, the client uses a reference value `reference` with `reference[responseTo] ≡ REGISTRATION`, i.e., the client will process the registration response in Line 40 of Algorithm 3. Let `selectedAS` $\equiv S^t.sessions[reference[session]][selected_AS]$. This is the same value as the client selects in Line 5 of Algorithm 8 when sending the registration request (Lemma 20). The client sends the registration request to the domain `s.oauthConfigCache[selectedAS][reg_ep].host`, with s being the state of the corresponding configuration (see Lines 4, 5, 10, 12, and Line 25 of Algorithm 8). As shown in Lemma 21, this is equal to `selectedAS`. As the AS processes this request, it follows that `selectedAS` $\in \text{dom}(as)$ (Lemma 2). The client stores the registration access token in $S^{t'}.asAccounts[selectedAS][reg_at]$ (see Line 46 and Line 49 of Algorithm 3), and then continues with `PREPARE_AND_SEND_INITIAL_REQUEST` (Algorithm 8, called in Line 57 of Algorithm 3). There, the client does not access this registration access token, i.e., in T , the client just stores the registration access token in $S^{t'}.asAccounts[selectedAS][reg_at]$ without emitting an event containing it.

In addition to storing the registration access token, the client stores the `reg_client_uri` value contained in the response, i.e., $S^{t'}.asAccounts[selectedAS][reg_client_uri].host \equiv selectedAS$ and $S^{t'}.asAccounts[selectedAS][reg_client_uri].path \equiv /manage$ (see Lines 45 and 49 of Algorithm 3).

Registration Access Token Stored at AS: The AS accesses the `reg_at` entry of a client dictionary only in three locations:

- At the `/manage` endpoint in Line 20 of Algorithm 11, where the AS expects an HTTP request containing the registration access token in its Authorization header. However, up to $(S^{t'}, E^{t'}, N^{t'})$, the client did not send a request containing this value, and the authorization server does not send PUT requests.
- At the `/manage` endpoint in Line 54 of Algorithm 11, where the AS expects an HTTP request containing the registration access token in its Authorization header. As in the previous case, such a request cannot exist up to $(S^{t'}, E^{t'}, N^{t'})$.
- In `REGISTER_CLIENT` (Algorithm 13), where the client sets this value to a fresh nonce (see Line 9, Line 14 and Line 24 of Algorithm 13). However, this cannot be for the same client identifier $clientId$ that has been issued previously, as otherwise, the registration access token would change, contradicting Lemma 15.

Thus, the AS will not access the registration access token unless it receives a request containing this token.

Registration Access Token Stored at Client: Let $U = (S^u, E^u, N^u) \rightarrow (S^{u'}, E^{u'}, N^{u'})$ be the processing step in which the client accesses the token stored in $asAccounts[selectedAS][reg_at]$ (with $selectedAS \in \text{dom}(as)$, as shown before). The client accesses the token only in Line 34 of Algorithm 9, where it prepares sending a client management request. For this, it first creates the Authorization header of the request containing the registration access token (Line 35). The client sends either a DELETE request (Line 40 of Algorithm 9) or a PUT request (Line 56 of Algorithm 9) to the domain $S^u(as).asAccounts[selectedAS][reg_client_uri] \equiv selectedAS$ (see Line 33 of Algorithm 9). Note that when sending the management request, the client calls the HTTPS_SIMPLE_SEND function with the CLIENT_MANAGEMENT reference.

In U , the client does not store the registration access token into a different location of its state and does not send any other requests.

Processing the Management Request: Let $P = (S^p, E^p, N^p) \rightarrow (S^{p'}, E^{p'}, N^{p'})$ be the processing step in which the AS processes the request at the /management path, i.e., in one of the following two places:

- Line 20 of Algorithm 11 (if the request is a PUT request): Only the client c can be the creator of the request, as up to this processing step, only c and as can derive the token and as as does not send PUT requests. The AS retrieves the registration access token in Line 25 of Algorithm 11, compares it to access token that it stores for the client identifier in the request in Line 26 of Algorithm 11, but does not change the registration access token stored for this client identifier. The AS responds with an HTTPS response containing the same reg_at and reg_uri values (see Line 38 of Algorithm 11).
- Line 54 of Algorithm 11 (if the request is a DELETE request): In this case, the AS compares the registration access token from the request to the token stored in its state; the AS only modifies $S^p(as).clients[clientId][active]$, for some $clientId'$, and does not send any messages.

Processing the Management Response: Let $Q = (S^q, E^q, N^q) \rightarrow (S^{q'}, E^{q'}, N^{q'})$ be the processing step in which the client processes the response. The client processes responses with the CLIENT_MANAGEMENT reference value only in Line 10 of Algorithm 3. It retrieves the access token in Line 22 of Algorithm 3 and stores this value into $S^{q'}.asAccounts[selectedAS][reg_at]$ (Line 23 of Algorithm 3). This is the same value as stored previously, as the AS does not change the token. Note that the client does not store the token in any other place and does not emit a message containing the token.

At this point, the client could repeat sending management requests. The AS would response as before and would respond with the same token. Overall, we conclude that the registration access token is a fresh nonce chosen by as and sent to c when registering the client, and then sent only to the AS, which will respond with the same value. ■

Lemma 17 (Secrecy of Client Keys Registered at AS). For

- every run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of $\mathcal{F}API$ with a network attacker,
- every authorization server $as \in AS$ that is honest in S^n ,
- every client $c \in C$ that is honest in S^n ,
- every client identifier $clientId \in \mathcal{T}_{\mathcal{N}}$ that has been issued to c by as in some processing step $R = (S^r, E^r, N^r) \rightarrow (S^{r'}, E^{r'}, N^{r'})$ in ρ (according to Definition 11),
- every configuration (S^i, E^i, N^i) (at position i in ρ),

it holds true that if $clientId \in {}^\diamond S^i(as).clients$, then:

$\exists k_{mtls}, k_{jwt} \in \mathcal{N}$ such that

- 1) $S^i(as).clients[clientId][mtls_key] \equiv \text{pub}(k_{mtls})$, and
- 2) $S^i(as).clients[clientId][jwt_key] \equiv \text{pub}(k_{jwt})$, and
- 3) every process $p \neq c$, we have $k_{mtls}, k_{jwt} \notin d_\emptyset(S^n(p))$.

PROOF. Note that an honest AS modifies its `clients` state subterm only in Line 51 and Line 57 of Algorithm 11 (i.e., the /manage endpoint) and Line 24 of Algorithm 13 (the /reg endpoint).

We do a proof by induction over i .

Base Case: $i \leq r'$: Initially, the `clients` state subterm of as is empty (Definition 8). As the AS issues $clientId$ in R , it follows that for all configurations (S', E', N') prior to $(S^{r'}, E^{r'}, N^{r'})$, $clientId \notin {}^\diamond S'(as).clients$, as the AS takes $clientId$ from $S^r(as).pendingClientIds$ (Line 2 of Algorithm 13). Initially, `pendingClientIds` is empty (Definition 8), and the AS adds values to this state subterm only in Line 5 of Algorithm 14, after ensuring that the value that is being stored is not part of the `clients` and `pendingClientIds` state subterms (Line 3 of Algorithm 14).

In R , the AS executes Line 24 of Algorithm 13, as both Line 51 and Line 57 of Algorithm 11 would require that $clientId \in {}^\diamond S^r(as).clients$ (see Line 22 and Line 56 of Algorithm 11).

Let $\langle x, y, \text{enc}_a(\langle \text{regReq}, k \rangle, pk_{as}) \rangle$ be the input event that as processes in R . The request $\text{enc}_a(\langle \text{regReq}, k \rangle, pk_{as})$ was created by c (see Definition 11). The AS takes the key values from regReq , i.e., $\exists i, j \in \mathbb{N}$ s.t.

- $S^{r'}(as).\text{clients}[clientId][\text{mtls_key}] \equiv \text{regReq.body}[\text{jwks}].i.[\text{val}]$, and
- $S^{r'}(as).\text{clients}[clientId][\text{jwt_key}] \equiv \text{regReq.body}[\text{jwks}].j.[\text{val}]$, and
- $\text{regReq.body}[\text{jwks}].i.[\text{use}] \equiv \text{TLS}$, and
- $\text{regReq.body}[\text{jwks}].j.[\text{use}] \equiv \text{sig}$

(see Lines 5-7, Line 14, and Line 24 of Algorithm 13).

A client creates POST requests containing jwks in the body only in Line 26 of Algorithm 8.

The client chooses the values $\text{regReq.body}[\text{jwks}].i.[\text{val}] \equiv \text{pub}(t_1)$, and $\text{regReq.body}[\text{jwks}].j.[\text{val}] \equiv \text{pub}(t_2)$, with t_1 and t_2 being fresh nonces (Lines 13-15 of Algorithm 8). The client calls the `HTTPS_SIMPLE_SEND` function with t_1 and t_2 in the first function argument $reference$, which stores the values (only) in the `pendingDNS` state subterm of the client (Line 2 of Algorithm 34). As the client sent the request to the AS, we conclude that the client processed the corresponding DNS response and stores $reference$ into `pendingRequests` in Line 15 of Algorithm 39 (and removes the value from `pendingDNS` in Line 17 of Algorithm 39).

Thus, the values t_1 and t_2 are stored only at the client, and only in the `pendingRequests` state subterm in a reference value with $reference[\text{responseTo}] \equiv \text{REGISTRATION}$.

If the client never processes the registration response, then it will not retrieve this `pendingRequests` entry, and as the client is honest in S^n , we conclude that the attacker cannot derive t_1 and t_2 .

If the client receives the response, it will process it in Line 40 of Algorithm 3 (as this is the only place where a client processes a response with $reference[\text{responseTo}] \equiv \text{REGISTRATION}$). The client retrieves both values from $reference$ and stores them in its `asAccounts` state subterm in Line 49 of Algorithm 3. Now, we can apply Lemma 4 and Lemma 6 and conclude that t_1 and t_2 will never leak, and in particular, will not be derivable by the attacker in S^n .

Induction Step: We assume that the statement is true for position i and will prove it for $i' := i + 1$. For this, we consider the processing step $I = (S^i, E^i, N^i) \rightarrow (S^{i'}, E^{i'}, N^{i'})$.

If the AS does not change $S^i(as).\text{clients}[clientId]$, then $S^{i'}(as).\text{clients}[clientId][\text{mtls_key}] \equiv S^i(as).\text{clients}[clientId][\text{mtls_key}]$ and $S^{i'}(as).\text{clients}[clientId][\text{jwt_key}] \equiv S^i(as).\text{clients}[clientId][\text{jwt_key}]$, and the property still holds true. Thus, we consider all cases in which the AS changes $S^i(as).\text{clients}[clientId]$. An honest AS modifies its `clients` state subterm only in Line 51 of Algorithm 11 and Line 57 of Algorithm 11 (i.e., the `/manage` endpoint) and in Line 24 of Algorithm 13 (the `/reg` endpoint).

Case 1: Line 51 of Algorithm 11 Here, the AS is processing a DCM update request. Let $\langle x, y, \text{enc}_a(\langle \text{updateReq}, k \rangle, pk_{as}) \rangle$ be the input event of the processing step (the message has this structure as the AS is executing the `PROCESS_HTTPS_REQUEST` function (Algorithm 11), which is only called by the generic HTTPS server in Line 9 of Algorithm 39, i.e., the message is an encrypted HTTP request, see Line 8 of Algorithm 39). The AS updates $S^i(as).\text{clients}[clientId]$ with $clientId \equiv \text{updateReq.body}[clientId]$ (Line 21 of Algorithm 11). The AS also checks that $\text{updateReq.headers}[\text{Authorization}][\text{Bearer}] \equiv S^i(as).\text{clients}[clientId][\text{reg_at}]$ (Line 25 and Line 26 of Algorithm 11).

As shown in Lemma 15, the registration access token never changes, i.e., $S^i(as).\text{clients}[clientId][\text{reg_at}] \equiv S^{r'}(as).\text{clients}[clientId][\text{reg_at}]$. As shown in Lemma 16, only c and as can derive this access token. As an honest AS never sends HTTPS requests with an Authorization header and `clientId` in the request body (the only request that an AS sends with an Authorization header is in Line 304 of Algorithm 11), it follows that c created the request.

The remaining proof is similar to the previous case (DCR): The AS takes the key values from the request, i.e., $\exists i, j \in \mathbb{N}$ s.t.

- $S^{i'}(as).\text{clients}[clientId][\text{mtls_key}] \equiv \text{updateReq.body}[\text{jwks}].i.[\text{val}]$, and
- $S^{i'}(as).\text{clients}[clientId][\text{jwt_key}] \equiv \text{updateReq.body}[\text{jwks}].j.[\text{val}]$, and
- $\text{updateReq.body}[\text{jwks}].i.[\text{use}] \equiv \text{TLS}$, and
- $\text{updateReq.body}[\text{jwks}].j.[\text{use}] \equiv \text{sig}$

(see Lines 29-31, Lines 35-36, and Line 51 of Algorithm 11).

An honest client creates PUT requests only in Line 56 of Algorithm 9 (i.e., when sending a DCM update request), and sets $\text{updateReq.body}[\text{jwks}].i.[\text{val}] := \text{pub}(t_3)$ and $\text{updateReq.body}[\text{jwks}].j.[\text{val}] := \text{pub}(t_4)$, with t_3, t_4 being fresh nonces (see Lines 43-46 and Line 55 of Algorithm 9).

In Line 56 of Algorithm 9, the client calls the `HTTPS_SIMPLE_SEND` function with t_3 and t_4 in the first function argument $reference'$, which stores the values (only) in the `pendingDNS` state subterm of the client (Line 2 of Algorithm 34). When processing the corresponding DNS response, the client stores $reference'$ into `pendingRequests` in Line 15 of Algorithm 39 (and removes the value from `pendingDNS` in Line 17 of Algorithm 39).

Thus, the values t_3 and t_4 are stored only at the client, and only in the `pendingRequests` state subterm in a reference value with $reference'[responseTo \equiv CLIENT_MANAGEMENT]$.

Once the client receives the response, it will process it in Line 10 of Algorithm 3 (as this is the only place where a client processes a response with the `CLIENT_MANAGEMENT` reference value). There, the client retrieves both values from $reference'$ and stores them in its `asAccounts` state subterm in Line 23. Now, we can again apply Lemma 4 and Lemma 6 and conclude that t_3 and t_4 will never leak, and in particular, will not be derivable by the attacker in S^n .

Case 2: Line 57 of Algorithm 11 In this case, the AS changes only $S^i(as).clients[clientId][active]$, i.e., the keys are the same as in S^i and the property still holds true.

Case 3: Line 24 of Algorithm 13 In this case, the AS does not change $S^i(as).clients[clientId]$: The AS chooses a client identifier $clientId'$ from $S^i(as).pendingClientIds$ (Line 2 of Algorithm 13) and stores $S^i(as).clients[clientId']$. However, $clientId \neq clientId'$, as `pendingClientIds` cannot contain a term used as a key for the `clients` state subterm. ■

Lemma 18 (Access Token can only be used by Honest Client). For

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

$$Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \rightarrow E_{out}^Q]{e_{in}^Q \rightarrow rs} (S^{Q'}, E^{Q'}, N^{Q'})$$

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

it holds true that:

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

(I) There exists a processing step

$$P = (S^P, E^P, N^P) \xrightarrow[rs \rightarrow E_{out}^P]{e_{in}^P \rightarrow rs} (S^{P'}, E^{P'}, N^{P'})$$

such that

- 1) either $P = Q$ or P prior to Q in ρ , and
 - 2) e_{in}^P is an event $\langle x, y, enc_a((m_{req}, k_1), k_2) \rangle$ for some x, y, k_1 , and k_2 where $m_{req} \in \mathcal{T}_{\mathcal{N}}$ is an HTTP request which contains a term (access token) t in its Authorization header, i.e., $t \equiv m_{req}.headers[Authorization].2$, and
 - 3) r is a fresh nonce generated in P at the resource endpoint of rs in Line 48 of Algorithm 18.
- (II) t is bound to a key $k \in \mathcal{T}_{\mathcal{N}}$, as , a client identifier $clientId \in \mathcal{T}_{\mathcal{N}}$ and id in S^Q (see Definition 10).
- (III) If there exists a client $c \in C$ such that $clientId$ has been issued to c by as in a previous processing step (see Definition 11), and if c is honest in S^n , then the message in e_{in}^P was created by c .

PROOF. An honest resource server sends HTTPS responses with a resource dictionary key only in Line 84 of Algorithm 18 and Line 45 of Algorithm 19.

Case 1: Line 84 of Algorithm 18

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_{in}^Q is an event containing an HTTPS request, as Algorithm 18 is only called by the generic HTTPS server in Line 9 of Algorithm 39. As the check done in Line 7 of Algorithm 39 was true and the `stop` in Line 8 was not executed, it follows that the input event of Algorithm 39 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 20 of Algorithm 18).

The resource that is sent out in Line 84 of Algorithm 18 is a freshly chosen nonce generated in the same processing step in Line 48 of Algorithm 18 (see also Line 75 and Line 81 of Algorithm 18). This concludes the proof of the first post-condition.

Second Postcondition As Line 84 of Algorithm 18 is executed, it follows that the condition in Line 50 of Algorithm 18 is false, i.e., $extractmsg(m_{req}.headers[Authorization].2)$ is a structured access token (see Lines 23 and 49).

The access token is signed by $authorizationServerOfResource^{rs}(resourceID)$: The value of $responsibleAS$ (in Line 16) is equal to

$$\begin{aligned}
& S^Q(rs).\text{resourceASMapping}[resourceID] && \text{(Line 16 of Algorithm 18)} \\
\equiv & s_0^{rs}.\text{resourceASMapping}[resourceID] && \text{(value is never changed)} \\
\in & \text{dom}(\text{authorizationServerOfResource}^{rs}(resourceID)) && \text{(Definition 9)}
\end{aligned}$$

As required by the precondition of the lemma, $as = \text{authorizationServerOfResource}^{rs}(resourceID)$ is honest in S^Q . The signature of the access token is checked in Line 64 of Algorithm 18 using the verification key

$$\begin{aligned}
& asInfo[as_key] \\
\equiv & S^Q(rs).\text{asInfo}[responsibleAS][as_key] && (responsibleAS \in \text{dom}(as), \text{Line 19}) \\
\equiv & s_0^{rs}.\text{asInfo}[responsibleAS][as_key] && \text{(value is never changed)} \\
\equiv & \text{signkey}(\text{dom}^{-1}(responsibleAS)) && \text{(Definition 9)} \\
\equiv & \text{signkey}(as)
\end{aligned}$$

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

- Line 17 of Algorithm 11: Endpoint returning public key
- Line 97 of Algorithm 11: Signing authorization response
- Line 200 of Algorithm 11: Signing access token
- Line 212 of Algorithm 11: Signing ID token
- Line 227 of Algorithm 11: Signing introspection response

Authorization responses, ID tokens, and introspection responses signed by an authorization server do not contain a `cnf` claim (see Lines 91-97 of Algorithm 11 for authorization responses, Lines 207-212 of Algorithm 11 for ID tokens, and Line 227 of Algorithm 11 for introspection responses). Thus, it follows that $\text{extractmsg}(m_{\text{req}}.\text{headers}[\text{Authorization}].2)$ is an access token created by as in Line 200 of Algorithm 11 (note that the access token checked by the RS contains a non-empty `cnf` value, see Line 29, Line 45, and Line 62 of Algorithm 18).

Let $O = (S^O, E^O, N^O) \xrightarrow[as \rightarrow E_{\text{out}}^O]{e_{\text{in}}^O \rightarrow as} (S^{O'}, E^{O'}, N^{O'})$ 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 $S^{O'}(as).\text{records}.i[\text{access_token}]$, for some natural number i (as Line 203 of Algorithm 11 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 157 and Line 164 of Algorithm 11.

The structured access token contains a value $\text{extractmsg}(m_{\text{req}}.\text{headers}[\text{Authorization}].2)[\text{sub}] \in {}^\diamond S^Q(rs).\text{ids}$ (Line 49, 71, and 72 of Algorithm 18). This identity is used as a dictionary key for storing the resource (see Line 74 of Algorithm 18). The `ids` stored at the resource server are never changed, i.e., $S^Q(rs).\text{ids} \equiv s_0^{rs}.\text{ids}$. When creating the access token, the authorization server takes this value from $S^O(as).\text{records}.i[\text{sub}]$ with the same i as above (Line 157 or Line 164 of Algorithm 11, see also 199 of Algorithm 11). As the remaining lines of the token endpoint do not change this value, it follows that $S^O(as).\text{records}.i[\text{sub}] \equiv S^{O'}(as).\text{records}.i[\text{sub}]$.

From the successful check of Line 62 of Algorithm 18 (as we assume that the resource server returns a resource in Line 84), it follows that either

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

as cnfValue is set in Line 29 or Line 45 of Algorithm 18.

The authorization server sets the `cnf` value of access tokens only in Line 199 of Algorithm 11. The value is determined either in Line 179 or Line 190 of Algorithm 11, and the authorization server stores the `cnf` value into the same record as the `access_token` and `sub` values, see Line 204 of Algorithm 11, i.e., $S^{O'}(as).\text{records}.i[\text{cnf}]$ is either $[\text{jkt} : \text{hash}(k)]$ or $[\text{x5t\#S256} : \text{hash}(k)]$, for some value k .

The record entry also contains the client id value clientId that was authenticated at the endpoint, see Line 148 and Line 165 of Algorithm 11. The AS does not change this value at the token endpoint, i.e., $S^{O'}(as).\text{records}.i[\text{client_id}]$ contains this client id.

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

Third Postcondition

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

Case 1.3.1: AS created the `cnf` value in Line 179 of Algorithm 11:

Client authenticated at AS. Let req_{token} be the token request that the AS processes in O , i.e., $e_{in}^O = \langle x', y', req_{token} \rangle$, for some values x', y' . In Line 179 of Algorithm 11, the AS sets the `cnf` value to $hash(\text{extractmsg}(req_{token}.headers[DPoP])[headers][jwk])$ (see Lines 170, 171, 172, and Line 179 of Algorithm 11).

As the identifier $clientId$ was authenticated at the token endpoint, and as this identifier has been issued to c , it follows that c created the request req_{token} in a previous processing step $L = (S^l, E^l, N^l) \rightarrow (S^l, E^l, N^l)$.

Key to which AT is bound to is only known to client. The token request contains either an authorization code or an authentication request identifier, i.e., $code \in req_{token}.body$ or $auth_req_id \in req_{token}.body$ (see Line 156 and Line 163 of Algorithm 11). An honest client creates requests containing an authorization code or an authentication request identifier only in Line 43 of Algorithm 4 and Line 40 of Algorithm 5. In both cases, it holds true that $\text{extractmsg}(req_{token}.headers[DPoP])[headers][jwk] \equiv pub(clientSignKey)$, with $clientSignKey \equiv S^l(c).asAccounts[selectedAS][sign_key]$ and some value $selectedAS$ (see Lines 12, 37-40 of Algorithm 4 and Lines 9, 34-37 of Algorithm 5). As shown in Lemma 4, only c can derive $clientSignKey$, i.e., $clientSignKey \notin d_\emptyset(S^m(p))$ for all processes $p \neq c$.

Request was created by client. As the structured access token contains the value $accessTokenContent[cnf].1 \equiv jkt$, and $accessTokenContent[cnf].2$ is set to $hash(pub(clientSignKey))$, and as the RS checks these values against the resource request m_{req} (Line 62 of Algorithm 18), it follows that the RS executed Line 45 of Algorithm 18 (as this is the only place where the RS creates a value $cnfValue$ with $cnfValue.1 \equiv jkt$). The corresponding key is taken from the resource request, i.e., the key is $\text{extractmsg}(m_{req}.headers[DPoP])[headers][jwk]$ (see Lines 31-33 of Algorithm 18)

All preconditions of Lemma 14 are true, with $dpopProof \equiv m_{req}.headers[DPoP]$:

- $checksig(dpopProof, pub(clientSignKey)) \equiv \top$ (see Line 34 of Algorithm 18)
- $\text{extractmsg}(dpopProof)[payload][htu].host \in \text{dom}(rs)$ (see Line 38 of Algorithm 18 and Lemma 2)
- $ath \in {}^\diamond \text{extractmsg}(dpopProof)[payload]$, (see Line 42 of Algorithm 18)
- $\text{extractmsg}(dpopProof)[payload][nonce] \in S^Q(rs).dpopNonces$ (see Line 40 of Algorithm 18)

As $clientSignKey \notin d_\emptyset(S^m(p))$ for all processes $p \neq c$ (see above), we can apply Lemma 14 and conclude that in S^Q , $dpopProof$ can only be known by c and rs . The only places where a resource server sends a request are Lines 60 and 82 of Algorithm 18. 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_{out}^Q in Line 47 of Algorithm 18. In the latter case, the resource server leaks the resource request – but only after invalidating the mTLS nonce (Lines 26f. of Algorithm 18) or DPoP nonce (Line 44 of Algorithm 18), i.e., processing this request again would lead to an empty E_{out}^Q in Line 26 of Algorithm 18, or Line 40 of Algorithm 18. 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_{in}^P .

Case 1.3.2: AS created the `cnf` value in Line 190 of Algorithm 11:

Note that in this case, $S^O(as).clients[clientId][client_type]$ is equal to `pkjwt_mTLS` or `mTLS_mTLS` (see Line 167 and Line 180 of Algorithm 11). The structured access token contains the value $accessTokenContent[cnf].1 \equiv x5t\#S256$, and $accessTokenContent[cnf].2$ is set to $hash(mTlsKey)$. The value $mTlsKey$ is set to $mtlsInfo.2$ in Line 189 of Algorithm 11. The sequence $mtlsInfo$ is chosen in Line 148 or Line 187 of Algorithm 11. In both cases, $mTlsKey$ is set to $S^m(as).clients[clientId][mtls_key]$, with (S^m, E^m, N^m) being some configuration prior to (S^O, E^O, N^O) :

- Line 148 of Algorithm 11: $mtlsInfo$ is the third entry of the return value of `AUTHENTICATE_CLIENT` (Algorithm 12). `AUTHENTICATE_CLIENT` determines the client identifier $clientId$ from the HTTP request and also determines the type of the client (see Lines 7, 8, 20, 21). As the type of the client is either `pkjwt_mTLS` or `mTLS_mTLS`, the body of the request does not contain a value `client_assertion`, as otherwise, the **stop** in Line 10 of Algorithm 12 would have prevented the authorization server to issue the access token. In particular, the **return** in Line 30 was executed and the third return value was taken from $S^O(as).mtlsRequests[clientId]$ (Line 19 of Algorithm 12; Note that this is the same client identifier to which the token is bound). Initially, the `mtlsRequests` subterm of the authorization server's state is empty (see Definition 8), i.e., the AS added $mtlsInfo$ in some processing step $M = (S^m, E^m, N^m) \rightarrow (S^{m'}, E^{m'}, N^{m'})$.

The authorization server adds values to `mtlsRequests` only in Line 238 of Algorithm 11. The second sequence entry is $S^m(as).clients[clientId][mtls_key]$ (see Line 235 of Algorithm 11).

- Line 187 of Algorithm 11: $mtlsInfo$ is taken from $S^O(as).mtlsRequests[clientId]$. As shown in the previous case, the second sequence entry of $mtlsInfo$ is equal to $S^m(as).clients[clientId][mtls_key]$, for some configuration previous (S^m, E^m, N^m) .

When adding values to $mtlsRequests$ in Line 238 of Algorithm 11, the authorization server ensures that the value of the key is not $\langle \rangle$ (Line 236 of Algorithm 11), i.e., $clientId \in \langle \rangle S^m(as).clients$. Thus, we can apply Lemma 17 and conclude that there exists a nonce k_{mtls} such that $S^m(as).clients[clientId][mtls_key] \equiv \text{pub}(k_{mtls})$ and for every process $p \neq c$ it holds true that $k_{mtls} \notin d_\emptyset(S^n(p))$.

The structured access token contains the values $accessTokenContent[cnf].1 \equiv \text{x5t\#S256}$ and $accessTokenContent[cnf].2 \equiv \text{hash}(\text{pub}(k_{mtls}))$. Thus, the resource server executes Line 29 of Algorithm 18 (in the processing step P). This means that e_{in}^P contains a value $mtlsNonce$ in the body of the request such that $\langle mtlsNonce, \text{pub}(k_{mtls}) \rangle \in \langle \rangle S^P(rs).mtlsRequests$ (see Lines 25, 26, 62).

If the client c is honest in S^n , then it is also honest in S^P , and we can apply Lemma 11 and conclude that only c and rs can derive $m_{req}.body[\text{TLS_binding}]$. 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 45 of Algorithm 19

First Postcondition In Line 45 of Algorithm 19, the resource server is processing an HTTP response $resp_{introsp}$ (with the reference $\text{TOKENINTROSPECTION}$, see Line 2 of Algorithm 19). An honest resource server sends HTTP requests with this reference value only by calling HTTPS_SIMPLE_SEND in Line 61 of Algorithm 18. Let $req_{introsp}$ be the corresponding request to $resp_{introsp}$. 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 $t \equiv m_{req}.headers[\text{Authorization}].2$ (Line 20 of Algorithm 18). The resource r that the resource server sends out in Line 45 of Algorithm 19 (in the processing step Q) was stored by the resource server in $S^{P'}.pendingResponses$ in Line 53 of Algorithm 18, and the resource was generated in Line 48 of Algorithm 18 (in the processing step P).

Second Postcondition The request $req_{introsp}$ was sent by rs to a domain of as : $responsibleAS$ in Line 16 of Algorithm 18 is a domain of as , as shown in the proof of the first case. Thus, it follows that $S^P(rs).asInfo[responsibleAS][as_introspect_ep]$ is $\langle \text{URL}, \text{S}, dom_{as}, /introspect, \langle \rangle, \perp \rangle$, with $dom_{as} \in \text{dom}(as)$ (see Definition 9).

Furthermore, $req_{introsp}$ contains the value $m_{req}.headers[\text{Authorization}].2$, see Line 23 and Line 59 of Algorithm 18. The authorization server as processes this request in the introspection endpoint in Line 216 of Algorithm 11. As the resource server did not stop in Line 23 of Algorithm 19, we conclude that the access token sent by the resource server in P is active, i.e., the authorization server executed Line 225 of Algorithm 11. Thus, there is a value $record$ in the $records$ state subterm of the authorization server's state with the access token (Line 220 of Algorithm 11), and in this record, there is a cnf and a $subject$ entry (Line 225 of Algorithm 11) 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 28 of Algorithm 19).

An honest authorization server adds cnf values to an entry of its $records$ state entry only in the token endpoint in Line 204 of Algorithm 11. Thus, this value is either $[\text{jkt}: \text{hash}(k)]$ (see Line 179 of Algorithm 11), or $[\text{x5t\#S256}: \text{hash}(k)]$ (see Line 190 of Algorithm 11), for some value k .

In addition, the record entry also contains the client id value $clientId$ that was authenticated at the endpoint, see Line 148 and Line 165 of Algorithm 11.

Third Postcondition The resource server checks in Line 25 of Algorithm 19 that the cnf value that the authorization server put into the response $resp_{introsp}$ is equal to the $cnfValue$ that the resource server stored in Line 53 of Algorithm 18 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 84 of Algorithm 18). 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 19 (Redirect URI Properties). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system \mathcal{FAP} with a network attacker, every configuration (S, E, N) in ρ , every authorization server $as \in \text{AS}$ that is honest in S , every client $c \in \text{C}$ that is honest in S , every client identifier $clientId$ that has been issued to c by as in a previous processing step (see Definition 11), and every $requestUri$, all redirect URIs for c stored at as are HTTPS URIs and belong to c . Or, more formally: Let $rec = S(as).authorizationRequests[requestUri]$, then $rec[client_id] \equiv clientId$ implies both $rec[redirect_uri].protocol \equiv \text{S}$, and $rec[redirect_uri].host \in \text{dom}(c)$

PROOF. Initially, the `authorizationRequests` state subterm of `as` is empty (see Definition 8). The only places in which an honest authorization server writes to its `authorizationRequests` state subterm are:

- Line 73 of Algorithm 11: Here, the authorization server does not change or create values under the `client_id` or `redirect_uri` keys.
- Line 142 of Algorithm 11: 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. Let `reqBody` := `req.body` if the request is not signed, and otherwise, let `reqBody` := `extractmsg(req.body)`. In order to get to Line 142 of Algorithm 11, `req` must contain valid client authentication data (see Lines 116 and 120), in particular, `reqBody` 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 12 together with Line 12 of Algorithm 12 and Line 120 of Algorithm 11 ensure that `extractmsg(reqBody[client_assertion])[iss] ≡ reqBody[client_id]`. We note that reaching Line 142 of Algorithm 11 implies that the current processing step will output an event (there are no **stops** between Line 142 and Line 144 of Algorithm 11). Hence, we can apply Lemma 13.

When reaching Line 142 of Algorithm 11, `req` also must contain a `redirectUri` value in `reqBody[redirect_uri]` (see also Line 123 of Algorithm 11). Furthermore, this `redirectUri` must be an HTTPS URI (Line 125 of Algorithm 11) and this is the value stored in the authorization server's `authorizationRequests` state subterm (in a record under the key `redirect_uri`), together with `reqBody[client_id]` (under key `client_id`).

Line 114 of Algorithm 11 ensures that `reqBody` contains a field `code_challenge_method` with value `S256`.

From Lemma 13, 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 68 of Algorithm 8 (with the corresponding part of the body containing the `code_challenge_method` value being chosen in Line 49 of Algorithm 8), we can conclude that the value of `reqBody[redirect_uri]` is the one selected in Lines 2f. of Algorithm 8. This implies `req.body[redirect_uri].host ∈ dom(c)` (or `extractmsg(req.body)[redirect_uri].host ∈ dom(c)` in the case of a signed request) and hence `rec[redirect_uri].host ∈ dom(c)`. ■

Lemma 20 (Integrity of Client's Session Storage). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every client $c \in \mathcal{C}$ that is honest in S , and every login session id $lsid$, we have that if $lsid \in S(c).sessions$, then all of the following hold true:

- 1) `selected_AS ∈ S(c).sessions[lsid]`
- 2) `cibaFlow ∈ S(c).sessions[lsid]`
- 3) for all configurations (S', E', N') after (S, E, N) in ρ we have $S'(c).sessions[lsid][selected_AS] \equiv S(c).sessions[lsid][selected_AS]$
- 4) for all configurations (S', E', N') after (S, E, N) in ρ we have $S'(c).sessions[lsid][cibaFlow] \equiv S(c).sessions[lsid][cibaFlow]$

PROOF. Since we have $S^0(c).sessions \equiv \langle \rangle$ (Definition 7), we know that if $lsid \in S(c).sessions$, such an entry must have been stored there by `c`. Clients only ever store/add such an entry in Line 10 of Algorithm 2 and Line 40 of Algorithm 2. In both cases, the keys `selected_AS` and `cibaFlow` are part of the stored entry, and 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 the keys `selected_AS` and `cibaFlow`.

It is easy to see that Line 10 and Line 40 of Algorithm 2 are indeed the only places in which a client stores any value under the `selected_AS` and `cibaFlow` keys in the `sessions` state subterm. Similarly, it is easy to check that these lines are also the only places in which a client (over)writes a whole entry in the `sessions` state subterm. Hence, we can conclude: The `selected_AS` and `cibaFlow` keys are present whenever a client adds an entry to the `sessions` state subterm and neither the value stored under these keys, nor the `sessions` entry itself are overwritten or removed anywhere, implying 1) and 2). In addition, if the client ever executes Line 10 or Line 40 of Algorithm 2 again, it will never overwrite an existing entry, because it will use a fresh login session id, thus we have 3) and 4). ■

Lemma 21 (Integrity of Client's oauthConfigCache). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every authorization server $as \in \mathcal{AS}$ that is honest in S , every client $c \in \mathcal{C}$ that is honest in S , and every domain $d \in \text{dom}(as)$, it holds true that if $d \in S(c).oauthConfigCache$, we have all of the following:

- 1) $S(c).oauthConfigCache[d][issuer] \equiv d$
- 2) $S(c).oauthConfigCache[d][auth_ep] \equiv \langle \text{URL}, S, d, /auth, \langle \rangle, \perp \rangle$
- 3) $S(c).oauthConfigCache[d][token_ep] \equiv \langle \text{URL}, S, d, /token, \langle \rangle, \perp \rangle$
- 4) $S(c).oauthConfigCache[d][par_ep] \equiv \langle \text{URL}, S, d, /par, \langle \rangle, \perp \rangle$
- 5) $S(c).oauthConfigCache[d][introspec_ep] \equiv \langle \text{URL}, S, d, /introspect, \langle \rangle, \perp \rangle$

- 6) $S(c).oauthConfigCache[d][jwks_uri] \equiv \langle \text{URL}, S, d, /jwks, \langle \rangle, \perp \rangle$
- 7) $S(c).oauthConfigCache[d][reg_ep] \equiv \langle \text{URL}, S, d, /reg, \langle \rangle, \perp \rangle$
- 8) $S(c).oauthConfigCache[d][backchannel_authentication_endpoint] \equiv \langle \text{URL}, S, d, /backchannel-authn, \langle \rangle, \perp \rangle$

We note that this implies that all these entries in $S(c).oauthConfigCache[d]$ 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 $S^0(c).oauthConfigCache \equiv \langle \rangle$ (Definition 7), i.e., the `oauthConfigCache` state subterm is initially empty. An honest client only ever writes to its `oauthConfigCache` state subterm in Line 38 of Algorithm 3 when processing an HTTPS response. Hence, $d \in S(c).oauthConfigCache$ implies that there must have been a processing step $Q = (S^Q, E^Q, N^Q) \rightarrow (S^{Q'}, E^{Q'}, N^{Q'})$ in ρ such that $d \notin S^Q(c).oauthConfigCache$ and $d \in S^{Q'}(c).oauthConfigCache$. In Q , `PROCESS_HTTPS_RESPONSE` must have been called with a *reference* as second argument, such that $reference[responseTo] \equiv \text{CONFIG}$. In addition, $reference[session]$ must contain a value $sessionId$ such that $S^Q(c).sessions[sessionId][selected_AS] \equiv m.body[issuer]$ (Line 36 of Algorithm 3). From Line 38 of Algorithm 3, we also know that $S^Q(c).sessions[sessionId][selected_AS] \equiv d$ (cf. Lemma 20). Hence, we already have that $d \in S(c).oauthConfigCache$ implies 1).

With Lemma 3, we have that there must be a processing step $P = (S^P, E^P, N^P) \rightarrow (S^{P'}, E^{P'}, N^{P'})$ 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 8. The accompanying message's host value there is $S^P(c).sessions[sessionId][selected_AS]$, i.e., by Lemma 20, d . That same message's path value is either `/.well_known/openid-configuration` or `/.well_known/oauth-authorization-server`. From Lemma 46, Algorithm 34, and Lines 10ff. of Algorithm 39 (and because as does not leak $tlskey(d)$), 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 11. From looking at those Lines, it is obvious that the response sent in Line 14 of Algorithm 11 contains a body with a dictionary fulfilling 2)–8). Using Lemma 46 once more, we can conclude that c processes such a response in Q and thus we have 2)–8). ■

Lemma 22 (Authorization code secrecy). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every authorization server $as \in \text{AS}$ that is honest in S , every client $c \in \text{C}$ that is honest in S , every client identifier $clientId$ that has been issued to c by as in a previous processing step (see Definition 11), every identity $id \in \text{ID}^{as}$ with $b = \text{ownerOfID}(id)$ being an honest browser in S , every authorization code $code \neq \perp$ for which there is a record $rec \in \langle \rangle S(as).records$ with $rec[code] \equiv code$, $rec[client_id] \equiv clientId$, and $rec[subject] \equiv id$ and every process $p \notin \{as, c, b\}$, it holds true that $code \notin d_0(S(p))$.

PROOF.

- 1) For $code$ to end up in $(S(as).records.x)[code]$ (with $x \in \mathbb{N}$), the as has to execute Line 89 of Algorithm 11, since the only other places where an honest authorization server writes to the – initially empty, see Definition 8 – `records` state subterm are:
 - Line 194 of Algorithm 11: This line overwrites the stored authorization code with \perp , i.e., codes written by this line are not relevant to this lemma.
 - Line 196 of Algorithm 11: This line overwrites a stored authorization request identifier with \perp .
 - Line 203 of Algorithm 11 and Line 204 of Algorithm 11: 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 157 of Algorithm 11).
 - Line 297 of Algorithm 11: Here, the client adds a new entry to `records`. The client takes a value from `cibaAuthnRequests` in Line 293 of Algorithm 11 and adds `subject`, `issuer`, and `auth_req_id` values in Lines 294f. of Algorithm 11. `cibaAuthnRequests` is initially empty (see Definition 8). The entries that the client adds to `cibaAuthnRequests` in Line 263 of Algorithm 11 do not contain a code value (see Lines 252ff. of Algorithm 11). The client modifies existing `cibaAuthnRequests` entries in Line 277 of Algorithm 11 (modifying the `ciba_auth2_reference` value), in Line 292 of Algorithm 11 (modifying the `authenticateUser` value), and in Line 9 of Algorithm 15 (modifying the `cibaUserAuthNNonce` value). Thus, we conclude that records in `cibaAuthnRequests` do not contain a code value, and therefore, records added in Line 297 of Algorithm 11 do not contain a code value.
- 2) A $code$ stored in Line 89 of Algorithm 11 is a fresh nonce (Line 88 of Algorithm 11). Hence, a $code$ generated by as in that line in some processing step $s_i \rightarrow s_{i+1}$ is not known to any process up to and including s_i . Let e_{in} be the event processed by as in $s_i \rightarrow s_{i+1}$. In order to reach Line 89 of Algorithm 11, e_{in} must contain an HTTPS request req to the

/auth2 endpoint. The only place in which an honest as sends out the $code$ value is the HTTPS response to req – i.e., if the sender of req is honest, this response is only readable by the sender of req .

- 3) In addition, req must contain a valid $identity$ – $password$ combination – because as stores $code$ along with $identity$ and $clientId$ only if $password \equiv \text{secretOfID}(identity)$. Since as does not send requests to itself and $\text{secretOfID}(identity)$ is only known to as and $\text{ownerOfID}(identity)$, req must have been created by $\text{ownerOfID}(identity)$ if the sender of req is honest. W.l.o.g., let $identity \equiv id$, i.e., req was created by b .
- 4) Since the origin header of req must be a domain of as and req must use the POST method, we know that req was initiated by a script of as . In particular, req must have been initiated by $script_as_form$ (as this is the only script ever sent by as that triggers requests to the /auth2 path; the only other script provided by as is $script_as_ciba_form$ (Algorithm 17), which triggers messages to the /ciba-auth2 endpoint). 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 102 of Algorithm 11). 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 $S(as).authorizationRequests[requestUri][redirect_uri]$ where $S(as).authorizationRequests[requestUri][client_id] \equiv clientId$. With Lemma 19, 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 2 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 2. If the request is processed in Lines 2ff. of Algorithm 2, c only responds with a script and does not use $code$ at all. Similarly, if the request is processed in Lines 35ff. of Algorithm 2, Lines 46ff. of Algorithm 2, or Lines 51ff. of Algorithm 2, the client would not use the $code$ value (and also not store the complete message in its state). This leaves us with Lines 12ff. of Algorithm 2; 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 4. There, the $code$ is included in the body of an HTTPS request under the key `code` (Line 8 of Algorithm 4).
- 5) The HTTPS request (“token request”) prepared in Lines 8ff. of Algorithm 4 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 2 and Line 69 of Algorithm 3). 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 8, Line 38 of Algorithm 3, and Lemma 21), 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 46 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 145ff. of Algorithm 11, It is easy to see that the $code$ is not stored or send out there, hence, it cannot leak. ■

Lemma 23 (Unique Code Verifier for Each Login Session ID at Client). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S^i, E^i, N^i) in ρ , every client $c \in \mathcal{C}$ that is honest in S^i with client identifier $clientId$ issued to c by as (in some processing step $s^{cid} \rightarrow s^{cid'}$), every login session id $lsid$, and every term $codeVerifier$, we have that $S^i(c).sessions[lsid][code_verifier] \equiv codeVerifier$ implies:

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

PROOF. We start by noting that an honest client only ever stores something in an entry in `sessions` under key `code_verifier` in Line 53 of Algorithm 8. The value stored there is always a fresh nonce (see Line 47 of Algorithm 8). Hence, we can conclude (II).

To get (I), we need to prove that a stored code verifier is never overwritten. For this, we show that a client executes Line 53 of Algorithm 8 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 8 (`PREPARE_AND_SEND_INITIAL_REQUEST`) is called. Note that the first argument to Algorithm 8 is the aforementioned $sessionId$:

Line 11 of Algorithm 2 Here, the first argument is a fresh nonce (see Line 9 of Algorithm 2), i.e., this line will never lead to Algorithm 8 being called a second time with a given $sessionId$.

Line 39 of Algorithm 3 This line is only executed when the client processes an HTTPS response such that Algorithm 3 (`PROCESS_HTTPS_RESPONSE`) was called with a $reference$ containing a key `responseTo` with value `CONFIG`. The $sessionId$ value used when calling Algorithm 8 is also taken from the $reference$ (see Line 32 of Algorithm 3). 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 39, where the value for *reference* is taken from the client's `pendingRequests` state subterm. The `pendingRequests` state subterm is initially empty (Definition 7) and the only place where elements are added to this state subterm is Line 15 of Algorithm 39. 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 34, 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* such that $reference[responseTo] \equiv \text{CONFIG}$. The only place where such a *reference* is passed to `HTTPS_SIMPLE_SEND` is Line 9 of Algorithm 8. However, this call always ends in a **stop** and the call happens *before* the client executes Line 53 of Algorithm 8 – hence, if an execution of Algorithm 8 leads to execution of Line 11 of Algorithm 2 and thus a subsequent call of Algorithm 8 (when processing the response), both calls use the same *sessionId*, but Line 53 of Algorithm 8 (i.e., storing a code verifier) is executed at most once.

Line 57 of Algorithm 3 This case is very similar to the previous one, except for the following changes: The `responseTo` value in question is `REGISTRATION` instead of `CONFIG`, and the (only) place in which `HTTPS_SIMPLE_SEND` is called with a suitable *reference* is Line 26 of Algorithm 8.

Line 63 of Algorithm 9 Here, the value for the first argument to Algorithm 8 is taken from a record in the client's `pendingCIBARRequests` state subterm (Line 61 of Algorithm 9). Since that record is immediately removed from said state subterm in Line 62 of Algorithm 9, before even calling Algorithm 8, this call cannot happen twice for a given record. Hence, we have to examine where these records come from. Initially, the `pendingCIBARRequests` state subterm is empty (Definition 7) and the only place where elements are added to this state subterm is Line 41 of Algorithm 2. There, the value in question is a fresh nonce (Line 38 of Algorithm 2). Hence, the call to Algorithm 8 in Line 63 of Algorithm 9 always uses a fresh value for the first argument. ■

Lemma 24 (Request URIs do not Leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every client identifier *clientId*, every authorization server $as \in AS$ that is honest in S , every client $c \in C$ that is honest in S and that has been issued client identifier *clientId* by as (in some processing step $s^{cid} \rightarrow s^{cid'}$), every browser $b \in B$ that is honest in S , every domain $d_c \in \text{dom}(c)$, every login session id *lsid*, every nonce *codeVerifier* with

- (1) $\langle \langle _Host, sessionId \rangle, \langle lsid, \top, \top, \top \rangle \rangle \in S(b).cookies[d_c]$, and
- (2) $S(c).sessions[lsid][code_verifier] \equiv codeVerifier$, and
- (3) $S(c).sessions[lsid][selected_AS] \in \text{dom}(as)$, and
- (4) c does not leak the authorization request for *lsid* (see Definition 22),

then all of the following hold true:

- (I) There is exactly one nonce *requestUri*, such that $S(as).authorizationRequests[requestUri][code_challenge] \equiv \text{hash}(codeVerifier)$, and
- (II) 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.

- (A) **PAR endpoint uses TLS.** All requests (and responses) at an authorization server's pushed authorization request (PAR) endpoint must be HTTPS requests (see Lines 2ff. of Algorithm 11), 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).
- (B) **hash(*codeVerifier*) does not leak.** We start off by showing that $\text{hash}(codeVerifier)$ does not leak to any process other than c and as . For this, we look at how *codeVerifier* (from (2)) 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_INITIAL_REQUEST` in Line 53 of Algorithm 8. That value is generated in the same function in Line 47 as a fresh nonce. Hence, at this point, $\text{hash}(codeVerifier)$ is only derivable by c . `PREPARE_AND_SEND_INITIAL_REQUEST` ends with the client sending a PAR request which contains $\text{hash}(codeVerifier)$ under the key `code_challenge`. So we have to check who receives/can decrypt 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 21), we know from (3) that the PAR request is sent to, i.e., encrypted for, 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 103ff. of Algorithm 11). There, the value stored under `code_challenge` – i.e., $\text{hash}(codeVerifier)$ – is stored in an authorization request record in the authorization server's authorization requests storage (see Lines 127, 137, and 142 of Algorithm 11). 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 89 of Algorithm 11). Hence, the value $\text{hash}(\text{codeVerifier})$ 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 8f. of Algorithm 4) 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 (3) and Lemma 21. 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 154 of Algorithm 11. This value is not stored by the authorization server, neither is it sent anywhere. Hence, `codeVerifier` does not leak.

- (C) as stores $\text{hash}(\text{codeVerifier})$. Because the cookie from (1) includes the `__Host` prefix and b is honest, that cookie must have been set by c : the cookies state subterm is initially empty (Definition 74), cookies with the `__Host` prefix are only added in Line 4 of Algorithm 29, where the browser ensures the cookie was received via a secure connection (Definition 70). Note that Line 11 of Algorithm 28 cannot add cookies with `httpsOnly` set to \top (such as the one in (1)) to the browser's state, because they get filtered out (see Definition 69).

Clients only ever set cookies with `sessionId` in the cookie name in two places: when processing a request to the `/start_ciba` endpoint in Lines 35ff. of Algorithm 2 – however, in that case, the corresponding record $S(c).\text{sessions}[lsid]$ in the client's session storage has the value \top stored under the `cibaFlow` key. This value never changes (Lemma 20), and Lemma 26 gives us $\text{code_verifier} \notin S(c).\text{sessions}[lsid]$, i.e., a contradiction to (2).

Hence, the only place left where a client sets cookie with `sessionId` in the cookie name is when processing PAR responses in Lines 58ff. of Algorithm 3. With (2) (note that a client will never change the value stored under `code_verifier`, see Lemma 23), this implies that c sent a PAR request containing $\text{hash}(\text{codeVerifier})$ to as (see (B)) and got a response (because the `reference[responseTo]` value to reach Lines 58ff. of Algorithm 3 must be PAR, see also Lemma 3). Hence, as must have processed that PAR request as described in (B). Part of that processing is to store the value of `code_challenge` from the request – i.e., $\text{hash}(\text{codeVerifier})$ here – in the authorization request storage. Thus, we can conclude that there must be some $\text{requestUri}'$ such that $S(as).\text{authorizationRequests}[\text{requestUri}'][\text{code_challenge}] \equiv \text{hash}(\text{codeVerifier})$.

- (D) **Proof for (I).** From (B), we have that only c and as know the value $\text{hash}(\text{codeVerifier})$ and do not use it in any request except for a single PAR request from c to as . From (C), we have that as stores $\text{hash}(\text{codeVerifier})$ as part of processing that PAR request. As as will use a fresh nonce as request URI for every processed PAR request (see Line 130 of Algorithm 11), and never changes the stored values (except for `code`), we can conclude that there is exactly one requestUri such that $S(as).\text{authorizationRequests}[\text{requestUri}][\text{code_challenge}] \equiv \text{hash}(\text{codeVerifier})$.
- (E) **Proof for (II).** 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, or more precisely, the only one who can decrypt, that PAR response is c in (A), 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 in Lines 58ff. of Algorithm 3. 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 (3)). By (4), we also know that c does not execute Line 75 of Algorithm 3, 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 64 of Algorithm 11 – 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 $\langle\langle_\text{Host}, \text{sessionId}\rangle, \langle lsid, \top, \top, \top \rangle\rangle$. Note that this is the only place where c sets such a cookie (see (C) for why this cookie cannot originate from Line 42 of Algorithm 2).

Since we know from (1) that b knows such a cookie, and (C) implies that c must have set this cookie, we know that the HTTPS response containing the redirect with requestUri , sent by c , was processed (and in particular: decrypted) by b , i.e., was sent to/encrypted for b .

We now only have to show that b does not leak requestUri . The aforementioned redirect contains a location header (Line 68 of Algorithm 3) and status code 303, hence b will enter the location header handling in Line 11 of Algorithm 29 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 29. 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 $requestUri \notin d_0(S(p))$. ■

Lemma 25 (CIBA Login Session IDs do not Leak). For any run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every browser $b \in \mathbb{B}$ that is honest in S^n , every client $c \in \mathbb{C}$ that is honest in S^n , every domain $d_c \in \text{dom}(c)$, every term $bindingMsg \in \mathcal{T}_{\mathcal{N}}$, every term $lsid \in \mathcal{T}_{\mathcal{N}}$, with

- (a) $\langle d_c, bindingMsg \rangle \in \langle \rangle S(b).cibaBindingMessages$, and
- (b) $S(c).sessions[lsid][binding_message] \equiv bindingMsg$

it hold true that only b and c know $lsid$, i.e., for all processes $p \notin \{b, c\}$, we have $lsid \notin d_0(S^n(p))$.

PROOF. Let $\langle d_c, bindingMsg \rangle \in \langle \rangle S(b).cibaBindingMessages$. Initially, the `cibaBindingMessages` state subterm of the browser is empty, i.e., $s_0^b.cibaBindingMessages \equiv \langle \rangle$ (Definition 74).

Thus, there exists a processing step

$$Q = (S^Q, E^Q, N^Q) \xrightarrow[b \rightarrow E_{out}^Q]{e_{in}^Q \rightarrow b} (S^{Q'}, E^{Q'}, N^{Q'})$$

with $(S^{Q'}, E^{Q'}, N^{Q'})$ prior to (S, E, N) in which the browser adds this pairing to `cibaBindingMessages`. An honest browser adds entries to `cibaBindingMessages` only in Line 66 of Algorithm 30.

Here, the browser is processing an HTTPS response, i.e., there exists a term m s.t. $e_{in}^Q = \langle _, _, m \rangle$ and there exists a key $k \in \mathcal{T}_{\mathcal{N}}$ and a term $plaintext \in \mathcal{T}_{\mathcal{N}}$ such that $plaintext \equiv \text{dec}_s(m, key)$, $\pi_1(plaintext) \equiv \text{HTTPResp}$ (Line 60 of Algorithm 30) and $binding_message \in \langle \rangle plaintext.body$ (Line 65 of Algorithm 30).

The browser stores the values $\langle request.host, plaintext.body[binding_message] \rangle$ into $S^{Q'}(b).cibaBindingMessages$, where $request$ is a term stored along with the key in `pendingRequests`, i.e., $\langle _, request, _, key, _ \rangle \in \langle \rangle S^{Q'}(b).pendingRequests$ (see Line 60 of Algorithm 30).

Initially, `pendingRequests` is empty (Definition 74). An honest browser adds values to `pendingRequests` with a key only when sending out HTTPS requests in Line 81 of Algorithm 30 and previously storing the request in `pendingRequests` (Line 76 of Algorithm 30), i.e., in a previous processing step, the browser emitted an HTTPS request with $request$ being the HTTP message. This HTTPS request is encrypted asymmetrically with the key $s_0^b.keyMapping[d_c]$ (note that the browser never changes its `keyMapping` state subterm).

I.e., only c can decrypt the message

Thus, the response was created by c in a previous processing step

$$P = (S^P, E^P, N^P) \xrightarrow[c \rightarrow E_{out}^P]{e_{in}^P \rightarrow c} (S^{P'}, E^{P'}, N^{P'})$$

with $(S^{P'}, E^{P'}, N^{P'})$ prior to $(S^{Q'}, E^{Q'}, N^{Q'})$. An honest client creates messages with the key `binding_message` used in the body only in Line 43 of Algorithm 2 and Line 43 of Algorithm 8. In the second case, the client creates an HTTP request (see Line 68 of Algorithm 8), i.e., the client created the response m (from e_{in}^Q) in Line 43 of Algorithm 2.

Before emitting the response, the client stores the binding message into $S^{P'}(c).sessions[lsid][binding_message]$ (Line 40 of Algorithm 2). Note that both the binding message and $lsid$ are chosen as fresh nonces (Line 38 and Line 39 of Algorithm 2).

Besides of using the session id as a dictionary key, the client creates a Set-Cookie header with the value $lsid$ (Line 42 of Algorithm 2) and adds this header to the response (Line 44 of Algorithm 2). Only b can decrypt the response. When processing the response, the browser adds $lsid$ to its cookies.

The browser will only send $lsid$ as cookies in requests to c , and only when sending HTTPS requests. When processing requests, the client does not store or send out cookie values received in requests. Also, the client never sends out a `sessions` dictionary key (i.e., session ids), and for a given record, never modifies the session id or binding message (i.e., a session id known by a different process never replaces $lsid$).

Lemma 26 (Client CIBA Session has no PKCE Verifier). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every client $c \in \mathbb{C}$ that is honest in S , every term $lsid \in \mathcal{T}_{\mathcal{N}}$, it holds true that if $S(c).sessions[lsid][cibaFlow] = \top$, then $code_verifier \notin S(c).sessions[lsid]$.

PROOF. We assume that $S(c).sessions[lsid][cibaFlow] = \top$ and $code_verifier \in S(c).sessions[lsid]$.

Initially, the `sessions` state subterm of c is empty (Definition 7). New sessions are added to `sessions` only in Line 10 of Algorithm 2 and Line 40 of Algorithm 2, and do not contain a `code_verifier` value. Let $Q = (S^Q, E^Q, N^Q) \rightarrow (S^{Q'}, E^{Q'}, N^{Q'})$ be the processing step in which the client updates the $lsid$ session by adding the `code_verifier` value.

This happens only in Line 53 of Algorithm 8. From the check done by the client in Line 42 of Algorithm 8, it follows that $S^Q(c).sessions[lsid][cibaFlow]$ is not \top . However, the client never changes the `cibaFlow` value of an existing session (Lemma 20), i.e., $S(c).sessions[lsid][cibaFlow]$ is not \top , contradicting the assumption.

Lemma 27 (Client Session Containing CIBA Data Implies CIBA Session). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every client $c \in \mathcal{C}$ that is honest in S , every term $lsid \in \mathcal{T}_{\mathcal{N}}$, it holds true that if for any $cibaKey \in \{\text{start_polling, auth_req_id, client_notification_token, binding_message, selected_identity}\}$, we have $cibaKey \in S(c).sessions[lsid]$, then $S(c).sessions[lsid][cibaFlow] = \top$.

PROOF. Initially, the sessions state subterm of c is empty (Definition 7). New sessions are added to a client's sessions state subterm only in Line 10 of Algorithm 2 and Line 40 of Algorithm 2.

In the latter case, we have the value of key `cibaFlow` being \top , which the client never changes once set (Lemma 20), i.e., nothing further to prove.

In the former case, we have to look at all places where one of the dictionary keys `start_polling`, `auth_req_id`, `client_notification_token`, `binding_message`, or `selected_identity` may be added to a session record and prove that such a session record must have a key `cibaFlow` with value \top .

Line 40 of Algorithm 2 Here, key `cibaFlow` is assigned the value \top .

Lines 124 and 126 of Algorithm 3 These lines can only be reached when the client is processing an HTTPS response as part of Algorithm 3 (Line 122 of Algorithm 3), where the `responseTo` key of the *reference* parameter of Algorithm 3 has the value `CIBA_AUTH_REQ`. By applying Lemma 3, we get that the client must have sent a corresponding request by calling `HTTPS_SIMPLE_SEND` with a matching *reference* value. This only happens in Line 70 of Algorithm 8, where the value of $reference[session]$ is set to a session where key `cibaFlow` has been assigned the value \top (see the check in Line 69 of Algorithm 8). Since the same $reference[session]$ is used when processing the response, i.e., in Lines 124 and 126 of Algorithm 3 (see Lines 32f. of Algorithm 3), and a client never changes this value (Lemma 20), we conclude that the value of key `cibaFlow` of the session in Lines 124 and 126 of Algorithm 3 must be \top .

Line 43 of Algorithm 8 (stored to a session in Line 54 of Algorithm 8) See Line 42 of Algorithm 8 – this line is only reachable if key `cibaFlow` has been assigned the value \top .

Line 45 of Algorithm 8 (stored to a session in Line 54 of Algorithm 8) See Line 42 of Algorithm 8 – this line is only reachable if key `cibaFlow` has been assigned the value \top . ■

Lemma 28 (AS Record Subject Uniqueness for Access Token). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every authorization server $as \in \mathcal{AS}$ that is honest in S , every values $i, j \in \mathbb{N}$, it holds true that if $S(as).records.i.2[access_token] \equiv S(as).records.j.2[access_token]$, then $S(as).records.i.2[subject] \equiv S(as).records.j.2[subject]$.

PROOF. Initially, the records state subterm of as is empty (Definition 8). The AS adds new records to records only in Line 89 of Algorithm 11 and Line 297 of Algorithm 11, and in both cases, the entry contains no `access_token` entry:

Values stored in Line 89 of Algorithm 11 are set to a record from `authorizationRequests` in Line 85 of Algorithm 11, and extended by `subject`, `issuer`, and `code` values. `authorizationRequests` is initially empty, and new values are only added at the PAR endpoint in Line 142 of Algorithm 11, where the newly created entry has no `access_token` value.

Similarly, values stored in Line 297 of Algorithm 11 are set to a record from `cibaAuthnRequests` in Line 293 of Algorithm 11, and extended by `subject`, `issuer`, and `auth_req_id` values. `cibaAuthnRequests` is initially empty, and new values are only added at the backchannel authentication endpoint in Line 263 of Algorithm 11, where the newly created entry has no `access_token` value.

The AS adds `access_token` entries to existing records only at the token endpoint in Line 203 of Algorithm 11. There, the access token is either created as a structured token in Line 200 of Algorithm 11, or chosen as a fresh nonce in Line 202 of Algorithm 11.

In the case of a structured token (Line 203 of Algorithm 11), if $S(as).records.i.2[access_token] \equiv S(as).records.j.2[access_token]$, it follows that both tokens contain the same sub value (see Line 199 of Algorithm 11), which is taken from the record entry that will be updated in Line 203 of Algorithm 11.

If the token is a fresh nonce (Line 202 of Algorithm 11), then both records are the same, i.e., $i = j$.

Note that after storing the access token, the values of the token and the subject information stored in the record are not changed by the AS.

Lemma 29 (CIBA Authentication Request Id links Binding Message and Login Session Id). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every client $c \in \mathcal{C}$ that is honest in S , every authorization server $as \in \mathcal{AS}$ that is honest in S , every term $authnReqId \in \mathcal{T}_{\mathcal{N}}$, every term $lsid \in \mathcal{T}_{\mathcal{N}}$, and every term $bindingMsg \in \mathcal{T}_{\mathcal{N}}$, if

- (a) $S(c).sessions[lsid][selected_AS] \in \text{dom}(as)$, and
- (b) $S(c).sessions[lsid][auth_req_id] \equiv authnReqId$, and
- (c) $S(as).records[authnReqId][binding_message] \equiv bindingMsg$

then it holds true that $S(c).sessions[lsid][binding_message] \equiv bindingMsg$.

PROOF. An honest client adds `auth_req_id` values to its sessions only in Line 124 of Algorithm 3, i.e., when processing HTTPS responses with the `CIBA_AUTH_REQ` reference value (Line 122 of Algorithm 3). In this case, the client is processing CIBA authentication responses. Let R be the processing step in which this happens. The client sent the corresponding authentication request in Line 70 of Algorithm 8 (as this is the only location where it uses the `CIBA_AUTH_REQ` reference value). Let $P = (S^P, E^P, N^P) \rightarrow (S^{P'}, E^{P'}, N^{P'})$ be the processing step in which the client sends the authentication request. As the client sends a CIBA authentication request, it follows that `lsid` is a session identifier for a CIBA session, i.e., $S^P(c).sessions[lsid][cibaFlow] \equiv \top$ (Line 69 of Algorithm 8).

Let $selectedAS := S^P(c).sessions[lsid][selected_AS]$. The client sends the authentication request to $S^P(c).oauthConfigCache[selectedAS][backchannel_authentication_endpoint] \equiv \langle \text{URL}, S, selectedAS, /backchannel-authn, \langle \rangle, \perp \rangle$ (see Lines 4, 5, 10, 28 and Line 68 of Algorithm 8 and Lemma 21). As the client never changes the `selected_AS` entry of its sessions, it follows that $selectedAS \in \text{dom}(as)$ (Precondition (a)), i.e., the only as can decrypt the HTTPS request.

Let $Q = (S^Q, E^Q, N^Q) \rightarrow (S^{Q'}, E^{Q'}, N^{Q'})$ be the processing step in which the AS processes the request. The AS processes the request at the `/backchannel-authn` endpoint, i.e., in Line 241 of Algorithm 11. Here, the AS chooses `authnReqId` as a fresh nonce (Line 262 of Algorithm 11) and creates the entry $S^{Q'}(as).cibaAuthnRequests[authnReqId]$. This entry is later moved into the `records` state subterm. The binding message is taken from the backchannel authentication request in Line 254 of Algorithm 11, where it is also added to the newly created authorization record entry.

The HTTPS response created by the AS contains `authnReqId` (Line 264 of Algorithm 11). The client processes the response in R and stores the request identifier in the session identified by `lsid` in Line 124 of Algorithm 3.

Lemma 30 (Client Identifier in asAccounts Implies Issuance). For any run ρ of a FAPI web system $\mathcal{F}API$ with a network attacker, every configuration (S, E, N) in ρ , every client $c \in \mathcal{C}$ that is honest in S , every AS $as \in \mathcal{AS}$ that is honest in S , every domain $d \in \text{dom}(as)$, if $S(c).asAccounts[d][client_id] \neq \langle \rangle$, then there exists a processing step prior to (S, E, N) in which $S(c).asAccounts[d][client_id]$ has been issued to c by as (according to Definition 11).

PROOF. Initially, `asAccounts` is empty (see Definition 7), and the client adds entries to it with a `client_id` dictionary key only in Line 23 of Algorithm 3 and Line 49 of Algorithm 3. In the first case, the client already stores an entry for the AS domain d and reuses the same `client_id` value (see Line 12 of Algorithm 3). Thus, there exists a processing step $R = (S^r, E^r, N^r) \rightarrow (S^{r'}, E^{r'}, N^{r'})$ in which the client executes Line 49 of Algorithm 3 and sets $S^{r'}(c).asAccounts[d][client_id]$ to the value stored in S . The client takes this value from the body of an HTTPS response (Line 44 of Algorithm 3). The client sent the corresponding HTTPS POST request in Line 26 of Algorithm 8 of a processing step $P = (S^p, E^p, N^p) \rightarrow (S^{p'}, E^{p'}, N^{p'})$, as this is the only place where the client calls `HTTPS_SIMPLE_SEND` with the `REGISTRATION` reference value (see also Line 40 of Algorithm 3).

The client sent the request to the domain d : As the client stores the client identifier into $S^{r'}(c).asAccounts[d]$, it follows that $S^r(c).sessions[sessionId][selected_AS] \equiv d$, with `sessionId` being the session identifier from the reference value retrieved in Line 32 of Algorithm 3 (see also Lines 33, 34, and 49). The session identifier is the same value that the client used when sending the request (as it is part of the reference value that is used as an input to the `HTTPS_SIMPLE_SEND` function). By applying Lemma 20, it follows that $S^p(c).sessions[sessionId][selected_AS] \equiv d$ (as the `selected_AS` value of a session never changes). The client sends the request to $S^p(c).oauthConfigCache[d][reg_ep].host$ (see Lines 4, 5, 10, 12, and 25 of Algorithm 8), which is equal to d (see Lemma 21).

This request is sent to the `/reg` endpoint (see Lemma 21). Let Q be the processing step in which the AS processes this request in Line 18 of Algorithm 11, where it calls Algorithm 13 (`REGISTER_CLIENT`). There, the AS responds with an HTTPS response with the 201 status code containing a `client_id` value (i.e., the client identifier that the client stores in R). Thus, all conditions of Definition 11 are fulfilled, and in particular, Q is the processing step in which the AS issued the client identifier. ■

B. Authorization Property

In this section, we show that the authorization property from Definition 13 holds.

Lemma 31 (Authorization). For

- every run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of $\mathcal{F}API$ with a network attacker,

- every resource server $rs \in \text{RS}$ that is honest in S^n ,
- every identity $id \in {}^{\langle \rangle} s_0^{rs}.\text{ids}$ with $b = \text{ownerOfID}(id)$ being an honest browser in S^n ,
- every processing step in ρ

$$Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \rightarrow E_{\text{out}}^Q]{e_{\text{in}}^Q \rightarrow rs} (S^{Q'}, E^{Q'}, N^{Q'})$$

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

it holds true that:

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

- (I) There exists a processing step

$$P = (S^P, E^P, N^P) \xrightarrow[rs \rightarrow E_{\text{out}}^P]{e_{\text{in}}^P \rightarrow rs} (S^{P'}, E^{P'}, N^{P'})$$

such that

- 1) either $P = Q$ or P prior to Q in ρ , and
 - 2) e_{in}^P is an event $\langle x, y, \text{enc}_a(\langle m_{\text{req}}, k_1 \rangle, k_2) \rangle$ 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., $t \equiv m_{\text{req}}.\text{headers}[\text{Authorization}].2$, and
 - 3) r is a fresh nonce generated in P at the resource endpoint of rs in Line 48 of Algorithm 18.
- (II) t is bound to a key $k \in \mathcal{T}_{\mathcal{N}}$, as , a client identifier $\text{clientId} \in \mathcal{T}_{\mathcal{N}}$ and id in S^Q (see Definition 10).
- (III) If there exists a client $c \in \mathbb{C}$ such that clientId has been issued to c by as in a previous processing step (see Definition 11), and if c is honest in S^n , then r is not derivable from the attackers knowledge in S^n (i.e., $r \notin d_\emptyset(S^n(\text{attacker}))$).

PROOF. Resource server sends resource to correct client. The first and the second postcondition are shown in Lemma 18, where we also showed that the message contained in the event e_{in}^P 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_{in}^P : If the resource server sends out the resource in Line 84 of Algorithm 18, then it encrypts the HTTPS response (symmetrically) with the key contained in e_{in}^P . Otherwise, the resource server sends out the response in Line 45 of Algorithm 19, encrypted (symmetrically) with the key contained in e_{in}^P (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 39 (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 39).

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.

The client processes the response of the resource server (containing the resource r) in Line 88 of Algorithm 3 (as a client sends out requests that have an Authorization header and a DPoP header or TLS_binding value in the body only by calling HTTPS_SIMPLE_SEND in Line 43 of Algorithm 6 with the reference RESOURCE_USAGE) in some processing step $R = (S^R, E^R, N^R) \xrightarrow[c \rightarrow E_{\text{out}}^R]{e_{\text{in}}^R \rightarrow c} (S^{R'}, E^{R'}, N^{R'})$ (R happens after Q).

Let sessionId be the session identifier for the session at the client, i.e., the value retrieved in Line 32 of Algorithm 3 when processing the resource response. This is either a session using the authorization code flow, or the CIBA flow, i.e., $S^R(c).\text{sessions}[\text{sessionId}][\text{cibaFlow}]$ is either \top or \perp . As shown in Lemma 20, this value is not changed by the client after initially choosing it.

Case 1: Authorization Code Flow Let $S^R(c).\text{sessions}[\text{sessionId}][\text{cibaFlow}] \equiv \perp$.

Redirection request was created by attacker. The client stores the resource into its sessions in Line 109 of Algorithm 3, but never access it again in any other location. (Note that for CIBA flows, the client would access resources stored into its session in Lines 51ff. of Algorithm 2, however, as for this session, cibaFlow is \perp and as the client never changes cibaFlow as shown in Lemma 20, it follows that the client would stop in Line 57 of Algorithm 2.) The client sends the resource as a response to a request $\text{req}_{\text{redir}}$ stored in $S^R.\text{sessions}[\text{sessionId}][\text{redirectEpRequest}]$, for some value sessionId (and in particular, encrypts the response with the key contained in $\text{req}_{\text{redir}}$), see Line 33, Line 112, and Line 113 of Algorithm 3.

An honest client sets redirectEpRequest values only in the redirection endpoint in Line 12 of Algorithm 2, i.e., $\text{req}_{\text{redir}}$ is a request that was previously received by the client. This request contains a value (an authorization code)

in $req_{\text{redir}}.\text{parameters}[\text{code}]$ (or $\text{extractmsg}(req_{\text{redir}}.\text{parameters}[\text{response}])[\text{code}]$ if the authorization response is signed), which the client puts into the token request in [Algorithm 4](#) (SEND_TOKEN_REQUEST).

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 6](#). In [Line 7](#) of [Algorithm 6](#), 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 6](#) is called only in [Line 83](#) of [Algorithm 3](#), and the value $tokenEPDomain$ is the domain of the token request, i.e., the client received the access token from $\text{authorizationServerOfResource}^{rs}(resourceID)$ (see [Definition 7](#)). This authorization server is honest, as required by the precondition of the lemma.

Attacker can derive authorization code issued for honest client and id. As shown for the second postcondition, the access token that the client received in the token response is bound to some key, the authorization server as , the client id $clientId$, and the identity id .

The authorization server created the access token in the token endpoint in [Line 145](#) of [Algorithm 11](#) in some processing step $T = (S^T, E^T, N^T) \xrightarrow[as \rightarrow E_{\text{out}}^T]{e_{\text{in}}^T \rightarrow as} (S^{T'}, E^{T'}, N^{T'})$. As noted above, the token request contains an authorization code, and also the grant type $authorization_code$ (as the request was sent in [Algorithm 4](#)), i.e., the AS executes [Lines 153ff.](#) of [Algorithm 11](#). The token request contains a code $code$ such that there is a record $rec \in {}^\diamond S^T(as).\text{records}$ with $rec[\text{code}] \equiv code$ and $code \not\equiv \perp$ ([Line 157](#) of [Algorithm 11](#)). Furthermore, the record has the following values:

- $rec[\text{clientId}] \equiv clientId$ (as the access token is bound to this client id),
- $rec[\text{subject}] \equiv id$ (as the access token is bound to this identity)

As shown in [Lemma 7](#), 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 22](#), i.e., such an authorization code cannot leak to the attacker.

Case 2: CIBA Flow Let $S^R(c).\text{sessions}[\text{sessionId}][\text{cibaFlow}] \equiv \top$.

Attacker requested resource at client endpoint Contrary to the previous case, the client model does not send the resource nonce immediately after receiving it. Instead, it also stores the resource nonce into its state in $S^R(c).\text{sessions}[\text{sessionId}][\text{resource}]$ ([Line 109](#) of [Algorithm 3](#)), but waits for the browser to send a request to the `/ciba_get_ssid_or_resource` endpoint. More precisely, for a session with `cibaFlow` being \top , the client model sends values stored under the `resource` key of the session only in [Line 68](#) of [Algorithm 2](#). Here, the client responds to an HTTPS request which includes the session id $sessionId$ in the Cookie header of the request ([Line 52](#) of [Algorithm 2](#)). As we assume that c sends r to the attacker, it follows that this request was created by the attacker, i.e., the attacker can derive $sessionId$ in S^R .

Resource Request sent for sessionId. For storing the resource nonce, the client retrieves the session identifier from a *reference* dictionary ([Line 32](#) of [Algorithm 3](#)), which is an input argument to the `PROCESS_HTTPS_RESPONSE` function ([Algorithm 3](#)). In addition, this dictionary contains the value $reference[\text{responseTo}] \equiv \text{RESOURCE_USAGE}$ ([Line 88](#) of [Algorithm 3](#)). The client sends resource requests only in [Algorithm 6](#) ([Line 43](#) of [Algorithm 6](#) is the only place where the client uses the `RESOURCE_USAGE` reference value). The value `sessionId` stored in *reference* is an input argument of [Algorithm 6](#), i.e., `USE_ACCESS_TOKEN` was called with $sessionId$.

Access token was sent by correct authorization server. Before sending the resource request, the client ensures that it previously sent the token request to the correct authorization server, i.e., the authorization server managing the resource: In [Line 7](#) of [Algorithm 6](#), 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 6](#) is called only in [Line 83](#) of [Algorithm 3](#), and the value $tokenEPDomain$ is the domain of the token request, i.e., the client received the access token from $\text{authorizationServerOfResource}^{rs}(resourceID)$ (see [Definition 7](#)). This authorization server is honest, as required by the precondition of the lemma.

Processing of Token Response. The client calls `USE_ACCESS_TOKEN` in [Line 83](#) of [Algorithm 3](#) when processing the token response. The session identifier (i.e., the input argument of `USE_ACCESS_TOKEN`) is taken from another dictionary, which we call *reference'*. *reference'* is an input argument of [Algorithm 3](#) (`PROCESS_HTTPS_RESPONSE`).

As the client calls [Line 83](#) of [Algorithm 3](#), it follows that $reference'[\text{responseTo}] \equiv \text{TOKEN}$ ([Line 78](#) of [Algorithm 3](#)), i.e., the client is processing the token response and uses the same session id when sending the token request.

Token Request. Let $T = (S^T, E^T, N^T) \xrightarrow[c \rightarrow E_{\text{out}}^T]{e_{\text{in}}^T \rightarrow c} (S^{T'}, E^{T'}, N^{T'})$ be the processing step in which the client creates and emits the token request. An honest client sends token requests only in Line 43 of Algorithm 4 and Line 40 of Algorithm 5 (these are the only places where the client sends requests using the TOKEN reference value). We first show that the token request was not created in Algorithm 4. The body of token requests created in Algorithm 4 contains the value `grant_type` set to `authorization_code`, and a `code_verifier`. As shown in Lemma 26, $S^T(c).\text{sessions}[sessionId][code_verifier]$ is empty, as a client never changes the `cibaFlow` value of a session and as $S^R(c).\text{sessions}[sessionId][cibaFlow] \equiv \top$.

Thus, when processing the request at the token endpoint, the authorization server would stop at Line 156 of Algorithm 11 and not issue an access token.

Token requests created in Algorithm 5 contain the `grant_type` `urn:openid:params:grant-type:ciba` and an authentication request identifier `authnReqId` value in the body of the request (Line 6 and Line 39 of Algorithm 5). Note that the client retrieves the request id from the session identified by `sessionId`, i.e., from $S^T(c).\text{sessions}[sessionId][auth_req_id]$ (Line 4 of Algorithm 5).

Let d_t be the domain to which the token request is sent. As noted above, d_t is a domain of as . It holds true that $d_c \equiv S^T(c).\text{session}[lsid][selected_AS]$, and therefore,

$$S^T(c).\text{session}[lsid][selected_AS] \in \text{dom}(as) \quad (6)$$

as the client takes the domain of the request from $S^T(c).\text{oauthConfigCache}[selectedAS][token_ep]$, with $selectedAS \equiv S^T(c).\text{sessions}[lsid][selected_AS]$ (see Lines 2, 3, 10, and Line 11 of Algorithm 5).

Record Entry in AS State. When processing the token request, the AS retrieves a record entry rec from its `records` state subterm such that $rec[auth_req_id] \equiv authnReqId$ (Line 161 and Line 164 of Algorithm 11).

An AS adds entries to its `records` state subterm only in Line 89 of Algorithm 11 and Line 297 of Algorithm 11. However, only the record added in Line 297 of Algorithm 11 contains an `auth_req_id` entry.

The client uses the access token contained in the token response; when processing the token request, the AS adds the access token that it creates and puts in the token response to the record entry (Line 203 of Algorithm 11), i.e., to rec . As shown for the second postcondition, there is a record $rec' \in \langle \rangle S^Q(as).\text{records}$ such that $rec'[\text{access_token}] \equiv t$ and $rec'[\text{subject}] \equiv id$. From Lemma 28, it follows that rec and rec' have the same subject entry, i.e., $rec[\text{subject}] \equiv id$.

An honest AS adds subject values to record entries only in Line 89 of Algorithm 11 (which we ruled out above) and Line 297 of Algorithm 11. Thus, it follows that the AS received an HTTPS request `ciba-auth2-req` at its `/ciba-auth2` endpoint (Line 281 of Algorithm 11) with $ciba\text{-}auth2\text{-}req.\text{body}[\text{identity}] \equiv id$ (see Line 282 and Line 294 of Algorithm 11). This request also contains the password of id , i.e., $ciba\text{-}auth2\text{-}req.\text{body}[\text{password}] \equiv \text{secretOfID}(id)$ (Line 286 of Algorithm 11). Thus, the request must have been created by $\text{ownerOfID}(id)$. In addition, this request contains a reference `auth2Reference` to the record entry, i.e., $auth2Reference \equiv ciba\text{-}auth2\text{-}req.\text{body}[\text{ciba_auth2_reference}]$ (Line 288 of Algorithm 11) such that $rec[\text{ciba_auth2_reference}] \equiv auth2Reference$ (Line 289 of Algorithm 11; note that this entry is taken from the `cibaAuthnRequests` state subterm and added to the `records`, see Line 297 of Algorithm 11).

The AS model adds `ciba_auth2_reference` values to `cibaAuthnRequests` only in Line 277 of Algorithm 11, where it is chosen as a fresh nonce. In this endpoint, the AS creates an HTTPS response referencing the `script_as_ciba_form` script, including the `auth2Reference` value Line 278 of Algorithm 11. In addition, it includes a binding message `bindingMsg` retrieved from the same `cibaAuthnRequests` entry (Line 274 of Algorithm 11), and a domain d_c of a client. Both the binding message, and the client id of the client used for determining the domain are taken from the entry stored in `cibaAuthnRequests`. The AS adds such entries only at the `backchannel-authn` endpoint, and the values are taken from the request (Line 254 of Algorithm 11). This endpoint requires client authentication (Line 242 of Algorithm 11), i.e., the client with the client id $rec[\text{client_id}]$ created the request.

Note that the binding message `bindingMsg` is stored rec , i.e., in the record entry at the AS identified by `authnReqId`. From Equation 6 and Lemma 20, it follows that $S(c).\text{sessions}[sessionId][selected_AS] \in \text{dom}(as)$. From Lemma 29, it follow that $S(c).\text{sessions}[sessionId][binding_message] \equiv bindingMsg$.

Browser stores client domain and binding message in state The browser accesses user credentials for a domain of the AS only when processing a script loaded from the AS. As the response contains the `ciba_auth2_reference` dictionary string, it follows that the browser processed the `script_as_ciba_form` script (Algorithm 17). As the browser sends the Post request, we conclude that it processes the CIBAFORM command successfully with the binding message `bindingMsg` and the client domain d_c . The browser processes this command in Line 52 of Algorithm 28 and checks whether the `cibaBindingMessages` contains a domain and a binding message. These values are taken

from the HTTPS response that includes the script and was sent by the AS, i.e., the domain is d_c and the binding message is $bindingMsg$.

Thus, we conclude that $\langle d_c, bindingMsg \rangle \in^{\langle \rangle} S(b).cibaBindingMessages$. As noted previously, it holds true that $S(c).sessions[sessionId][binding_message] \equiv bindingMsg$. However, this contradicts [Lemma 25](#), i.e., the attacker cannot derive $sessionId$. ■

C. Authentication Property

In this section, we show that the authentication property from [Definition 15](#) holds. This will be a proof by contradiction, i.e., we assume that there is a FAPI web system $\mathcal{F}API$ in which the authentication property is violated and deduce a contradiction.

Assumption 2. There exists a FAPI web system with a network attacker $\mathcal{F}API$ such that there exists a run ρ of $\mathcal{F}API$ with a configuration (S, E, N) in ρ , some $c \in \mathcal{C}$ that is honest in S , some identity $id \in \text{ID}$ with $as = \text{governor}(id)$ being an honest AS and $b = \text{secretOfID}(id)$ 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., $n \in d_\emptyset(S(\text{attacker}))$).

Lemma 32 (Authentication Property Holds). [Assumption 2](#) is a contradiction.

PROOF. By [Assumption 2](#), there is a service session identified by n for id at c , and hence, by [Definition 14](#), we have that there is a session id x and a domain $d \in \text{dom}(\text{governor}(id)) = \text{dom}(as)$ with $S(c).sessions[x][loggedInAs] \equiv \langle d, id \rangle$ and $S(c).sessions[x][serviceSessionId] \equiv n$. [Assumption 2](#) says that n is derivable from the attacker’s knowledge. Since we have $S(c).sessions[x][serviceSessionId] \equiv n$, 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 14](#) of [Algorithm 7](#). 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 $S(c).sessions[x][loggedInAs]$, in this case $\langle d, id \rangle$ (this is the only place where the client sets the `loggedInAs` value).

CHECK_ID_TOKEN, in turn, is only called in a single place: When processing an HTTPS response to a token request, in [Line 87](#) of [Algorithm 3](#). From the check in [Line 85](#) of [Algorithm 3](#), we know that this response came from (one of) as ’s token endpoints: From [Lines 2, 3, and 13](#) of [Algorithm 7](#), it follows that $S(c).sessions[x][selected_AS] \equiv d$. In [Line 85](#) of [Algorithm 3](#), the client checks whether the host of the corresponding token request is equal to $S(c).oauthConfigCache[selectedAS][token_ep].host$, with $selectedAS \equiv S(c).sessions[x][selected_AS]$ (see [Lines 32-34](#) and [Line 84](#) of [Algorithm 3](#)). As shown in [Lemma 21](#), $S(c).oauthConfigCache[selectedAS][token_ep].host \equiv selectedAS$, i.e., the client sent the token request to as . Let req_{token} be the token request, and $R = (S^r, E^r, N^r) \rightarrow (S^{r'}, E^{r'}, N^{r'})$ the processing step in which the client emits the token request.

In the following, we show that the client identifier in the token request has been issued by as to c in a previous processing step (according to [Definition 11](#)): The client sends token requests only in [Algorithm 4](#) and [Algorithm 5](#). In both cases, it sets the client identifier value in the request, i.e., $req_{token}.body[client_id]$ (if the client uses mTLS client authentication) or $\text{extractmsg}(req_{token}.body[client_assertion])[iss]$ (if the client uses `private_key_jw` client authentication) to the value $S^r(c).asAccounts[selectedAS][client_id]$ with the same value $selectedAS$ as before, i.e., the domain of the AS to which the client sends the token request (see [Line 10](#) of [Algorithm 4](#) and [Line 7](#) of [Algorithm 5](#)). Let $clientId$ be this client identifier. From [Lemma 30](#), it follows that this client identifier value was issued by as to c in a previous processing step (note that $clientId \neq \langle \rangle$, as otherwise, the AS would not send a response).

Since as is an honest authorization server, it will only reply to a token request if that request contains a valid authorization code or a valid authentication request identifier (see the two cases in [Lines 152ff.](#) of [Algorithm 11](#)). We distinguish these cases now.

Case 1: Token Request contains Authorization Code: If $req_{token}.body[grant_type] \equiv authorization_code$, then the token request contains a `code` such that there is a record $rec \in^{\langle \rangle} S(as).records$ with $rec[code] \equiv code$, $rec[client_id] \equiv clientId$, and $rec[subject] \equiv id$.

The client sends token requests with the `authorization_code` grant type with an authorization code only in [Algorithm 4](#) (SEND_TOKEN_REQUEST). The corresponding session at the client contains a value `code_verifier` (note that the AS checks this value in [Line 156](#) of [Algorithm 11](#)). By applying [Lemma 26](#), we conclude that the session at the client does not contain a `cibaFlow` value being \top . As shown in [Lemma 20](#), each client session always contains a `cibaFlow` value, and this value is never changed by the client. The `cibaFlow` value is either \top or \perp , see [Line 10](#) of [Algorithm 2](#) and [Line 40](#) of [Algorithm 2](#). Thus, we conclude that for the session for which the client creates req_{token} , the value of `cibaFlow` is \perp .

As the `cibaFlow` value of this session is \perp , it follows that the client sends the service session id n only in the response in Line 19 of Algorithm 7. (The only other place where the client accesses and sends out this value is in Lines 51ff. of Algorithm 2, however, this happens only if `cibaFlow` $\neq \perp$, see Line 57 of Algorithm 2).

By tracking backwards from Line 14 of Algorithm 7, 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 7 must have sent an HTTPS request req to c containing the aforementioned `code` (see also Lemma 7).

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 (the client does not send messages to itself, and messages with an authorization code sent by the AS are sent as HTTPS responses). 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 7). 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 2.

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 22.

Case 2: Token Request contains Authentication Request Identifier: In this case, $req_{token}.body[grant_type] \equiv \text{urn:openid:params:grant-type:ciba}$ and $auth_req_id \in req_{token}.body$ (see Line 160 and Line 163 of Algorithm 11). The client creates requests with these values in the body only in Line 40 of Algorithm 5 (SEND_CIBA_TOKEN_REQUEST). The client calls this function only in the following places:

- Line 50 of Algorithm 2: In this case, the corresponding session contains a value `client_notification_token` (see Line 49 of Algorithm 2). The client sets this notification token value only in Line 45 of Algorithm 8, i.e., only if the `cibaFlow` value of the corresponding session has the value \top (Line 42 of Algorithm 8).
- Line 59 of Algorithm 9: In this case, the corresponding session has the value `start_polling` (Line 58 of Algorithm 9). As shown in Lemma 27, the `cibaFlow` value of the session is \top .

As shown in Lemma 20, the client never changes the `cibaFlow` value of a session.

Record Entry in AS State. When processing the token request, the AS retrieves a record entry rec from its `records` state subterm such that $rec[auth_req_id] \equiv authnReqId$ (Line 161 and Line 164 of Algorithm 11).

An AS adds entries to its `records` state subterm only in Line 89 of Algorithm 11 and Line 297 of Algorithm 11. However, only the record added in Line 297 of Algorithm 11 contains an `auth_req_id` entry.

For the login, the client uses the ID token contained in the token response. The AS creates the ID token in Lines 206ff. of Algorithm 11. It sets the sub value of the ID token to id (Line 208 of Algorithm 11), i.e., $rec[subject] \equiv id$ (as this identity is logged in at the client when processing the token response.)

An honest AS adds `subject` values to record entries only in Line 89 of Algorithm 11 and Line 297 of Algorithm 11. However, as this particular record contains an `auth_req_id` value, it must have been created in Line 297 of Algorithm 11. Thus, it follows that the AS received an HTTPS request $req_{ciba-auth2}$ at its `/ciba-auth2` endpoint (Line 281 of Algorithm 11) with $req_{ciba-auth2}.body[identity] \equiv id$ (see Line 282 and Line 294 of Algorithm 11). This request also contains the password of id , i.e., $req_{ciba-auth2}.body[password] \equiv \text{secretOfID}(id)$ (Line 286 of Algorithm 11). Thus, the request must have been created by `ownerOfID(id)`. In addition, this request contains a reference `auth2Reference` to the record entry, i.e., $auth2Reference \equiv req_{ciba-auth2}.body[ciba_auth2_reference]$ (Line 288 of Algorithm 11) such that $rec[ciba_auth2_reference] \equiv auth2Reference$ (Line 289 of Algorithm 11; note that this entry is taken from the `cibaAuthnRequests` state subterm and added to the `records`, see Line 297 of Algorithm 11).

The AS model adds `ciba_auth2_reference` values to `cibaAuthnRequests` only in Line 277 of Algorithm 11, where it is chosen as a fresh nonce. In this endpoint, the AS creates an HTTPS response referencing the `script_as_ciba_form` script, including the `auth2Reference` value in Line 278 of Algorithm 11. In addition, it includes a binding message `bindingMsg` and a domain d_c of a client. Both the binding message, and the client id of the client used for determining the domain are taken from the entry stored in `cibaAuthnRequests` (see Lines 274f. of Algorithm 11). The AS adds such binding message and client identifier values to entries of the `cibaAuthnRequests` state subterm only at the `backchannel-authn` endpoint, and the values are taken from the corresponding request (see Lines 246, 252, and Line 254 of Algorithm 11). This endpoint requires client authentication (Line 242 of Algorithm 11), i.e., the client with the client id $rec[client_id]$ created the request, see Lemma 13. (Note that an honest AS never changes the `client_id` value of an existing `cibaAuthnRequests` or `records` entry).

Note that the binding message `bindingMsg` is taken from the request to `backchannel-authn` and then stored into the `cibaAuthnRequests` entry (Line 254 of Algorithm 11). This is the same value stored in $rec[binding_message]$, i.e., the record entry at the AS identified by $authnReqId$.

Binding Message Stored at Client As shown previously, it holds true that $S(c).sessions[x][selected_AS] \in \text{dom}(as)$. It also holds true that $S(c).sessions[x][auth_req_id] \equiv authnReqId$ (as this is the authentication request identifier that the client used for the token request). Furthermore, when responding to the request to the `/ciba-auth` endpoint, the AS includes the binding message stored in the record identified by `authnReqId`. For a given record entry, the AS does not change the binding message value, i.e., $S(as).records[authnReqId][binding_message] \equiv bindingMsg$. From Lemma 29, it follows that $S(c).sessions[x][binding_message] \equiv bindingMsg$.

Browser stores client domain and binding message in state The browser accesses user credentials for a domain of the AS only when processing a script loaded from the AS. As the response contains the `ciba_auth2_reference` dictionary string, it follows that the browser processed the `script_as_ciba_form` script (Algorithm 17). As the browser sends the POST request, we conclude that it processes the CIBAFORM command successfully with the binding message $bindingMsg$ and the client domain d_c . The browser processes this command in Line 52 of Algorithm 28 and checks whether the `cibaBindingMessages` contains a domain and a binding message. These values are taken from the HTTPS response that includes the script and was sent by the AS, i.e., the domain is d_c and the binding message is $bindingMsg$. Thus, we conclude that $\langle d_c, bindingMsg \rangle \in S(b).cibaBindingMessages$ (note that the browser does not remove or modify existing `cibaBindingMessages` values). As noted previously, it holds true that $S(c).sessions[x][binding_message] \equiv bindingMsg$. From Lemma 25, it follows that the attacker cannot derive the session identifier x .

Process Requesting Service Session ID can derive x : The client sends the service session id stored in Line 14 of Algorithm 7 only in Line 68 of Algorithm 2 (the only other place is Line 19 of Algorithm 7, which we can rule out as the `cibaFlow` value of the session is \top). For responding in Line 68 of Algorithm 2, the client expects a request $req_{get-ssid}$ with a session id cookie, and responds with the corresponding service session id. Thus, it follows that $req_{get-ssid}$ contains x as the session id cookie. As shown for the first case, the sender of this request cannot be b , c , or as (as neither of those processes would leak the service session id). Thus, the request must have been created by the attacker. However, this contradicts the fact that the attacker cannot derive x , as shown above. ■

D. Session Integrity for Authentication Property

In this section, we show that the session integrity for authentication properties from Definition 23 and Definition 24 hold.

We start by proving that the property for authorization code flows (Definition 23) holds. This will be a proof by contradiction, hence we begin by assuming the opposite:

Assumption 3. There exists a FAPI web system with a network attacker $\mathcal{F}API$ such that there exists a run ρ of $\mathcal{F}API$ with a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , a browser b honest in S , an authorization server $as \in AS$, an identity id , a client $c \in C$ honest in S , and a nonce $lsid$ s.t. $S(c).sessions[lsid][cibaFlow] \equiv \perp$, with $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$ and c did not leak the authorization request for $lsid$, such that

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

Lemma 33 (Session Integrity for Authentication for Authorization Code Flows Property Holds). Assumption 3 is a contradiction.

PROOF.

(I) We have that $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$. With Definition 16, we know that c sent out a service session id associated with $lsid$ to b (i.e., set a cookie $\langle \text{serviceSessionId}, \langle ssid, \top, \top, \top \rangle \rangle$, and stored $ssid$ in its sessions storage). For a session with $S(c).sessions[lsid][cibaFlow] \equiv \perp$, such a cookie is only set by a client if its `CHECK_ID_TOKEN` function was called with $lsid$ as the first argument – which, in turn, can only happen in Line 87 of Algorithm 3 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 (i.e., with a matching nonce and symmetric key, and with $reference[responseTo] \equiv \text{TOKEN}$). Clients only send such token requests in Line 43 of Algorithm 4, i.e., after calling `SEND_TOKEN_REQUEST` in Line 34 of Algorithm 2, when processing an HTTPS request req_{redir} .

Note: Clients also send token requests in Line 40 of Algorithm 5; however, Algorithm 5 is only called in Line 50 of Algorithm 2 and Line 59 of Algorithm 9, and in both cases, the corresponding session in the client's state must contain CIBA-specific values – `client_notification_token`, and `start_polling`, respectively – and thus, by Lemma 27, $S(c).sessions[lsid][cibaFlow] \equiv \top$, which contradicts this lemma's preconditions.

Hence, we look at how a client can reach Line 34 of Algorithm 2. req_{redir} must contain a cookie $\langle \langle _Host, sessionId \rangle: lsid \rangle$ (Line 13 of Algorithm 2), and $lsid$ is used as session id to store req_{redir} in the client's

session storage in Line 33 of Algorithm 2 under the key `redirectEpRequest` (this is also the only place where a client stores something under this key).

When executing `CHECK_ID_TOKEN` (during the Q from Definition 16), the message (HTTP response) with the aforementioned service session id cookie is sent to and encrypted for the sender of req_{redir} , because c looks these values up in the login session record stored in $S(c).sessions[lsid]$ under the key `redirectEpRequest`. Hence, the sender of req_{redir} , i.e., b by Definition 16, must have included the aforementioned cookie with $lsid$ in its request.

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 (see (C) in the proof of Lemma 24). A cookie with the properties shown above is only set in Line 69 of Algorithm 3. 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 (see Line 64 of Algorithm 3). A session entry with key `startRequest` is only ever created in Line 10 of Algorithm 2. 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 2).

Due to the origin check and the POST method, this request must have been sent by a script (POST) under one of c 's origins (origin check). 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 10 (because that is the only place in which that script issues a POST request).

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 $\langle \text{Set-Cookie}, [\langle _Host, sessionId \rangle: \langle lsid, \top, \top, \top \rangle] \rangle$.

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

- (II) Again, we have 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 $S(c).sessions[lsid][\text{loggedInAs}] \equiv \langle d, id \rangle$, we know – since `CHECK_ID_TOKEN` produced an output – that for $t_c := \text{extractmsg}(t)$, we have $t_c[\text{iss}] \equiv d$, $t_c[\text{sub}] \equiv id$, and $t_c[\text{aud}] \equiv \text{clientId}$ (for some clientId). Due to the check in Line 85 of Algorithm 3, this id token must have been sent by as (because $d \in \text{dom}(as)$). as will only output such a term t if there is a record rec in as 's records state subterm with $rec[\text{subject}] \equiv id$, $rec[\text{client_id}] \equiv \text{clientId}$, and $rec[\text{code_challenge}] \equiv \text{codeChallenge}$ (for some value of codeChallenge).

Note that an AS only creates such id tokens in Lines 207ff. of Algorithm 11, and only after clientId has been issued to c by as (Definition 11).

By construction of c and tracking of $S(c).sessions[lsid]$ in c 's state, it is easy to see that once c reaches `CHECK_ID_TOKEN`, the session storage $S(c).sessions[lsid]$ must contain a key `code_verifier` under which a nonce codeVerifier is stored. We note that $S(b).cookies[d_c]$ must contain a cookie $\langle \langle _Host, sessionId \rangle, \langle lsid, \top, \top, \top \rangle \rangle$ for $d_c \in \text{dom}(c)$, because b sends a cookie $[\langle _Host, sessionId \rangle: lsid]$ 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 $\langle _Host, sessionId \rangle$, it will do so with the attributes set as shown here.

Hence, we can apply Lemma 24 (note that $S(c).sessions[lsid][\text{loggedInAs}] \equiv \langle d, id \rangle$ with $d \in \text{dom}(as)$ implies $S(c).sessions[lsid][\text{selected_AS}] \equiv d \in \text{dom}(as)$). I.e., we now have that there is exactly one nonce requestUri such that $S(as).authorizationRequests[\text{requestUri}][\text{code_challenge}] \equiv \text{hash}(\text{codeVerifier})$, and only b , c , and as know requestUri .

We know from Line 158 of Algorithm 11 that the token request which leads to as issuing t must contain a code verifier such that $\text{hash}(\text{codeVerifier}) \equiv \text{rec}[\text{code_challenge}]$ (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` (see proof for (I)). There, c selects the value for the code verifier 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 (the request of b that triggered the token request).

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 76ff. of Algorithm 11). 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 $\text{authZrec}[\text{auth2_reference}] \equiv \text{auth2Reference}$ 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 24, we established that there is exactly one requestUri in as 's state such that $S(as).authorizationRequests[\text{requestUri}][\text{code_challenge}] \equiv \text{hash}(\text{codeVerifier})$. Therefore, $\text{reqUri} \equiv \text{requestUri}$. 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 POST request to $/auth2$ as well. As b is honest, this can only happen through a script – together with the origin header check in Line 76 of Algorithm 11, and $script_as_form$ (Algorithm 16) being the only script ever sent by as which can send requests to the $/auth2$ endpoint, 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 $rec[subject] \equiv id$ 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 $auth2Reference$.

Therefore, we have authenticated $_{\rho}^{Q''}(b, c, id, as, lsid)$ which – together with (I) from above – contradicts Assumption 3, therefore proving the lemma. ■

This leaves us with the property for FAPI-CIBA flows (Definition 24). This will be a proof by contradiction, hence we begin by assuming the opposite:

Assumption 4. There exists a FAPI web system with a network attacker $\mathcal{F}API$ such that there exists a run ρ of $\mathcal{F}API$ with a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , a browser b honest in S and behaves according to Assumption 1, an authorization server $as \in AS$, an identity id , a client $c \in C$ honest in S , and a nonce $lsid$ s.t. $S(c).sessions[lsid][cibaFlow] \equiv \top$, and $loggedIn_{\rho}^Q(b, c, id, as, lsid)$ such that

- (I) there is no processing step Q' prior to Q in ρ such that $startedCIBA_{\rho}^{Q'}(b, c, lsid)$, or
- (II) as is honest in S , and there is no processing step Q'' prior to Q in ρ such that $authenticatedCIBA_{\rho}^{Q''}(b, c, id, as, lsid)$.

Lemma 34 (Session Integrity for Authentication for FAPI-CIBA Flows Property Holds). Assumption 4 is a contradiction.

PROOF. We start with some helpful intermediate results:

- (A) From $loggedIn_{\rho}^Q(b, c, id, as, lsid)$, we know that during Q , c emits an event which contains an HTTPS response with a header $\langle \text{Set-Cookie}, [serviceSessionId: \langle ssid, \top, \top, \top \rangle] \rangle$ (for some nonce $ssid$). Such a cookie (with name $serviceSessionId$) is only included in a client's output in Line 17 of Algorithm 7, and in Lines 51ff. of Algorithm 2. In the former case, this only happens if $S(c).sessions[lsid][cibaFlow] \equiv \perp$ (Line 15 of Algorithm 7) – which contradicts this lemma's preconditions. I.e., **the event emitted by c during Q must originate from Lines 51ff. of Algorithm 2.**

This in turn implies:

- (A.i) During Q , c processed an HTTPS request sent by b which contained a cookie with name $\langle _Host, sessionId \rangle$ and value $lsid$: with a different value, the response which c emits during Q would not contain $S(c).sessions[lsid][serviceSessionId]$ – note that due to Line 14 of Algorithm 7 being the only place where c writes a $serviceSessionId$ into one of its sessions, and the value is a fresh nonce, there is no $lsid' \neq lsid$ such that $S(c).sessions[lsid][serviceSessionId] \equiv S(c).sessions[lsid'][serviceSessionId]$.
- (A.ii) $serviceSessionId \in S(c).sessions[lsid]$: otherwise, c would not set a $serviceSessionId$ header.

(B)

- (B.i) From (A), we know that b must have sent an HTTPS request to c with a cookie with name $\langle _Host, sessionId \rangle$ and value $lsid$. Since $S^0(b).cookies \equiv \langle \rangle$ (Definition 3), that cookie must have been stored in b 's state in some processing step. However, since said cookie has the $_Host$ prefix, it must have been set by c (see (C) in the proof of Lemma 24). Hence, there must have been an HTTPS response sent from c to b (i.e., encrypted for b , i.e., with a key used by b in a previous HTTPS request to c) which contained a corresponding Set-Cookie header.
- (B.ii) Clients only include a Set-Cookie header with a cookie named $\langle _Host, sessionId \rangle$ in two places: when handling requests to the $/start_ciba$ endpoint (Lines 35ff. of Algorithm 2), and when processing a PAR response (Lines 58ff. of Algorithm 3).

With Lemma 3, we know that Lines 58ff. of Algorithm 3 can only be executed if the client previously called $HTTPS_SIMPLE_SEND$ with a $reference$ value such that $reference[responseTo] \equiv PAR$. This, in turn, only happens in Line 72 of Algorithm 8 – however, due to the check in Line 69 of Algorithm 8, this can only happen if $S(c).sessions[lsid][cibaFlow] \equiv \perp$, which contradicts this lemma's preconditions (note that the value for $cibaFlow$ never changes, see Lemma 20).

Therefore, the $\langle _Host, sessionId \rangle$ cookie must originate from the client's $/start_ciba$ endpoint in Lines 35ff. of Algorithm 2 (in some processing step R prior to Q).

- (C) Since the request to c 's $/start_ciba$ endpoint (see (B)), in response to which c set the $sessionId$ cookie, was sent by b (otherwise, b would not have used that cookie in Q), we can use Assumption 1 to conclude $b = ownerOfID(S(c).sessions[lsid][selected_identity])$ (see Lines 37 and 40 of Algorithm 2).

(D) Given the above, we can now prove that our assumption is a contradiction, starting with (I): Recall (B) and the processing step R , in which c emits an HTTPS response to a request from b , and in that response, sets the `sessionId` cookie. There, the value for said cookie is a fresh nonce (Line 38 of Algorithm 2), and in order to even reach that line, c must be processing an HTTPS request for a domain of c (Lines 7f. of Algorithm 39, Definition 7, and Appendix A-C: if the request were not for a domain of c , c would not be able to decrypt it or would stop in Line 8 of Algorithm 39). Furthermore, that request must be for path `/start_ciba`, and since b was able to decrypt the response (and thus learn $lsid$), the request must have been created by b in some processing step Q' prior to R .

In summary, during Q' , browser b emits an HTTPS request to a domain of c , with path `/start_ciba`, and – during some later processing step R , client c processes this request and emits an event with an HTTPS response to b 's request with a Set-Cookie header with a cookie named $\langle_Host, sessionId\rangle$ with value $lsid$.

Hence, we conclude $startedCIBA_{\rho}^{Q'}(b, c, lsid)$.

(E) We can now focus on (II):

- (E.i) From (A.ii) we have $serviceSessionId \in S(c).sessions[lsid]$. Values under `serviceSessionId` are only added to a session in the client's state in Line 14 of Algorithm 7, where a fresh nonce is stored there. Let P be the processing step in which $S(c).sessions[lsid][serviceSessionId]$ was set to the value $ssid$ used in Q (i.e., in `loggedIn`, see Definition 16). I.e., c must have reached Line 14 of Algorithm 7 during P , and – since we have $S(c).sessions[lsid][cibaFlow] \equiv \top$ and Lemma 20 – P must have reached the **stop** in Line 21 of Algorithm 7.
- (E.ii) Due to Line 13 of Algorithm 7 and $loggedIn_{\rho}^Q(b, c, id, as, lsid)$ (see Definition 16), the id token processed during P must be a (signed, see Line 6 of Algorithm 7, we're looking at the extracted value here) dictionary $idToken$ with $idToken[sub] \equiv id$.
- (E.iii) Line 14 of Algorithm 7 from (E.i) can only be reached if the client calls `CHECK_ID_TOKEN` (Algorithm 7) during P . Since `CHECK_ID_TOKEN` is only called in Line 87 of Algorithm 3, c must have processed an HTTPS response to a token request (i.e., $reference[responseTo] \equiv \text{TOKEN}$) during P . Due to the check in Line 85 of Algorithm 3, Lemma 21, the origin of $selectedAS$ from the client's session storage used in Line 13 of Algorithm 7, and precondition $loggedIn_{\rho}^Q(b, c, id, as, lsid)$ (see Definition 16), we conclude that the token request to which c processes a response in P , was sent to and encrypted for as . Lemma 46 thus gives us that the token response processed by c during P must have been created by as .
- (E.iv) With Lemma 3, we know that for c to process an HTTPS response with $reference[responseTo] \equiv \text{TOKEN}$ during P , there must have been some previous processing step $P_c^{\text{tokReq}} = (S^{\text{tokReq}}, E^{\text{tokReq}}, N^{\text{tokReq}}) \rightarrow (S^{\text{tokReq}'}, E^{\text{tokReq}'}, N^{\text{tokReq}'})$ during which c called `HTTPS_SIMPLE_SEND` with such a reference value. This only happens in two places: Line 43 of Algorithm 4, and Line 40 of Algorithm 5.
- (E.v) P_c^{tokReq} must have ended in Line 40 of Algorithm 5. Proof by contradiction, assume P_c^{tokReq} ended in Line 43 of Algorithm 4: the check in Line 3 of Algorithm 4 must have succeeded, i.e., $code_verifier \in S^{\text{tokReq}}(c).sessions[lsid]$. However, we have $S(c).sessions[lsid][cibaFlow] \equiv \top$, which together with Lemma 20 gives us $S^{\text{tokReq}}(c).sessions[lsid][cibaFlow] \equiv \top$ – which is a contradiction to Lemma 26.
- (E.vi) The body of the token request produced by c in P_c^{tokReq} contains a key `grant_type` with value `urn:openid:params:grant-type:ciba`, and a key `auth_req_id` with value $authnReqId := S^{\text{tokReq}}(c).sessions[lsid][auth_req_id]$. Note: The session id used in P_c^{tokReq} , i.e., $lsid$, is the same as in $loggedIn_{\rho}^Q(b, c, id, as, lsid)$: it is passed from P_c^{tokReq} to P as part of the *reference*, see Lemma 3; during P , that session id is the key under which the (fresh) service session id (which is then used in Q) is stored, see (E.i).
- (E.vii) From (E.iii), we know that as must have created the HTTPS (token) response processed by c in P in some prior processing step $P_{as}^{\text{tokRes}} = (S^{\text{tokRes}}, E^{\text{tokRes}}, N^{\text{tokRes}}) \rightarrow (S^{\text{tokRes}'}, E^{\text{tokRes}'}, N^{\text{tokRes}'})$. Furthermore, the id token in that response must contain id , stored under key `sub` (see (E.ii)). An AS only produces id tokens (more formally: emits events with an HTTPS response, whose body contains a key `id_token`) in Line 215 of Algorithm 11, i.e., the token endpoint. Hence, there must be a record rec in $S^{\text{tokRes}}(as).records$ such that $rec[subject] \equiv id$ (see Lines 164 and 208 of Algorithm 11).
- (E.viii) Such records are only added to as ' state in two places: at the `/auth2` endpoint in Lines 86 and 89 of Algorithm 11, and at the `/ciba-auth2` endpoint in Lines 294 and 297 of Algorithm 11.
- From (E.vi), we know that the token request produced by c in P_c^{tokReq} contains a key `grant_type` with value `urn:openid:params:grant-type:ciba`, and an `auth_req_id` in its body. Hence, as must have executed Lines 160ff. of Algorithm 11 during P_{as}^{tokRes} , and the execution must have reached Line 215 of Algorithm 11 (otherwise, there would be no token response for c to process, which contradicts (E.iii)). This implies that the record rec contains a key `auth_req_id`, and $rec[auth_req_id] \notin \{\langle \rangle, \perp\}$ (Lines 163 and 164 of Algorithm 11. Furthermore, $rec[auth_req_id]$ must be equivalent to the $authnReqId$ in the token request from (E.vi).

An AS only adds a value $\neq \perp$ to a record in its `records` state subterm in Line 297 of Algorithm 11, i.e., the `/ciba-auth2` endpoint. Hence, rec must have been created by as while processing a request to its `/ciba-auth2` endpoint in a processing step $P_{as}^{\text{auth2}} = (S^{\text{auth2}}, E^{\text{auth2}}, N^{\text{auth2}}) \rightarrow (S^{\text{auth2}'}, E^{\text{auth2}'}, N^{\text{auth2}'})$ prior to P_{as}^{tokRes} .

(E.ix) We now look at P_{as}^{auth2} in detail:

- (E.ix.1) The request processed by as during P_{as}^{auth2} contains (in its body) the identity id' (under key `identity`) and password $password$ (under key `password`), such that $password \equiv \text{secretOfID}(id')$ (Line 286 of Algorithm 11).
- (E.ix.2) The identity id' must be equivalent to the identity stored in as ' CIBA AuthN Requests storage, i.e., $S^{\text{auth2}}(as).\text{cibaAuthnRequests}[\text{authnReqId}'][\text{selected_identity}] \equiv id'$ (Line 290 of Algorithm 11).
- (E.ix.3) The contents of $S^{\text{auth2}}(as).\text{cibaAuthnRequests}[\text{authnReqId}']$, after adding a few key-value pairs in Lines 294ff. of Algorithm 11, are stored in as ' `records` state subterm (Line 297 of Algorithm 11). Said additional key-value pairs are: $\langle \text{subject}, id' \rangle$, and $\langle \text{issuer}, d_{as} \rangle$ (for a $d_{as} \in \text{dom}(as)$), and $\langle \text{auth_req_id}, \text{authnReqId}' \rangle$. Since we are looking at the record rec from (E.vii) and (E.viii), we must have $\text{authnReqId}' \equiv \text{authnReqId}$ and $id \equiv id'$.
- (E.ix.4) An AS only adds records to its `cibaAuthnRequests` state subterm when processing an HTTPS request to its `/backchannel-authn` endpoint in Line 263 of Algorithm 11. There, the key under which a new record is stored (i.e., authnReqId) is a fresh nonce. Furthermore, the value for the `selected_identity` key of the record is taken from the processed request's body (under key `login_hint`, see Line 255 of Algorithm 11). Let $P_{as}^{\text{bcAuthN}} = (S^{\text{bcAuthN}}, E^{\text{bcAuthN}}, N^{\text{bcAuthN}}) \rightarrow (S^{\text{bcAuthN}'}, E^{\text{bcAuthN}'}, N^{\text{bcAuthN}'})$ be the processing step in which as creates the record stored under authnReqId in its `cibaAuthnRequests` state subterm.
- (E.ix.5) The request to the `/backchannel-authn` endpoint processed in P_{as}^{bcAuthN} must have been created by c : the token request processed by as in P_{as}^{tokRes} must contain client authentication for the same client (Line 165 of Algorithm 11) as the `/backchannel-authn` request (rec from above is a copy of the record created in Line 252 of Algorithm 11 and the `client_id` value is never changed), c created the token request (see (E.iii)), and because c and as are honest, we can apply Lemma 30 and Lemma 13 (the preconditions to the latter lemma follow from the processing of a request to the `/backchannel-authn` endpoint).
- (E.ix.6) Since c used authnReqId in its token request, and by (E.vi), we have $\text{authnReqId} = S^{\text{tokReq}}(c).\text{sessions}[\text{lsid}][\text{auth_req_id}]$, the `login_hint` in c 's request to the `/backchannel-authn` endpoint (which as processes in P_{as}^{bcAuthN}) must be from the same session, i.e., $S^{\text{tokReq}}(c).\text{sessions}[\text{lsid}][\text{selected_identity}]$ (note that the client never overwrites this value after initially setting it).

Overall, this gives us

$$\begin{aligned}
 S(c).\text{sessions}[\text{lsid}][\text{selected_identity}] &\equiv && \text{(E.ix)} \\
 S^{\text{auth2}}(as).\text{cibaAuthnRequests}[\text{authnReqId}][\text{selected_identity}] &\equiv && \text{Line 290 of Algorithm 11} \\
 id' &\equiv && \text{(E.ix.3)} \\
 rec[\text{subject}] &\equiv && \text{(E.vii)} \\
 id &&&
 \end{aligned}$$

However, from (C), we have that b is the owner of identity $S(c).\text{sessions}[\text{lsid}][\text{selected_identity}]$, and hence, b is the owner of id .

- (E.ix.7) Since b is honest and $\text{ownerOfID}(id) = b$, $\text{secretOfID}(id)$ is only known to b and as . Since as does not send HTTPS requests with key `identity` in the request body to a path `/ciba-auth2`, the request processed by as during P_{as}^{auth2} must have been created by b (see also (E.ix.1)).
- (E.x) The request to as ' `/ciba-auth2` endpoint must be a POST request with the origin header set to the request's host value, which must be a domain of as (Line 281 of Algorithm 11). As b is honest, it will only set the origin header for a request sent by some script accordingly if it is triggered, selects a document loaded from an origin of as , and executes said script.
- (E.xi) The request body processed by as in P_{as}^{auth2} must contain a key `ciba_auth2_reference` with a value auth2Reference , such that there is a value $\text{authnReqId}'$ with $S^{\text{auth2}}(as).\text{cibaAuthnRequests}[\text{authnReqId}'][\text{ciba_auth2_reference}] \equiv \text{auth2Reference}$ (Line 289 of Algorithm 11). Such a value is only stored to the AS state at the `/ciba-auth` endpoint, where a fresh nonce is generated and stored under key `ciba_auth2_reference` (Line 277 of Algorithm 11). This, in turn, only happens when processing an HTTPS request to the as ' `/ciba-auth` endpoint; and the sender of said request receives auth2Reference in the response, together with $\text{script}_{as} \text{ciba}_f \text{orm}$. Since the sender of the request processed by as in P_{as}^{auth2} is b (E.ix.7), and b is an honest browser, b will only use auth2Reference when executing the script $\text{script}_{as} \text{ciba}_f \text{orm}$. I.e., there must have been some processing step Q'' prior to P_{as}^{tokRes} and hence prior to Q , in which b was triggered, selected a document loaded from an origin of as (E.x), and

executed $script_{a,sc}cibaform$ in that document (this is the only script sent by as which sends requests to $/ciba-auth2$). Furthermore, in Line 6 of Algorithm 17, b must have selected id .

By inspection of Line 17 of Algorithm 17, it is obvious that the $scriptstate$ in use during Q'' must contain a key $ciba_auth2_reference$; and the value must be $auth2Reference$ with $S^{auth2}(as).cibaAuthnRequests[authnReqId][ciba_auth2_reference] \equiv auth2Reference$ (see above). Since as never changes that value, and with (E.ix.3) we get $S(as).cibaAuthnRequests[authnReqId][ciba_auth2_reference] \equiv auth2Reference$.

(E.ix.6) additionally gives us $authnReqId = S^{tokReq}(c).sessions[lsid][auth_req_id]$, and since c never changes this value, we get $authnReqId = S(c).sessions[lsid][auth_req_id]$.

Overall, this gives us $authenticatedCIBA_{\rho}^{Q''}(b, c, id, as, lsid)$.

(F) Hence, we have $loggedIn_{\rho}^Q(b, c, id, as, lsid)$ (D) and $authenticatedCIBA_{\rho}^{Q''}(b, c, id, as, lsid)$, which gives us a contradiction to Assumption 4. \blacksquare

E. Session Integrity for Authorization Property

In this section, we show that the session integrity properties for authorization from Definition 25 and Definition 26 hold. We start by proving that the property for authorization code flows (Definition 25) holds.

Lemma 35 (Session Integrity for Authorization Property Holds). For every run ρ of \mathcal{FAPI} , every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , every browser b that is honest in S , every $as \in AS$, every identity u , every client $c \in C$ that is honest in S , every $rs \in RS$ that is honest in S , every nonce r , every nonce $lsid$ such that $S(c).sessions[lsid][cibaFlow] \equiv \perp$, we have that if $accessesResource_{\rho}^Q(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 22), then (1) there exists a processing step Q' in ρ (before Q) such that $started_{\rho}^{Q'}(b, c, lsid)$, and (2) if as is honest in S , then there exists a processing step Q'' in ρ (before Q) such that $authenticated_{\rho}^{Q''}(b, c, u, as, lsid)$.

PROOF. (1). Due to $accessesResource_{\rho}^Q(b, r, u, c, rs, as, lsid)$ (Definition 21), 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 header (with a $sessionId$ cookie) only in Line 69 of Algorithm 3 and Line 42 of Algorithm 2 – however, we can immediately exclude the latter due to precondition $S(c).sessions[lsid][cibaFlow] \equiv \perp$. Hence, c must have set this cookie by executing Line 69 of Algorithm 3.

The remaining proof is analogous to the proof of the first postcondition of Lemma 33.

(2). $accessesResource_{\rho}^Q(b, r, u, c, rs, as, lsid)$ implies that during Q , c executed Line 114 of Algorithm 3 or Line 68 of Algorithm 2. However, we can immediately exclude the latter due to Line 57 of Algorithm 2 and precondition $S(c).sessions[lsid][cibaFlow] \equiv \perp$.

Client received resource from rs . As the client executes Line 114 of Algorithm 3 during Q , and as $S'(c).sessions[lsid][resourceServer] \in \text{dom}(rs)$ (see $accessesResource$) is only set in Line 110 of Algorithm 3, it follows that c received the resource r in a response $resp_{resource}$ from rs .

I.e., c must have sent a corresponding resource request to rs , and the resource response's body processed by c during Q contained r under key $resource$.

Resource request contains access token associated with u at as . An honest resource server sends out an HTTP response $resp_{resource}$ with $resource \in resp_{resource}.body$ either in Line 84 of Algorithm 18 or Line 45 of Algorithm 19. Let $P_{rs}^{resResp} = (S^{resResp}, E^{resResp}, N^{resResp}) \rightarrow (S'^{resResp}, E'^{resResp}, N'^{resResp})$ be the processing step in which rs sends $resp_{resource}$. As shown in the proof of Lemma 31, for this to happen, the resource server must have received a resource request $req_{resource}$ containing an access token t (either in $P_{rs}^{resResp}$ or in another processing step prior to $P_{rs}^{resResp}$ – let $P_{rs}^{resReq} = (S^{resReq}, E^{resReq}, N^{resReq}) \rightarrow (S'^{resReq}, E'^{resReq}, N'^{resReq})$ be the processing step in which rs receives $req_{resource}$).

Furthermore, as the resource server stored the resource in $S(rs).resourceNonces[u][resourceId]$ (see $accessesResource$, for some $resourceId \in \mathcal{T}_{\mathcal{C}}$), it follows that $req_{resource}.path \equiv resourceId$ (see Line 74 of Algorithm 18 for the structured access token case, and Line 53 of Algorithm 18, as well as Lines 9 and 36 of Algorithm 19 for the introspection case). Thus, we have that the value $responsibleAS$ chosen by the resource server in Line 16 of Algorithm 18 during P_{rs}^{resReq} is a domain of as (the resource server never changes the $resourceASMapping$ subterm of its state, see also Definition 9, and from $accessesResource$, we have $as = authorizationServerOfResource^{rs}(resourceID)$).

We now look at the two places in which rs could have produced $resp_{resource}$ during $P_{rs}^{resResp}$: If rs returns the resource r in Line 84 of Algorithm 18, then the access token is a structured JWT signed by as (Line 64 of Algorithm 18) and containing the sub value u (Line 71 of Algorithm 18, and Line 74 of Algorithm 18: r was stored under u , see $accessesResource$). Otherwise, if r is returned in Line 45 of Algorithm 19, then the resource server received an introspection 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 with introspection), it follows that the authorization server as has a record rec in the records subterm of its state with $rec[\text{access_token}] \equiv t$ and $rec[\text{subject}] \equiv u$.

Token request was sent to as . An honest client sends resource requests only in Algorithm 6, which is called only in Line 83 of Algorithm 3, i.e., after receiving the token response. The check in Line 7 of Algorithm 6 ensures that the token request req_{token} was sent to as (as the client calls Algorithm 6 with the domain of the token request, see Line 83 of Algorithm 3). From this, it follows that $S(c).\text{sessions}[lsid][\text{selected_AS}]$ is a domain of as , as the client sends the token request to this domain, see Lines 6, 13, and 14 of Algorithm 4. Note that the token request in question must have been sent from Line 43 of Algorithm 4: the only other place in which c sends token requests is Line 40 of Algorithm 5. However, in the latter case, Line 50 of Algorithm 2 must have been executed, which in turn implies $\text{auth_req_id} \in S(c).\text{sessions}[lsid]$ – with Lemma 27, this gives us $S(c).\text{sessions}[lsid][\text{cibaFlow}] \equiv \top$, which contradicts this lemma’s preconditions.

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 8 of Algorithm 4 and recall that req_{token} must have been sent in Line 43 of Algorithm 4). As the authorization server responds to that request with an access token t , it follows that the checks at the token endpoint in Line 145 of Algorithm 11 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 Lines 154 and 158 of Algorithm 11).

An authorization server adds records with a key $code$ to its records state subterm only in Line 89 of Algorithm 11, and the sequence that is added is taken from the authorizationRequests state subterm, see Line 85 of Algorithm 11. In this processing step, the authorization server also creates the authorization code (Line 88 of Algorithm 11) and associates the identity with the code (Line 86 of Algorithm 11).

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 $/\text{auth2}$ endpoint of as , and the request to $/\text{auth2}$ contained a value auth2reference in its body equal to $S''(as).\text{authorizationRequests}[\text{requestUri}][\text{auth2_reference}]$ (with S'' being the state of a configuration prior to Q ; see also Line 84 of Algorithm 11). The authorization server received the requestUri value at the auth endpoint, i.e., each process that can derive the request URI value, can potentially have sent the $/\text{auth}$ request, and received auth2reference in the response.

As $S(c).\text{sessions}[lsid][\text{selected_AS}]$ is a domain of as , it follows that the client sent a pushed authorization request to as in Line 72 of Algorithm 8 in a previous processing step. In this processing step, the client chose the PKCE verifier $pkce_cv$ in Line 47 of Algorithm 8 and stored this value into the $lsid$ session in Line 53 of Algorithm 8.

Now, we can apply Lemma 24 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 $/\text{auth}$ endpoint, it follows that the request to the $/\text{auth}$ endpoint of as was sent by b . The remaining argumentation is the same as for the proof of Lemma 33. ■

This leaves us with the property for FAPI-CIBA flows (Definition 26). This will be a proof by contradiction, hence we begin by assuming the opposite:

Assumption 5. There exists a FAPI web system with a network attacker $\mathcal{F}API$ such that there exists a run ρ of $\mathcal{F}API$ with a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ , a browser b that is honest in S and behaves according to Assumption 1, an authorization server $as \in \text{AS}$, an identity id , a client $c \in \text{C}$ honest in S , a resource server $rs \in \text{RS}$ that is honest in S , a nonce r , and a nonce $lsid$ s.t. $S(c).\text{sessions}[lsid][\text{cibaFlow}] \equiv \top$, and $\text{accessesResource}_\rho^Q(b, r, id, c, rs, as, lsid)$ such that

- (I) there is no processing step Q' prior to Q in ρ such that $\text{startedCIBA}_\rho^{Q'}(b, c, lsid)$, or
- (II) as is honest in S , and there is no processing step Q'' prior to Q in ρ such that $\text{authenticatedCIBA}_\rho^{Q''}(b, c, id, as, lsid)$.

Lemma 36 (Session Integrity for Authorization for FAPI-CIBA Flows Property Holds). Assumption 5 is a contradiction.

PROOF. We start with some helpful intermediate results:

- (A) From $\text{accessesResource}_\rho^Q(b, r, id, c, rs, as, lsid)$, we know that during Q , c emits an event which contains an HTTPS response whose body is r . Furthermore, due to precondition $S(c).\text{sessions}[lsid][\text{cibaFlow}] \equiv \top$, c must have executed Line 68 of Algorithm 2 during Q , which implies $S(c).\text{sessions}[lsid][\text{resource}] \equiv r$ (Line 66 of Algorithm 2). I.e., **the event emitted by c during Q must originate from Lines 51ff. of Algorithm 2.**
- (B) For c to emit any event during Q (recall (A): c executes Lines 51ff. of Algorithm 2, i.e., must have reached Line 68 of Algorithm 2 in Q), the request processed in Q must have contained a Cookie header with a cookie named $\langle _Host, \text{sessionId} \rangle$ with value $lsid$ (Line 52 of Algorithm 2).
- (C) From $\text{accessesResource}_\rho^Q(b, r, id, c, rs, as, lsid)$, we know $\langle \langle _Host, \text{sessionId} \rangle, \langle lsid, y, z, z' \rangle \rangle \in {}^\diamond S'(b).\text{cookies}[d]$ for some $d \in \text{dom}(c)$. Since that cookie has the $_Host$ prefix and is stored under a domain of c , it must have been set by c (see (C) in the proof of Lemma 24). Hence, there must have been an HTTPS response $res_{\text{start-ciba}}$ sent from c to b (i.e., encrypted for b , i.e., with a key used by b in a previous HTTPS request to c) which contained a corresponding Set-Cookie header.

From here, we can apply the exact same argumentation as in (B.ii) in the proof of Lemma 34, and get: the $\langle _Host, sessionId \rangle$ cookie must originate from the client's `/start_ciba` endpoint in Lines 35ff. of Algorithm 2 (in some processing step R prior to Q).

- (D) The request to c 's `/start-ciba` endpoint (in R , i.e., to which c responded with $res_{start-ciba}$) must have been sent by b : from (C), we have that in R , c executed Lines 35ff. of Algorithm 2, i.e., the cookie set by c is a fresh nonce (Line 38 of Algorithm 2). If the request came from some process $p \neq b$, then b would not have processed the response (due to a missing entry in b 's `pendingRequests` state subterm), and therefore $S'(b).cookies[d]$ would not contain a `sessionId` cookie with value $lsid$.

Let Q' be the processing step in which b emits the corresponding HTTPS request.

With the exact same argumentation as in (C) in the proof of Lemma 34, we can now conclude $b = ownerOfID(S(c).sessions[lsid][selected_identity])$.

- (E) With (C) and (D), we can apply the same argumentation as in (D) in the proof of Lemma 34, and conclude $startedCIBA_p^{Q'}(b, c, lsid)$.
- (F) From (A), we have $S(c).sessions[lsid][resource] \equiv r$. A client only stores something under key `resource` in one of its sessions in Line 109 of Algorithm 3 when processing an HTTP (`resource`) response $res_{resource}$ during some processing step $P_c^{storeR} = (S^{storeR}, E^{storeR}, N^{storeR}) \rightarrow (S^{storeR'}, E^{storeR'}, N^{storeR'})$, hence $res_{resource}.body[resource] \equiv r$. However, c will only process such a response if it also sent a corresponding request $req_{resource}$ (otherwise, there is no matching entry in c 's `pendingRequests`, see Lines 7ff. of Algorithm 39).

In that same processing step P_c^{storeR} , c also executes Line 110 of Algorithm 3, i.e., sets a value for $S(c).sessions[lsid][resourceServer]$. That value is the host to which c sent $req_{resource}$. Since c does not change that value anywhere else, we can use `accessesResource` to conclude that the host of $req_{resource}$ must be a domain of rs .

Since rs and c are honest, we can apply Lemma 46 and get that rs must have created $res_{resource}$.

- (G) From $accessesResource_p^Q(b, r, id, c, rs, as, lsid)$, we have $r \in \langle \rangle S'(rs).resourceNonces[id][resourceId]$ and $as = authorizationServerOfResource^{rs}(resourceID)$ (for some value $resourceId \in \mathcal{T}_{\mathcal{N}}$).
- (H) For $r \in \langle \rangle S'(rs).resourceNonces[id][resourceId]$ from (G), we note that there are only two places in which an RS stores something in its `resourceNonces` state subterm: Line 74 of Algorithm 18 and Line 36 of Algorithm 19 (and that term is initially “empty”, see Definition 9). Let $P = (S^{i_p}, E^{i_p}, N^{i_p}) \rightarrow (S^{i'_p}, E^{i'_p}, N^{i'_p})$ be the processing step during which rs stores r in its `resourceNonces` state subterm.
- (I) P from (H) is unique, i.e., there are no values $id', resourceId'$ with $id \neq id' \vee resourceId \neq resourceId'$ such that $r \in \langle \rangle S'(rs).resourceNonces[id'][resourceId']$, which we prove by looking at the two places in which an RS adds anything to the `resourceNonces` state subterm:

Line 74 of Algorithm 18 The value stored in `resourceNonces` is a fresh nonce (Line 48 of Algorithm 18).

Line 36 of Algorithm 19 The value stored in `resourceNonces` is taken from a record retrieved from the `pendingResponses` state subterm (Lines 3 and 10 of Algorithm 19). Line 3 of Algorithm 19 is the only place where an RS reads something from its `pendingResponses` state subterm, and the record read there is immediately deleted (Line 4 of Algorithm 19). Hence, a value stored in `pendingResponses` is used at most once. Furthermore, the only place where entries are added to this state subterm is Line 53 of Algorithm 18 – where the value for key `resource` is a fresh nonce (which is not stored or sent anywhere else).

Hence, the path of $req_{resource}$ must have been the $resourceId$ from (G) (obvious for case Line 74 of Algorithm 18, in the case of Line 36 of Algorithm 19, the resource id is taken from the same record as the resource itself in Line 8 of Algorithm 19).

- (J) We can now look at the two cases for P from (H) and (I) separately:

(J.i) rs executed Line 74 of Algorithm 18 during P .

(J.i.1) The id under which P stores r during P is taken from a term t under key `sub` (Line 71 of Algorithm 18). That same term t must also contain a key `cnf` (Line 62 of Algorithm 18).

(J.i.2) t must have been created by as : the signature verification in Line 64 of Algorithm 18 uses verification key $verifKey := S^{i_p}(rs).asInfo[d_{as}][as_key]$ (see Line 19 of Algorithm 18), where $d_{as} = S^{i_p}(rs).resourceASMapping[resourceId]$. Since `resourceASMapping` is never changed, initialized using `authorizationServerOfResource` (see Definition 9), and from (G) we have $as = authorizationServerOfResource^{rs}(resourceID)$, this implies $d_{as} \in \text{dom}(as)$. Hence, by Definition 9, we have $verifKey \equiv \text{pub}(\text{signkey}(as))$.

Therefore, we can apply Lemma 8 for the signed t (note that the signed t is known to rs in P) and get that as must have created the signed t .

(J.i.3) Authorization servers only create signed dictionaries with keys `sub` and `cnf` in Line 200 of Algorithm 11. Recall (J.i.1): the value of the `sub` key in t must be id . Hence, by Line 199 of Algorithm 11, there must

be a record rec in as ' records state subterm with $rec[\text{subject}] \equiv id$. From there, we can apply the same argumentation as in the proof of Lemma 34, and hence get that is a processing step Q'' prior to Q in ρ such that $\text{authenticatedCIBA}_\rho^{Q''}(b, c, id, as, lsid)$.

(J.ii) rs executed Line 36 of Algorithm 19 during P .

(J.ii.1) Recall (I): There must have been some processing step P' prior to P during which rs executed Line 53 of Algorithm 18. P' must have ended in Line 60 of Algorithm 18, i.e., with rs sending an introspection request. That introspection request is sent to a domain of as (see above, and recall $as = \text{authorizationServerOfResource}^{rs}(\text{resourceID})$), i.e., encrypted for as . Since rs and as are honest, we can apply Lemma 46 and get that the HTTP response $res_{\text{introspect}}$ processed by rs during P must have been created by as .

(J.ii.2) The body of $res_{\text{introspect}}$ must be a dictionary such that $res_{\text{introspect}}.\text{body}[\text{sub}] \equiv id$ (see (I) and Line 27 of Algorithm 19), and $\text{active} \in res_{\text{introspect}}.\text{body}$. Such an HTTP response is only created by an authorization server in Line 225 of Algorithm 11. This, in turn, requires a record rec in as ' records state subterm with $rec[\text{subject}] \equiv id$ (Line 225 of Algorithm 11). Furthermore, we must have $\text{access_token} \in rec$. Such records are only stored by an AS in Line 203 of Algorithm 11, hence allowing us to apply the same argumentation as above, and hence get that is a processing step Q'' prior to Q in ρ such that $\text{authenticatedCIBA}_\rho^{Q''}(b, c, id, as, lsid)$.

(K) Hence, we have $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$ (E) and $\text{authenticatedCIBA}_\rho^{Q''}(b, c, id, as, lsid)$, which gives us a contradiction to Assumption 5. ■

F. Non-Repudiation Properties

In this section, we show that the non-repudiation properties hold.

Lemma 37 (Non-repudiation for Signed Authorization Requests (Definition 27) Holds). For every run $\rho = ((S^0, E^0, N^0), \dots)$ of $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$, every configuration (S^n, E^n, N^n) in ρ , every process $as \in \text{AS}$ that is honest in S^n , every request uri $requestUri$, we have that if $S^n(as).\text{authorizationRequests}[requestUri][\text{signed_par}] \equiv \top$, then all of the following hold true:

- (I) There exists a processing step $Q = (S, E, N) \xrightarrow{e_{\text{in}} \rightarrow as} (S', E', N')$ with (S, E, N) prior to (S^n, E^n, N^n) in ρ , such that $requestUri \notin S(as).\text{authorizationRequests}$ and $requestUri \in S'(as).\text{authorizationRequests}$.
- (II) $e_{\text{in}} = \langle x, y, m \rangle$ contains a message m of the form $\text{enc}_a(\langle \langle \text{HTTPReq}, \cdot, \text{POST}, \text{selectedAS}, /par, \cdot, \rangle, \text{body} \rangle, \cdot, \cdot)$, where $body$ is of the form $\text{sig}(par, signKey)$ and $\text{selectedAS} \in \text{dom}(as)$.
- (III) If there is a process $c \in \mathbb{C}$ which is honest in S^n , and a configuration (S^i, E^i, N^i) in ρ with $S^i(c).\text{asAccounts}[\text{selectedAS}][\text{sign_key}] \equiv signKey$, then there is a processing step $P = (S^j, E^j, N^j) \rightarrow (S^{j+1}, E^{j+1}, N^{j+1})$ in ρ prior to Q during which c signs par (as contained in e_{in}) in Line 63 of Algorithm 8.

PROOF. (I) Initially, an authorization server's $\text{authorizationRequests}$ state subterm is empty (Definition 8). Hence, we have $requestUri \notin S^0(as).\text{authorizationRequests}$. By induction, we get that there must be some processing step Q in ρ , during which $requestUri$ is added to the $\text{authorizationRequests}$ state subterm.

(II) The only place in which an honest AS adds a new record to its $\text{authorizationRequests}$ state subterm is Line 142 of Algorithm 11.⁹ This line is only executed when processing an HTTPS request, i.e., a message of the form $\text{enc}_a(\langle \langle \text{HTTPReq}, \cdot, \text{method}, \text{host}, \text{path}, \cdot, \rangle, \text{body} \rangle, \cdot)$ (see Lines 7ff. of Algorithm 39). Due to Line 103 of Algorithm 11, we know $path \equiv /par$ and $\text{method} \equiv \text{POST}$. Furthermore, Lemma 2 gives us $host \in \text{dom}(as)$. We can also infer that $body$ must have the form $\text{sig}(par, signKey)$ from Lines 106 and 107 of Algorithm 11, since we know that $\text{requireSignedPAR} \equiv \top$ from this lemma's precondition, together with Lines 138 and 142 of Algorithm 11.

(III) In the following, we assume that there is a process $c \in \mathbb{C}$, honest in S^n , and a configuration (S^i, E^i, N^i) such that $S^i(c).\text{asAccounts}[\text{host}][\text{sign_key}] \equiv signKey$. Since $signKey$ belongs to an honest client, Lemma 4 gives us $signKey \notin d_\emptyset(S^n(p))$ for all processes $p \neq c$, i.e., only c can derive $signKey$. Therefore, only c can have created a term $\text{sig}(par, signKey)$ (see the equational theory in Figure 5).

From Lines 107 and 109 of Algorithm 11, we know that par must be a dictionary with at least the keys aud with value $host$, and client_id with a client id registered with as . An honest client – like c – creates signatures only in a few places:

Line 22 of Algorithm 4 The signed value is a dictionary, but does not contain a key client_id .

Line 39 of Algorithm 4 The signed value is a dictionary, but does not contain a key client_id .

Line 19 of Algorithm 5 The signed value is a dictionary, but does not contain a key client_id .

Line 36 of Algorithm 5 The signed value is a dictionary, but does not contain a key client_id .

⁹Note that Line 73 of Algorithm 11 does not add a new record, but extends an existing one, see Line 67 of Algorithm 11.

- Line 26 of Algorithm 6** The signed value is a dictionary, but does not contain a key `client_id`.
Line 39 of Algorithm 6 The signed value is a dictionary, but does not contain a key `client_id`.
Line 40 of Algorithm 8 The signed value is a dictionary, but does not contain a key `client_id`.
Line 63 of Algorithm 8 The signed value meets the aforementioned conditions.

Note that since only c can create such a signature, and par contains a fresh nonce (Line 47 of Algorithm 8), the term $\text{sig}(par, \text{signKey})$ cannot be derivable by any process prior to $P^{\text{createPAR}}$.

Hence, we conclude that c must have signed par during some processing step $P^{\text{createPAR}}$ in Line 63 of Algorithm 8. ■

Lemma 38 (Non-repudiation for Signed Authorization Responses (Definition 28) Holds). For every run $\rho = ((S^0, E^0, N^0), \dots)$ of $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$, every configuration (S^n, E^n, N^n) in ρ , every process $c \in \mathcal{C}$ that is honest in S^n , every session id $sessionId$, we have that if

- (1) there exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \rightarrow c} (S', E', N')$ with (S, E, N) prior to (S^n, E^n, N^n) in ρ such that $\text{redirectEpRequest} \notin S(c).\text{sessions}[sessionId]$ and $\text{redirectEpRequest} \in S'(c).\text{sessions}[sessionId]$, and
- (2) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $\text{enc}_a(\langle \langle \text{HTTPReq}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle, \text{redirect_ep}, \text{parameters}, \text{headers}, \cdot, \cdot, \cdot \rangle, \cdot, \cdot, \cdot)$, and
- (3) $S^n(c).\text{sessions}[sessionId][\text{requested_signed_authz_response}] \equiv \top$,

then all of the following hold true:

- (I) The term parameters from (2) above is a dictionary with at least a key `response` with value $\text{sig}(\text{authzResponse}, \text{signKey})$, with authzResponse being a dictionary with at least the keys `iss` and `code`.
- (II) If there is an $as \in \text{AS}$ with $S^n(as).\text{jwt} \equiv \text{signKey}$, and as honest in S^n , then there is a processing step $P = (S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ prior to Q in ρ , and as signed authzResponse (as contained in e_{in}) during P in Line 97 of Algorithm 11.

PROOF. We start by noting that (2) is actually implied by (1): an honest client – such as c – only adds a value with key `redirectEpRequest` to a value of its `sessions` state subterm in Line 33 of Algorithm 2. This line, in turn, is only executed when processing an HTTPS request, i.e., a message of the form $\text{enc}_a(\langle \langle \text{HTTPReq}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle, \text{path}, \text{parameters}, \text{headers}, \cdot, \cdot, \cdot \rangle, \cdot, \cdot, \cdot)$, see Lines 7ff. of Algorithm 39, where $\text{path} \equiv \text{/redirect_ep}$ (see Line 12 of Algorithm 2).

- (I) Since (3) implies that during Q , c must execute Lines 18ff. of Algorithm 2, and not stop due to the checks in Lines 19, 22, and 27 of Algorithm 2, we know:
 - $\text{response} \in \text{parameters}$: otherwise $\text{parameters}[\text{response}] \equiv \langle \rangle$ (Definition 41), and by Figure 5, $\text{checksig}(\langle \rangle, k) \not\equiv \top$ for any k , and
 - $\text{parameters}[\text{response}]$ must be of the form $\text{sig}(\text{authzResponse}, \text{signKey})$ (see equational theory in Figure 5), and
 - Lines 21, 24, and 27 of Algorithm 2 imply that authzResponse must be a dictionary with at least the keys `code` and `iss`.

- (II) For the following, we assume that there is an $as \in \text{AS}$ with $S^n(as).\text{jwt} \equiv \text{signKey}$, and as honest in S^n . By applying Lemma 8, we get $\text{signKey} \notin d_0(S^n(p))$ for all processes $p \neq as$, i.e., only as can derive signKey . Therefore, only as can have created a term $\text{sig}(\text{authzResponse}, \text{signKey})$ (see Figure 5). From the same lemma – in conjunction with Figure 5, and the fact that c knows $\text{sig}(\text{authzResponse}, \text{signKey})$ in Q – we also get that as must have created that term in some processing step P prior to Q in ρ .

An honest AS only signs terms in a few places (recall: the signed term authzResponse is a dictionary with at least the keys `iss` and `code`):

- Line 200 of Algorithm 11** The signed value is a dictionary, but it does not contain a key `iss`, nor a key `code`.
Line 212 of Algorithm 11 The signed value is a dictionary with a key `iss`, but it does not contain a key `code`.
Line 227 of Algorithm 11 The signed value is a dictionary with a key `iss`, but it does not contain a key `code`.
Line 97 of Algorithm 11 The signed value is a dictionary with keys `iss` and `code`.

Since Line 97 of Algorithm 11 is the only place in which an honest AS signs a term meeting the above conditions, we conclude that as must have signed authzResponse in a processing step P prior to Q in ρ . ■

Lemma 39 (Non-repudiation for Signed Introspection Responses (Definition 29) Holds). For every run $\rho = ((S^0, E^0, N^0), \dots)$ of $\mathcal{F}\mathcal{A}\mathcal{P}\mathcal{I}$, every configuration (S^n, E^n, N^n) in ρ , every process $rs \in \text{RS}$ that is honest in S^n , every request id $requestId$, we have that if there exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \rightarrow rs} (S', E', N')$ in ρ such that $S(rs).\text{pendingResponses}[requestId][\text{requestSignedIntrospecResponse}] \equiv \top$, and $requestId \notin S'(rs).\text{pendingResponses}$, and (S, E, N) prior to (S^n, E^n, N^n) in ρ , then all of the following hold true:

- (I) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $\text{enc}_s(\langle \langle \text{HTTPResp}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle, \text{body} \rangle, \cdot)$, where body is of the form $\text{sig}(\text{introspecResponse}, \text{signKey})$.

- (II) If there is an $as \in AS$ with $S^n(as).jwk \equiv signKey$, and as honest in S^n , then there is a processing step $P = (S^i, E^i, N^i) \xrightarrow{as \rightarrow E_{out}} (S^{i+1}, E^{i+1}, N^{i+1})$ prior to Q in ρ , and as signed $introspecResponse$ (as contained in e_{in} above) during P in Line 227 of Algorithm 11.

PROOF. (I) The preconditions imply that $requestId \in S(rs).pendingResponses$ (otherwise, $S(rs).pendingResponses[requestId]$ would have the value $\langle \rangle \neq \top$). Hence, rs must have removed the entry with key $requestId$ from its $pendingResponses$ state subterm during Q . An honest RS only removes dictionary keys (i.e., entries) from this state subterm in Line 4 of Algorithm 19. This, in turn, implies that during Q , rs processes an HTTPS response – that is, a message of the form $enc_s(resp, \cdot)$, where $resp \in HTTPResponses$, and hence $resp$ is of the form $\langle HTTPResp, \cdot, \cdot, \cdot, body \rangle$ (see Lines 19ff. of Algorithm 39).

Since $S(rs).pendingResponses[requestId][requestSignedIntrospecResponse] \equiv \top$, we know that rs executed Lines 16ff. of Algorithm 19 without stopping due to the checks in Lines 16 and 19 of Algorithm 19. We can therefore conclude that $resp.body$ must be of the form $sig(introspecResponse, signKey)$ (see Line 16 of Algorithm 19).

Furthermore, in order for Q to actually finish with a changed state (i.e., to reach Line 45 of Algorithm 19 and store the changes to $pendingResponses$), we need $extractmsg(resp.body)$ to be a dictionary with at least the key $token_introspection$ (otherwise, execution would stop in Line 23 of Algorithm 19, see also Line 21 of Algorithm 19).

- (II) For the following, we assume that there is an $as \in AS$ with $S^n(as).jwk \equiv signKey$, and as honest in S^n . By applying Lemma 8, we get $signKey \notin d_\emptyset(S^n(p))$ for all processes $p \neq as$, i.e., only as can derive $signKey$. Therefore, only as can have created a term $sig(introspecResponse, signKey)$ (see Figure 5). From the same lemma – in conjunction with Figure 5, and the fact that rs knows $sig(introspecResponse, signKey)$ in Q – we also get that as must have created that term in some processing step P prior to Q in ρ .

An honest AS only signs terms in a few places (recall: the signed term $introspecResponse$ is a dictionary with at least the key $token_introspection$):

Line 200 of Algorithm 11 The signed value is a dictionary, but it does not contain a key $token_introspection$.

Line 212 of Algorithm 11 The signed value is a dictionary, but it does not contain a key $token_introspection$.

Line 227 of Algorithm 11 The signed value is a dictionary with a key $token_introspection$.

Line 97 of Algorithm 11 The signed value is a dictionary, but it does not contain a key $token_introspection$.

Since Line 227 of Algorithm 11 is the only place in which an honest AS signs a term meeting the above conditions, we conclude that as must have signed $introspecResponse$ in a processing step P prior to Q in ρ . ■

Lemma 40 (Properties of VERIFY_REQUEST_SIGNATURE (Algorithm 21)). For any function call $VERIFY_REQUEST_SIGNATURE(m, verificationKey)$ to return \top , the arguments must meet all of the following conditions:

- (I) $m.headers$ must exist, be a dictionary, and $Signature \in m.headers$
- (II) $req \in m.headers[Signature]$
- (III) $Signature-Input \in m.headers$ and $req \in m.headers[Signature-Input]$
- (IV) $m.headers[Signature][req] \equiv sig(sigBase, sigKey)$ for some $sigBase, sigKey$
- (V) $m.headers[Signature-Input][req]$ is a sequence $\langle coveredComponents, metadata \rangle$ (there may be additional sequence elements after those two), where $metadata$ is a dictionary with at least a key tag with value $fapi-2-request$, and $coveredComponents$ is a sequence with at least the following elements: $\langle @method, \langle \rangle \rangle$, $\langle @target-uri, \langle \rangle \rangle$, $\langle authorization, \langle \rangle \rangle$, and $\langle content-digest, \langle \rangle \rangle$.
- (VI) The value of $sigBase$ from (IV) is a dictionary with the following properties:
- (VI.a) $sigBase[\langle @method, \langle \rangle \rangle] \equiv m.method$
- (VI.b) $sigBase[\langle @target-uri, \langle \rangle \rangle] \equiv \langle URL, S, m.host, m.path, m.parameters, \perp \rangle$
- (VI.c) $sigBase[\langle authorization, \langle \rangle \rangle] \equiv m.headers[Authorization]$
- (VI.d) $sigBase[\langle content-digest, \langle \rangle \rangle] \equiv hash(m.body)$
- (VI.e) $sigBase.2[tag] \equiv fapi-2-request$ and $keyid \in sigBase.2$

PROOF. We start by noting that there is only one place in which Algorithm 21 returns \top , namely Line 17 of Algorithm 21. Hence, all execution paths not leading to this line do not return \top .

(I) Obvious from Line 2 of Algorithm 21.

(II) Proof by contradiction: if $req \notin m.headers[Signature]$, then $m.headers[Signature][req] = \langle \rangle$ (Definition 41). $extractmsg(\langle \rangle)$ in Line 6 of Algorithm 21 is undefined (Figure 5), and therefore $VERIFY_REQUEST_SIGNATURE$ does not return anything. This is a contradiction to $VERIFY_REQUEST_SIGNATURE(m, verificationKey)$ returning \top .

(III) Proof by contradiction: if $req \notin m.headers[Signature-Input]$ or $Signature-Input \notin m.headers$, then $m.headers[Signature-Input][req] = \langle \rangle$ (Definition 41). Hence, $coveredComponents = \langle \rangle$ in Line 5 of Algorithm 21. Recall Definition 42 and Figure 5: $\langle \rangle.1 := \pi_1(\langle \rangle) := \diamond$. Therefore the first check in Line 8

of Algorithm 21 boils down to `@method` \notin \diamond which is undefined (\diamond is not a dictionary, see Definition 41), and consequently, `VERIFY_REQUEST_SIGNATURE` does not return anything. This is a contradiction to `VERIFY_REQUEST_SIGNATURE(m , $verificationKey$)` returning \top .

- (IV) The signature verification in Line 15 of Algorithm 21 must return \top for `VERIFY_REQUEST_SIGNATURE` to return \top , which requires the first argument given to `checksig(\cdot , \cdot)` to match `sig(\ast , \ast)` (see Figure 5).
- (V) The value of `$m.headers[Signature-Input][req]$` is stored in a variable `$coveredComponents$` in Line 5 of Algorithm 21, which is used in several checks in Line 8 of Algorithm 21, verifying the presence of the elements required in (V).
- (VI) From (V), we have `$\langle @method, \langle \rangle \rangle$` , `$\langle @target-uri, \langle \rangle \rangle$` , `$\langle authorization, \langle \rangle \rangle$` , and `$\langle content-digest, \langle \rangle \rangle$` \in $\langle \rangle$ `$coveredComponents.1$` (from Line 5 of Algorithm 21). Hence, these are covered by the loop in Lines 11ff. of Algorithm 21, and thus passed to `IS_COMPONENT_EQUAL` (Algorithm 1) in Line 12 of Algorithm 21 with `m` as the first, the `$sigBase$` as the second to last, and the respective component identifier as the last argument.

It is easy to see that the results of `IS_COMPONENT_EQUAL` are only \top (and hence, the checks in Algorithm 21 on these results succeed) if conditions (VI.a), (VI.b), and (VI.c) are fulfilled. Furthermore, this gives us `$m.headers[Content-Digest] \equiv sigBase[\langle content-digest, \langle \rangle \rangle]$` – which, together with Line 3 of Algorithm 21, gives us (VI.d).

This leaves us with (VI.e) to prove, which is ensured by the check in Line 9 of Algorithm 21.

Lemma 41 (Non-repudiation for Signed Resource Requests (Definition 30) Holds). For every run $\rho = ((S^0, E^0, N^0), \dots)$ of $\mathcal{F}API$, every configuration (S^n, E^n, N^n) in ρ , every process $rs \in RS$ that is honest in S^n , we have that if

- (1) there exists a processing step $Q = (S, E, N) \xrightarrow{rs \rightarrow E_{out}} (S', E', N')$ in ρ such that $E_{out} = \langle \langle x, y, resRes \rangle, leakedRequest \rangle$, with (S, E, N) prior to (S^n, E^n, N^n) , and
- (2) during Q , either Line 69 of Algorithm 18 or Line 33 of Algorithm 19 was executed,

then all of the following hold true:

- (I) `$resRes$` is of the form `$enc_s(\langle HTTPResp, \cdot, \cdot, \cdot, body \rangle, \cdot)$` with `$body \equiv [resource: resource]$` .
- (II) There exists a processing step $R = s^r \xrightarrow{e_{in} \rightarrow rs} s^{r'}$ prior or equal to Q in ρ such that $e_{in} = \langle x, y, resReq \rangle$, and rs generated `$resource$` during R in Line 48 of Algorithm 18.
- (III) `$resReq$` is of the form `$enc_a(\langle \langle HTTPReq, \cdot, method, host, path, parameters, headers, body \rangle, \cdot, \cdot \rangle)$` with `$Signature \in headers$` , `$Signature-Input \in headers$` , and `$headers[Signature]$` being a dictionary with at least a key `req` with value `$sig(signatureBase, clientSignKey)$` .
- (IV) `$headers[Signature-Input][req]$` is a sequence `$\langle coveredComponents, metadata \rangle$` (there may be additional sequence elements after those two), where `$metadata$` is a dictionary with at least a key `tag` with value `$fapi-2-request$` , and `$coveredComponents$` is a sequence with at least the following elements: `$\langle @method, \langle \rangle \rangle$` , `$\langle @target-uri, \langle \rangle \rangle$` , `$\langle authorization, \langle \rangle \rangle$` , and `$\langle content-digest, \langle \rangle \rangle$` .
- (V) `$signatureBase$` is of the form `$[\langle @method, \langle \rangle \rangle: method, \langle @target-uri, \langle \rangle \rangle: \langle URL, S, host, path, parameters, \perp \rangle, \langle authorization, \langle \rangle \rangle: headers[Authorization], \langle content-digest, \langle \rangle \rangle: hash(body)] + \langle \rangle [tag: fapi-2-request, keyid: keyId]$` for some `$keyId$` ; however, the dictionaries may contain additional elements.
- (VI) If there is a client $c \in C$ which is honest in S^n , a domain `$selectedAS$` , and an index $j \leq n$ such that $S^j(c).asAccounts[selectedAS][sign_key] \equiv clientSignKey$, then there is a processing step $P = (S^i, E^i, N^i) \xrightarrow{c \rightarrow E'_{out}} (S^{i+1}, E^{i+1}, N^{i+1})$ prior to R in ρ , and c signed `$signatureBase$` (as contained in e_{in} above) during P in Line 39 of Algorithm 6.

PROOF. (I) From (1), we have that during Q , rs outputs two events. With that in mind, we look at the two cases of (2):

Line 69 of Algorithm 18 After making it to this line, the only possibility to output two events is the stop in Line 84 of Algorithm 18. There, the first event contains a message m' , created in Line 81 of Algorithm 18 as a message of the form `$enc_s(\langle HTTPResp, \cdot, 200, \cdot, body \rangle, \cdot)$` . The value for `$body$` is created in Line 75 of Algorithm 18 as `$[resource: resource]$` .

Line 33 of Algorithm 19 After making it to this line, the only possibility to output two events is the stop in Line 45 of Algorithm 19. There, the first event contains a message m' , created in Line 42 of Algorithm 19 as a message of the form `$enc_s(\langle HTTPResp, \cdot, 200, \cdot, body \rangle, \cdot)$` . The value for `$body$` is created in Line 37 of Algorithm 19 as `$[resource: resource]$` .

- (II) From (I), we know that rs created a resource response body in one of the following places during Q , for which we will determine where the `$resource$` value originates from:

Line 75 of Algorithm 18 The value for `$resource$` was generated in Line 48 of Algorithm 18 of the same processing step, i.e., $R = Q$. We note that some input event is always required for a processing step (see Definition 54), hence concluding this sub-proof.

Line 37 of Algorithm 19 In this case, the value for `$resource$` is taken from rs' `$pendingResponses$` state subterm (Lines 3 and 10 of Algorithm 19). Corresponding entries in rs' `$pendingResponses$` state subterm are only created

in Line 53 of Algorithm 18, where the value for the dictionary key `resource` is a fresh nonce generated in Line 48 of Algorithm 18. I.e., there must have been some processing step R during which rs generated a value in Line 48 of Algorithm 18, and stored that value into its state in Line 53 of Algorithm 18. After storing that value to its state, rs calls `HTTPS_SIMPLE_SEND` in Line 61 of Algorithm 18, which ends the processing step (in particular, without executing Algorithm 19), therefore $R \neq Q$, and R prior to Q . We again note that some input event is always required for a processing step (see Definition 54), hence concluding this sub-proof.

(III) From (II), we know that during R , rs processed an input event $\langle y, x, resReq \rangle$, and executed Line 48 of Algorithm 18. Hence, rs must have executed Algorithm 18, which is only called in Line 9 of Algorithm 39. This in turn only happens if the input message, i.e., $resReq$, is of the form $enc_a(\langle \langle \text{HTTPReq}, \cdot, method, host, path, parameters, headers, body \rangle, \cdot \rangle, \cdot)$ (see Lines 7ff. of Algorithm 39).

For the remaining conditions, we distinguish between the two possible cases $R = Q$ and $R \neq Q$ established in the proof of (II) above.

Case $R = Q$. In the proofs of (I) and (II), we established that execution during R must reach the stop in Line 84 of Algorithm 18. Therefore, the check in Line 50 of Algorithm 18 must have come up negative, i.e., execution continued in Line 62 of Algorithm 18 (otherwise, R would have stopped inside `HTTPS_SIMPLE_SEND`). Furthermore, (2) gives us that Line 69 of Algorithm 18 was executed during R , i.e., $expectSignedRequest$ had value \top in Line 66 of Algorithm 18. Hence, none of the checks in Lines 62, 64, and 69 of Algorithm 18 failed (i.e., lead to a parameterless stop). For the check in Line 69 of Algorithm 18 to succeed, the call to `VERIFY_REQUEST_SIGNATURE` (Algorithm 21) in Line 68 of Algorithm 18 must return \top . This allows us to apply Lemma 40, concluding this sub-proof.

Case $R \neq Q$.

Storing $resReq$ during R . In the proofs of (I) and (II), we established that execution during R must reach the call to `HTTPS_SIMPLE_SEND` in Line 61 of Algorithm 18, and hence, the stop in Line 3 of Algorithm 34 – in other words, all changes to rs ' state made in Algorithm 18 are indeed stored (i.e., execution did not finish at a parameterless stop). This includes the record stored to the `pendingResponses` state subterm in Line 53 of Algorithm 18. Note that the key $requestId$ under which the whole record gets stored is a fresh nonce (i.e., there are no key “collisions”, since Line 53 of Algorithm 18 is the only place in which an honest RS adds elements to its `pendingResponses` state subterm). Said record includes, among other dictionary elements, a key `originalRequest` with value m , where m is the first argument given to Algorithm 18 – which is only called in Line 9 of Algorithm 39 with the decrypted first sequence element of the input event as the first element, i.e., $m \equiv dec_a(resReq, k).1$ with the “correct” k , and hence $m \equiv \langle \text{HTTPReq}, \cdot, method, host, path, parameters, headers, body \rangle$. In addition to m , the record also includes a key `resource` with value $resource$, i.e., the value created in Line 48 of Algorithm 18 (see (I) and (II)). So, at the end of processing step R , m is stored under key `originalRequest`, together with $resource$ under key `resource`, as part of a record stored under some (unique) $requestId$ in the `pendingResponses` state subterm.

Linking R and Q , Accessing $resReq$ During Q . Records stored in the `pendingResponses` state subterm are only accessed in Line 3 of Algorithm 19, and deleted from the state immediately after accessing them. Hence, each of those records – which each contain a fresh resource, see proof of (II) – is accessed at most once, including the one accessed during Q , i.e., R and Q can be uniquely “linked” via the resource stored during R and output during Q . Hence, the record r in `pendingResponses` “used” during Q , and in particular m and $resource$ within r , are indeed the values stored during R .

Signature Check During Q . From (2) (and $R \neq Q$), we know that during Q , rs must have executed Line 33 of Algorithm 19 and (see proof for (I)) Q finished at the stop in Line 45 of Algorithm 19. This implies that – among others – the check in Line 34 of Algorithm 19 succeeded, i.e., did not lead to a parameterless stop. I.e., the call to `VERIFY_REQUEST_SIGNATURE` in Line 33 of Algorithm 19 must have returned \top . The first argument in that call is $origReq$, which is taken from the aforementioned record r in Line 7 of Algorithm 19, i.e., $origReq \equiv r[\text{originalRequest}]$, which is the value stored as m during R (note: in the context of Q , m refers to the introspection response, and no longer to the resource request, hence the new variable name $origReq$).

Since $origReq \equiv dec_a(resReq, k).1$ (see above), we can apply Lemma 40 to conclude this sub-proof.

(IV) With the same argumentation as in the proof of (III), we can apply Lemma 40 and immediately get (IV).

(V) With the same argumentation as in the proof of (III), we can apply Lemma 40 and immediately get (V).

(VI) For the following, we assume that there is a client $c \in C$, honest in S^n , an issuer identifier $selectedAS$, and an index $j \leq n$ such that $S^j(c).asAccounts[selectedAS][\text{sign_key}] \equiv clientSignKey$ (i.e., the key used for the signature from (III)).

Since c is honest, and we have $selectedAS \in S^j(c).asAccounts$, Lemma 4 gives us $clientSignKey \notin d_\emptyset(S^n(p))$ for all $p \neq c$, i.e., only c can derive $clientSignKey$. Therefore, only c can have created a term $\text{sig}(signatureBase, clientSignKey)$ (see Figure 5).

From (V), we have some conditions on the structure of *signatureBase*. With those in mind, we can look at all places in which an honest client creates signatures:

- Line 22 of Algorithm 4** The signed value does not meet the conditions from (V).
- Line 39 of Algorithm 4** The signed value does not meet the conditions from (V).
- Line 19 of Algorithm 5** The signed value does not meet the conditions from (V).
- Line 36 of Algorithm 5** The signed value does not meet the conditions from (V).
- Line 26 of Algorithm 6** The signed value does not meet the conditions from (V).
- Line 39 of Algorithm 6** The signed value meets the conditions from (V).
- Line 40 of Algorithm 8** The signed value does not meet the conditions from (V).
- Line 63 of Algorithm 8** The signed value does not meet the conditions from (V).

Since only *c* can have created a term $\text{sig}(\text{signatureBase}, \text{clientSigKey})$, honest clients only create such a term in Line 39 of Algorithm 6, and this term is part of the input event in processing step *R*, we conclude that there must be a processing step *P* prior to *R* in ρ during which *c* signed *signatureBase* in Line 39 of Algorithm 6. ■

Lemma 42 (Non-repudiation for Signed Resource Responses (Definition 31) Holds). For every run $\rho = ((S^0, E^0, N^0), \dots)$ of $\mathcal{F}API$, every configuration (S^n, E^n, N^n) in ρ , every client $c \in \mathbb{C}$ which is honest in S^n , every session id $\text{sessionId} \in S^n(c).\text{sessions}$ such that

- (1) $S^n(c).\text{sessions}[\text{sessionId}][\text{expect_signed_resource_res}] \equiv \top$, and
- (2) $S^n(c).\text{sessions}[\text{sessionId}][\text{resource}] \equiv \text{resource}$,

then all of the following hold true:

- (I) There exists a processing step $P = (S, E, N) \xrightarrow{e_{in} \rightarrow c} (S', E', N')$ in ρ with (S, E, N) prior to (S^n, E^n, N^n) where $e_{in} = \langle x, y, m \rangle$, with *m* having the form $\text{enc}_s(\langle \text{HTTPResp}, \cdot, \text{status}, \text{headers}, \text{body} \rangle, \cdot)$, where $\text{body} \equiv [\text{resource}: \text{resource}]$, and $S(c) \neq S'(c)$.
- (II) $\text{headers}[\text{Signature-Input}]$ is a dictionary with at least a key *res* such that $\text{headers}[\text{Signature-Input}][\text{res}]$ is a sequence with at least two elements. For those first two elements, *components*, and *metadata*, we have $\langle @\text{status}, \langle \rangle \rangle$, $\langle \text{content-digest}, \langle \rangle \rangle \in \langle \rangle \text{components}$, and *metadata* is a dictionary with at least the key *tag* such that $\text{metadata}[\text{tag}] \equiv \text{fapi-2-response}$.
- (III) $\text{headers}[\text{Signature}]$ is a dictionary with at least a key *res* such that $\text{headers}[\text{Signature}][\text{res}] \equiv \text{sig}(\text{signatureBase}, \text{rsSigKey})$.
In addition, *signatureBase* is of the form $[\langle @\text{status}, \langle \rangle \rangle: \text{status}, \langle \text{content-digest}, \langle \rangle \rangle: \text{hash}(\text{body})] + \langle \rangle [\text{tag}: \text{fapi-2-response}, \text{keyid}: \text{keyId}']$ for some *keyId'*; however, the dictionaries may contain additional elements.
- (IV) There exists a domain $\text{rsDom} \in S^n(c).\text{rsSigKeys}$ such that $S^n(c).\text{rsSigKeys}[\text{rsDom}] \equiv \text{pub}(\text{rsSigKey})$.
- (V) If process $\text{rs} := \text{dom}^{-1}(\text{rsDom})$ is honest in S^n , then there is a processing step $Q = s \xrightarrow{\text{rs} \rightarrow E_{\text{out}}} s'$, and *rs* signed the resource response contained in *m* during *Q* in Line 6 of Algorithm 20.

PROOF. (I) Precondition (2) implies that *c* must have stored some value under key *resource* in a record within its *sessions* state subterm prior to (S^n, E^n, N^n) (and the *sessions* state subterm is initially empty, see Definition 7). An honest client only stores values under that key in Line 109 of Algorithm 3. Algorithm 3 is only called in Line 26 of Algorithm 39, which only happens if the input event to the current processing step is an encrypted HTTP response (see Lines 19ff. of Algorithm 39), i.e., matches $\langle *, *, \text{enc}_s(\langle \text{HTTPResp}, *, *, *, * \rangle, *) \rangle$.

Furthermore, execution during such a processing step must of course reach Line 109 of Algorithm 3 (and end in a stop with a state parameter, otherwise, nothing gets stored under key *resource*, contradicting (2)). The value stored to *c*'s *sessions* state subterm in that line is taken from the decrypted (Line 20 of Algorithm 39) input message's body component under the key *resource* (Line 108 of Algorithm 3). Hence, $\text{dec}_s(m, k).\text{body}[\text{resource}]$ (for the "correct" *k*) is equivalent to *resource* as stored in the *sessions* state subterm.

Signature Check During *P*. From (1) we have $S^n(c).\text{sessions}[\text{sessionId}][\text{expect_signed_resource_res}] \equiv \top$.

Since execution during *P* must reach Line 109 of Algorithm 3 (otherwise $S(c) = S'(c)$), we know that Line 89 of Algorithm 3 must have been executed as well. Because *resource* can only be stored in Line 109 of Algorithm 3 after *expect_signed_resource_res* was stored in Line 90 of Algorithm 3, $S^n(c).\text{sessions}[\text{sessionId}][\text{expect_signed_resource_res}] \equiv \top$ implies that *expectSignedResponse* is chosen as \top in Line 89 of Algorithm 3 during *P*. Hence, during *P*, Lines 91ff. of Algorithm 3 are executed.

- (II) As shown in "Signature Check During *P*" above, Lines 91ff. of Algorithm 3 are executed during *P*. $\text{headers}[\text{Signature-Input}][\text{res}]$ must be a sequence with at least two elements, as otherwise, accessing the sequence elements in Line 98 of Algorithm 3 would be undefined, and hence, *P* would not exist in ρ .

Furthermore, the same check ensures $\langle @status, \langle \rangle \rangle, \langle content-digest, \langle \rangle \rangle \in \langle \rangle headers[Signature-Input][res].1$, and $headers[Signature-Input][res].2[tag] \equiv fapi-2-response$ – if this check fails, P stops without parameters in Line 99 of Algorithm 3, which is a contradiction to (I).

- (III) As shown in “Signature Check During P ” above, Lines 91ff. of Algorithm 3 are executed during P , and from (I), we have that P does not end in a parameterless stop. This implies that the checks in Lines 100, 104, and 106 of Algorithm 3 succeed (i.e., the if-conditions are false).

$res \in headers[Signature]$ Proof by contradiction: if $res \notin headers[Signature]$, then $headers[Signature][res] \equiv \langle \rangle$ (Definition 41). $extractmsg(\langle \rangle)$ in Line 97 of Algorithm 3 is undefined (Figure 5), and therefore P is not in ρ . This is a contradiction to (I).

$headers[Signature][res] \sim sig(*, *)$ Proof by contradiction: assume $headers[Signature][res] \not\sim sig(*, *)$. Therefore $\forall k. checksig(headers[Signature][res], k) \neq \top$ in Line 106 of Algorithm 3, hence P stops without parameters, which is a contradiction to (I).

$signatureBase[\langle @status, \langle \rangle \rangle] \equiv status$ We have $\langle @status, \langle \rangle \rangle \in \langle \rangle headers[Signature-Input][res].1$ from (II). Therefore, $IS_COMPONENT_EQUAL$ (Algorithm 1) in Line 103 of Algorithm 3 gets called with $\langle @status, \langle \rangle \rangle$ as last argument, $signatureBase$ as second-to-last argument, and the decrypted m as first argument. It is easy to see that Algorithm 1 only returns \top (which is needed, as otherwise P would stop without parameters due to Line 104 of Algorithm 3), if $signatureBase[\langle @status, \langle \rangle \rangle] \equiv status$.

$signatureBase[\langle content-digest, \langle \rangle \rangle] \equiv hash(body)$ From Line 92 of Algorithm 3 and (I) (P does not stop without parameters), we get $headers[Content-Digest] \equiv hash(body)$. With the same argumentation as above for $@status$, we get $signatureBase[\langle content-digest, \langle \rangle \rangle] \equiv headers[Content-Digest]$. Combining these, we get $signatureBase[\langle content-digest, \langle \rangle \rangle] \equiv hash(body)$.

Value of tag and existence of keyid in $signatureBase$ Obvious from Line 100 of Algorithm 3 and (I).

- (IV) From (I), we know that the signature check in Line 106 of Algorithm 3 must succeed. Hence, the value for $pubKey$ used there must be $pub(rsSigKey)$ (see Figure 5). Furthermore, this value is taken from c 's $rsSigKeys$ state subterm in Line 96 of Algorithm 3 with a key $rsDom$. Note: if $rsDom \notin S(c).rsSigKeys$, the value of $pubKey$ would be $\langle \rangle \not\sim pub(*)$.
- (V) From Definition 7, we have $dom^{-1}(rsDom) \in RS$ for all $rsDom \in S^0(c).rsSigKeys$. However, since an honest client never changes the contents of its $rsSigKeys$ state subterm, we also get $dom^{-1}(rsDom) \in RS$ for all $rsDom \in S(c).rsSigKeys$, and hence, we can assume that $rs := dom^{-1}(rsDom) \in RS$ is an honest resource server (in S^n). Furthermore, Definition 7 gives us $S(c).rsSigKeys[rsDom] \equiv pub(signkey(rs))$. This allows us to apply Lemma 10: only rs can derive $rsSigKey$, and hence, only rs can have created a term $sig(signatureBase, rsSigKey)$ (see Figure 5). An honest resource server only creates signatures in Line 6 of Algorithm 20, and since c processes $sig(signatureBase, rsSigKey)$, i.e., a signature created by rs , in P , there must have been a processing step Q during which rs created $sig(signatureBase, rsSigKey)$, i.e., signed $signatureBase$. ■

G. Proof of Theorem

Theorem 1 follows from Lemmas 31-39, Lemma 41, and Lemma 42.

$$\text{dec}_a(\text{enc}_a(x, \text{pub}(y)), y) = x \quad (7)$$

$$\text{dec}_s(\text{enc}_s(x, y), y) = x \quad (8)$$

$$\text{checksig}(\text{sig}(x, y), \text{pub}(y)) = \top \quad (9)$$

$$\text{extractmsg}(\text{sig}(x, y)) = x \quad (10)$$

$$\text{checkmac}(\text{mac}(x, y), y) = \top \quad (11)$$

$$\text{extractmsg}(\text{mac}(x, y)) = x \quad (12)$$

$$\pi_i(\langle x_1, \dots, x_n \rangle) = x_i \text{ if } 1 \leq i \leq n \quad (13)$$

$$\pi_j(\langle x_1, \dots, x_n \rangle) = \diamond \text{ if } j \notin \{1, \dots, n\} \quad (14)$$

$$\pi_j(t) = \diamond \text{ if } t \text{ is not a sequence} \quad (15)$$

Figure 5. Equational theory for Σ .

APPENDIX E TECHNICAL DEFINITIONS

Here, we provide technical definitions of the WIM. These follow the descriptions in [12, 16–21].

A. Terms and Notations

Definition 32 (Signature Σ). We define the signature Σ , over which we will define formal terms, as the union of the following pairwise disjoint sets:

Constants $\mathcal{C} = \mathbb{S} \cup \text{IPs} \cup \{\perp, \top, \diamond\}$ with the three sets pairwise disjoint. \mathbb{S} is the set of all (ASCII) strings, including the empty string ε . IPs is 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: $\text{pub}(\cdot)$, $\text{enc}_a(\cdot, \cdot)$, $\text{dec}_a(\cdot, \cdot)$, $\text{enc}_s(\cdot, \cdot)$, $\text{dec}_s(\cdot, \cdot)$, $\text{sig}(\cdot, \cdot)$, $\text{checksig}(\cdot, \cdot)$, $\text{mac}(\cdot, \cdot)$, $\text{checkmac}(\cdot, \cdot)$, $\text{extractmsg}(\cdot)$, $\text{hash}(\cdot)$.

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

Projection Symbols to access sequence elements: $\pi_i(\cdot)$ for all $i \in \mathbb{N}_0$. Note that formally, projection symbols are also function symbols.

Definition 33 (Nonces and Terms). By $X = \{x_0, x_1, \dots\}$ we denote a set of variables and by \mathcal{N} we denote an infinite set of constants (*nonces*) such that Σ , X , and \mathcal{N} are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X \cup \mathcal{C}$, then t is a term. (2) If $f \in \Sigma$ is an n -ary function symbol for some $n \geq 0$ and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.

By \equiv we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by the theory associated with Σ (see Figure 5). For example, we have that $\pi_1(\text{dec}_a(\text{enc}_a(\langle a, b \rangle, \text{pub}(k)), k)) \equiv a$.

Definition 34 (Ground Terms, Messages, Placeholders, Protomessages). By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set \mathcal{M} of messages (over \mathcal{N}) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

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

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

Definition 35 (Events and Protoevents). An *event* (over IPs and \mathcal{M}) is a term of the form $\langle a, f, m \rangle$, for $a, f \in \text{IPs}$ 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}^ν are called *protoevents* and are denoted \mathcal{E}^ν . By $2^{\mathcal{E}\langle \cdot \rangle}$ (or $2^{\mathcal{E}^\nu\langle \cdot \rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle \rangle$, $\langle \langle a, f, m \rangle, \langle a', f', m' \rangle, \dots \rangle$, etc.).

Definition 36 (Normal Form). Let t be a term. The *normal form* of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Figure 5. For a term t , we denote its normal form as $t\downarrow$.

Definition 37 (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 $t \sim pattern$. For a sequence of patterns $patterns$ we write $t \sim patterns$ to denote that t matches at least one pattern in $patterns$.

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

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

$$\langle \langle \perp, \top \rangle, \langle \top, 23 \rangle, \langle a, b \rangle, \langle \top, \perp \rangle \rangle | p = \langle \langle \top, 23 \rangle, \langle \top, \perp \rangle \rangle .$$

Definition 38 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$, and $t_1, \dots, t_n \in \mathcal{T}_N$.

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

Definition 39 (Sequence Notations). Let $t = \langle t_1, \dots, t_n \rangle$ and $r = \langle r_1, \dots, r_m \rangle$ be sequences, s a set, and x, y terms. We define the following operations:

- $t \subset^{\langle \rangle} s \iff t_1, \dots, t_n \in s$
- $x \in^{\langle \rangle} t \iff \exists i: t_i = x$
- $t +^{\langle \rangle} y := \langle t_1, \dots, t_n, y \rangle$
- $t \cup r := \langle t_1, \dots, t_n, r_1, \dots, r_m \rangle$
- $t -^{\langle \rangle} y := \begin{cases} \langle t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n \rangle & \text{if } \exists i: t_i = x \text{ (i.e., } y \in^{\langle \rangle} t) \\ t & \text{otherwise (i.e., } y \notin^{\langle \rangle} t) \end{cases}$

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 $M = \{m_1, \dots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \dots, m_n \rangle$. The order of the elements does not matter; one is chosen arbitrarily.

Definition 40 (Dictionaries). A dictionary over X and Y is a term of the form

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

where $k_1, \dots, k_n \in X$, $v_1, \dots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \dots, n\}$, an *element* of the dictionary with key k_i and value v_i . We often write $[k_1: v_1, \dots, k_n: v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$. 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 41 (Operations on Dictionaries). Let $z = [k_1: v_1, k_2: v_2, \dots, k_n: v_n]$ be a dictionary with unique keys, i.e., $\forall i, j: k_i \neq k_j$. In addition, let t and v be terms. We define the following operations:

- $t \in z \iff \exists i \in \{1, \dots, n\}: k_i = t$
- $z[t] := \begin{cases} v_i & \text{if } \exists k_i \in z: t = k_i \\ \langle \rangle & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- $z - t := \begin{cases} [k_1: v_1, \dots, k_{i-1}: v_{i-1}, k_{i+1}: v_{i+1}, \dots, k_n: v_n] & \text{if } \exists k_i \in z: t = k_i \\ z & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- 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 $t = \langle t_1, \dots, t_n \rangle$, we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 42 (Pointers). A *pointer* is a sequence of non-negative integers. We write $\tau.\bar{p}$ for the application of the pointer \bar{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 $\bar{p} = \langle 3, 1 \rangle$, the subterm of τ at the position \bar{p} is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3, 1 \rangle = \tau.3.\bar{p} = \tau.3.3.1 = e$.

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

Definition 43 (Concatenation of Sequences). For a sequence $a = \langle a_1, \dots, a_i \rangle$ and a sequence $b = \langle b_1, b_2, \dots \rangle$, we define the *concatenation* as $a \cdot b := \langle a_1, \dots, a_i, b_1, b_2, \dots \rangle$.

Definition 44 (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 occurrence 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 45. A *URL* is a term of the form

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

with $protocol \in \{P, S\}$ (for **plain** (HTTP) and **secure** (HTTPS)), a domain $host \in \text{Doms}$, $path \in \mathbb{S}$, $parameters \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, 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 \perp . We sometimes also write URL_{path}^{host} to denote the URL $\langle \text{URL}, S, host, path, \langle \rangle, \perp \rangle$.

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 42):

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

2) Origins

Definition 46. An *origin* is a term of the form $\langle host, protocol \rangle$ with $host \in \text{Doms}$ and $protocol \in \{P, S\}$. We write Origins for the set of all origins.

Example 5. For example, $\langle F00, S \rangle$ is the HTTPS origin for the domain F00, while $\langle \text{BAR}, P \rangle$ is the HTTP origin for the domain BAR.

3) Cookies

Definition 47. 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 $value \in \mathcal{T}_{\mathcal{N}}$, $secure, session, httpOnly \in \{\top, \perp\}$. 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 Cookies for the set of all cookies and $\text{Cookies}'$ 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.¹⁰ 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 [8]) of a cookie is set (i.e., $name$ consists of two parts and $name.1 \equiv \text{__Host}$), 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 $\langle name, value \rangle$.

¹⁰Note that *secure* cookies can be set over unencrypted connections (c.f. RFC 6265).

4) HTTP Messages

Definition 48. An *HTTP request* is a term of the form shown in (16). An *HTTP response* is a term of the form shown in (17).

$$\langle \text{HTTPReq}, \text{nonce}, \text{method}, \text{host}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \quad (16)$$

$$\langle \text{HTTPResp}, \text{nonce}, \text{status}, \text{headers}, \text{body} \rangle \quad (17)$$

The components are defined as follows:

- $\text{nonce} \in \mathcal{N}$ serves to map each response to the corresponding request.
- $\text{method} \in \text{Methods}$ is one of the HTTP methods.
- $\text{host} \in \text{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $\text{path} \in \mathbb{S}$ indicates the resource path at the server side.
- $\text{status} \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard).
- $\text{parameters} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ contains URL parameters.
- $\text{headers} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ contains request/response headers. The dictionary elements are terms of one of the following forms:
 - $\langle \text{Origin}, o \rangle$ where o is an origin,
 - $\langle \text{Set-Cookie}, c \rangle$ where c is a sequence of cookies,
 - $\langle \text{Cookie}, c \rangle$ where $c \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ (note that in this header, only names and values of cookies are transferred),
 - $\langle \text{Location}, l \rangle$ where $l \in \text{URLs}$,
 - $\langle \text{Referer}, r \rangle$ where $r \in \text{URLs}$,
 - $\langle \text{Strict-Transport-Security}, \top \rangle$,
 - $\langle \text{Authorization}, \langle \text{username}, \text{password} \rangle \rangle$ where $\text{username}, \text{password} \in \mathbb{S}$ (this header models the ‘Basic’ HTTP Authentication Scheme, see [39]),
 - $\langle \text{ReferrerPolicy}, p \rangle$ where $p \in \{\text{noreferrer}, \text{origin}\}$.
- $\text{body} \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

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

Example 6 (HTTP Request and Response).

$$r := \langle \text{HTTPReq}, n_1, \text{POST}, \text{example.com}, \text{/show}, \langle \langle \text{index}, 1 \rangle \rangle, \quad (18)$$

$$[\text{Origin} : \langle \text{example.com}, \mathbb{S} \rangle], \langle \text{foo}, \text{bar} \rangle \rangle$$

$$s := \langle \text{HTTPResp}, n_1, 200, \langle \langle \text{Set-Cookie}, \langle \langle \text{SID}, \langle n_2, \perp, \perp, \top \rangle \rangle \rangle \rangle, \langle \text{somescript}, x \rangle \rangle \quad (19)$$

An HTTP POST request for the URL <http://example.com/show?index=1> is shown in (18), with an Origin header and a body that contains $\langle \text{foo}, \text{bar} \rangle$. A possible response is shown in (19), 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 E-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 49. An *encrypted HTTP request* is of the form $\text{enc}_a(\langle m, k' \rangle, k)$, where $k \in \text{terms}$, $k' \in \mathcal{N}$, and $m \in \text{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\text{enc}_s(m', k')$, where $m' \in \text{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses HTTPSRequests or HTTPSResponses, respectively.

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

Example 7.

$$\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}})) \quad (20)$$

$$\text{enc}_s(s, k') \quad (21)$$

The term (20) shows an encrypted request (with r as in (18)). It is encrypted using the public key $\text{pub}(k_{\text{example.com}})$. The term (21) is a response (with s as in (19)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (20).

5) DNS Messages

Definition 50. A *DNS request* is a term of the form $\langle \text{DNSResolve}, \text{domain}, \text{nonce} \rangle$ where $\text{domain} \in \text{Doms}$, $\text{nonce} \in \mathcal{N}$. We call the set of all DNS requests DNSRequests .

Definition 51. A *DNS response* is a term of the form $\langle \text{DNSResolved}, \text{domain}, \text{result}, \text{nonce} \rangle$ with $\text{domain} \in \text{Doms}$, $\text{result} \in \text{IPs}$, $\text{nonce} \in \mathcal{N}$. We call the set of all DNS responses DNSResponses .

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

C. Atomic Processes, Systems and Runs

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

Definition 52 (Generic Atomic Processes and Systems). A *generic atomic process* is a tuple

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

where $I^p \subseteq \text{IPs}$, $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^\nu} \wr \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (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 (η_1, η_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 53 (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 sequence¹¹ (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 54 (Processing Steps). A *processing step of the system* \mathcal{P} is of the form

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

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) $E_{\text{out}}^\nu \subseteq 2^{\mathcal{E}^\nu} \wr \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$ 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 occurrences 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 55 (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 $(S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

¹¹Here: Not in the sense of terms as defined earlier.

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 $\langle a, a, \text{TRIGGER} \rangle$ for each $a \in \text{IPs}$, 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 56 (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 $\{\text{enc}_a(\langle a, b, c \rangle, \text{pub}(k)), k\}$, i.e., $a \in d_\emptyset(\{\text{enc}_a(\langle a, b, c \rangle, \text{pub}(k)), k\})$.

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

Definition 57 (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 58 (Atomic Attacker Process). An (atomic) attacker process for a set of sender addresses $A \subseteq \text{IPs}$ 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 $E = \langle \langle a_1, f_1, m_1 \rangle, \dots, \langle a_n, f_n, m_n \rangle \rangle$ with $n \in \mathbb{N}$, $a_1, \dots, a_n \in \text{IPs}$, $f_1, \dots, f_n \in A$, $m_1, \dots, m_n \in d_{V_{\text{process}}}(\{e, s\})$.

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** $\langle \text{Constant}, x, y \rangle \equiv t$ **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 **Constant** $\equiv \pi_1(t)$ and if $|\langle \text{Constant}, x, y \rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

Placeholder	Usage
ν_1	Algorithm 30, new window nonces
ν_2	Algorithm 30, new HTTP request nonce
ν_3	Algorithm 30, lookup key for pending HTTP requests entry
ν_4	Algorithm 28, new HTTP request nonce (multiple lines)
ν_5	Algorithm 28, new subwindow nonce
ν_6	Algorithm 29, new HTTP request nonce
ν_7	Algorithm 29, new document nonce
ν_8	Algorithm 25, lookup key for pending DNS entry
ν_9	Algorithm 22, new window nonce
ν_{10}, \dots	Algorithm 28, replacement for placeholders in script output

Table II: List of placeholders used in browser algorithms.

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 33). Table II 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_{path}^d to describe the following URL term: $\langle \text{URL}, S, d, path, \langle \rangle \rangle$. If the domain d belongs to some distinguished process P and it is the only domain associated to this process, we may also write URL_{path}^P . For a (secure) origin $\langle d, S \rangle$ 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 origin_P .

G. Browsers

Here, we present the formal model of browsers.

1) Scripts

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

Definition 59 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 60 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_{\mathcal{L}} \times \mathcal{T}_{\mathcal{L}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{L}}, s' \in \mathcal{T}_{\mathcal{L}}(V_{\text{script}})$ 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 λ_1, \dots which the browser will replace by (otherwise unused) placeholders ν_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 61 (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 62. A *window* is a term of the form $w = \langle \text{nonce}, \text{documents}, \text{opener} \rangle$ with $\text{nonce} \in \mathcal{N}$, $\text{documents} \subset \langle \rangle$ Documents (defined below), $\text{opener} \in \mathcal{N} \cup \{\perp\}$ where $d.\text{active} = \top$ 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 $w.\text{activedocument}$ 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., $b.\text{opener} = a.\text{nonce}$.

Definition 63. A *document d* is a term of the form

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

where $\text{nonce} \in \mathcal{N}$, $\text{location} \in \text{URLs}$, $\text{headers} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, $\text{referrer} \in \text{URLs} \cup \{\perp\}$, $\text{script} \in \mathcal{T}_{\mathcal{N}}$, $\text{scriptstate} \in \mathcal{T}_{\mathcal{N}}$, $\text{scriptinputs} \in \mathcal{T}_{\mathcal{N}}$, $\text{subwindows} \subset \langle \rangle$ Windows, $\text{active} \in \{\top, \perp\}$. A *limited document* is a term of the form $\langle \text{nonce}, \text{subwindows} \rangle$ 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 $d.\text{origin}$ to denote the origin of the document, i.e., the term $\langle d.\text{location}.\text{host}, d.\text{location}.\text{protocol} \rangle \in \text{Origins}$.

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

Definition 64. For two window terms w and w' we write

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

if $w \in \langle \rangle w'.\text{activedocument}.\text{subwindows}$. 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 65. A reference for a normal HTTP(S) request is a sequence of the form $\langle \text{REQ}, \text{nonce} \rangle$, where nonce is a window reference. A reference for a XMLHttpRequest is a sequence of the form $\langle \text{XHR}, \text{nonce}, \text{xhrreference} \rangle$, 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 40.

Definition 66. The *set of states* $Z_{\text{webbrowser}}$ of a web browser atomic Dolev-Yao process consists of the terms of the form

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

with the subterms as follows:

- $\text{windows} \subset \langle \rangle$ Windows contains a list of window terms (modeling top-level windows, or browser tabs) which contain documents, which in turn can contain further window terms (iframes).
- $\text{ids} \subset \langle \rangle \mathcal{T}_{\mathcal{N}}$ is a list of identities that are owned by this browser (i.e., belong to the user of the browser).
- $\text{secrets} \in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$ 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.
- $\text{localStorage} \in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$ stores the data saved by scripts using the localStorage API (separated by origins).

- $sessionStorage \in [OR \times \mathcal{T}_{\mathcal{N}}]$ for $OR := \{\langle o, r \rangle \mid o \in \text{Origins}, r \in \mathcal{N}\}$ similar to `localStorage`, but the data in `sessionStorage` is additionally separated by top-level windows.
- $keyMapping \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$ maps domains to TLS encryption keys.
- $sts \subset^{\diamond}$ `Doms` stores the list of domains that the browser only accesses via TLS (strict transport security).
- $DNSAddress \in \text{IPs}$ defines the IP address of the DNS server.
- $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ contains one pairing per unanswered DNS query of the form $\langle reference, request, url \rangle$. In these pairings, $reference$ is an HTTP(S) request reference (as above), $request$ contains the HTTP(S) message that awaits DNS resolution, and url contains the URL of said HTTP request. The pairings in $pendingDNS$ are indexed by the DNS request/response nonce.
- $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ contains pairings of the form $\langle reference, request, url, key, f \rangle$ with $reference$, $request$, and url as in $pendingDNS$, key is the symmetric encryption key if HTTPS is used or \perp otherwise, and f is the IP address of the server to which the request was sent.
- $isCorrupted \in \{\perp, \text{FULLCORRUPT}, \text{CLOSECORRUPT}\}$ 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.
- $tlskeys \in [\text{Doms} \times \mathcal{N}]$ 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 , `Subwindows(s)` denotes the set of all pointers¹² to windows in the window list $s.\text{windows}$ and (recursively) the subwindows of their active documents. We exclude subwindows of inactive documents and their subwindows. With `Docs(s)` we denote the set of pointers to all active documents in the set of windows referenced by `Subwindows(s)`.

Definition 67. For a browser state s we denote by `Subwindows(s)` the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\diamond} s.\text{windows}$ there is a $\bar{p} \in \text{Subwindows}(s)$ such that $s.\bar{p} = w$. (2) For all $\bar{p} \in \text{Subwindows}(s)$, the active document d of the window $s.\bar{p}$ and every subwindow w of d there is a pointer $\bar{p}' \in \text{Subwindows}(s)$ such that $s.\bar{p}' = w$.

Given a browser state s , the set `Docs(s)` of pointers to active documents is the minimal set such that for every $\bar{p} \in \text{Subwindows}(s)$ with $s.\bar{p}.\text{activedocument} \neq \langle \rangle$, there exists a pointer $\bar{p}' \in \text{Docs}(s)$ with $s.\bar{p}' = s.\bar{p}.\text{activedocument}$.

By `Subwindows+(s)` and `Docs+(s)` we denote the respective sets that also include the inactive documents and their subwindows.

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

Definition 68. Let s be a browser state and d a document. By `Clean(s, d)` we denote the term that equals $s.\text{windows}$ 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 69. For a sequence of cookies (with pairwise different names) $oldcookies$, a sequence of cookies $newcookies$, and a string $protocol \in \{\text{P}, \text{S}\}$, the set `CookieMerge(oldcookies, newcookies, protocol)` is defined by the following algorithm: From $newcookies$ remove all cookies c that have $c.\text{content.httpOnly} \equiv \top$ or where $(c.\text{name}.1 \equiv __Host) \wedge ((protocol \equiv \text{P}) \vee (c.\text{secure} \equiv \perp))$. For any $c, c' \in^{\diamond} newcookies$, $c.\text{name} \equiv c'.\text{name}$, remove the cookie that appears left of the

¹²Recall the definition of a pointer in Definition 42.

Algorithm 22 Web Browser Model: Determine window for navigation.

```
1: function GETNAVIGABLEWINDOW( $\bar{w}$ ,  $window$ ,  $noreferrer$ ,  $s'$ )
2:   if  $window \equiv \_BLANK$  then  $\rightarrow$  Open a new window when  $\_BLANK$  is used
3:   if  $noreferrer \equiv \top$  then
4:     let  $w' := \langle \nu_9, \langle \rangle, \perp \rangle$ 
5:   else
6:     let  $w' := \langle \nu_9, \langle \rangle, s'.\bar{w}.nonce \rangle$ 
7:     let  $s'.windows := s'.windows + \langle \rangle w'$ 
       $\hookrightarrow$  and let  $\bar{w}'$  be a pointer to this new element in  $s'$ 
8:     return  $\bar{w}'$ 
9:   let  $\bar{w}' \leftarrow \text{NavigableWindows}(\bar{w}, s')$  such that  $s'.\bar{w}'.nonce \equiv window$ 
       $\hookrightarrow$  if possible; otherwise return  $\bar{w}$ 
10:  return  $\bar{w}'$ 
```

Algorithm 23 Web Browser Model: Determine same-origin window.

```
1: function GETWINDOW( $\bar{w}$ ,  $window$ ,  $s'$ )
2:   let  $\bar{w}' \leftarrow \text{Subwindows}(s')$  such that  $s'.\bar{w}'.nonce \equiv window$ 
       $\hookrightarrow$  if possible; otherwise return  $\bar{w}$ 
3:   if  $s'.\bar{w}'.activatedocument.origin \equiv s'.\bar{w}.activatedocument.origin$  then
4:     return  $\bar{w}'$ 
5:   return  $\bar{w}$ 
```

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_{old}, c_{new}) with $c_{old} \in \langle \rangle oldcookies$, $c_{new} \in \langle \rangle newcookies$, $c_{old}.name \equiv c_{new}.name$, add c_{new} to m if $c_{old}.content.httpOnly \equiv \perp$ and add c_{old} to m otherwise. The result of $\text{CookieMerge}(oldcookies, newcookies, protocol)$ 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 70. For a sequence of cookies (with pairwise different names) *oldcookies*, a cookie c , and a string $protocol \in \{P, S\}$ (denoting whether the HTTP response was received from an insecure or a secure origin), the sequence $\text{AddCookie}(oldcookies, c, protocol)$ is defined by the following algorithm: Let $m := oldcookies$. If $(c.name.1 \equiv _Host) \wedge \neg((protocol \equiv S) \wedge (c.secure \equiv \top))$, then return m , else: Remove any c' from m that has $c.name \equiv c'.name$. 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 [4], Section 5.1.4 for this definition.

Definition 71. The set $\text{NavigableWindows}(\bar{w}, s')$ is the set $\bar{W} \subseteq \text{Subwindows}(s')$ of pointers to windows that the active document in \bar{w} is allowed to navigate. The set \bar{W} is defined to be the minimal set such that for every $\bar{w}' \in \text{Subwindows}(s')$ the following is true:

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

b) *Functions:*

- The function `GETNAVIGABLEWINDOW` (Algorithm 22) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\bar{w}$ 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.

Algorithm 24 Web Browser Model: Cancel pending requests for given window.

```

1: function CANCELNAV(reference, s')
2:   remove all  $\langle \text{reference}, \text{req}, \text{url}, \text{key}, f \rangle$  from s'.pendingRequests for any req, url, key, f
3:   remove all  $\langle x, \langle \text{reference}, \text{message}, \text{url} \rangle \rangle$  from s'.pendingDNS
      $\hookrightarrow$  for any x, message, url
4:   return s'

```

Algorithm 25 Web Browser Model: Prepare headers, do DNS resolution, save message.

```

1: function HTTP_SEND(reference, message, url, origin, referrer, referrerPolicy, a, s')
2:   if message.host  $\in \langle \rangle$  s'.sts then
3:     let url.protocol := S
4:   let cookies :=  $\{ \langle c.\text{name}, c.\text{content.value} \rangle \mid c \in \langle \rangle s'.\text{cookies}[\text{message.host}]$ 
      $\hookrightarrow \wedge (c.\text{content.secure} \equiv \top \implies (\text{url.protocol} \equiv \text{S})) \}$ 
5:   let message.headers[Cookie] := cookies
6:   if origin  $\neq \perp$  then
7:     let message.headers[Origin] := origin
8:   if referrerPolicy  $\equiv$  no-referrer then
9:     let referrer :=  $\perp$ 
10:  if referrer  $\neq \perp$  then
11:    if referrerPolicy  $\equiv$  origin then
12:      let referrer :=  $\langle \text{URL}, \text{referrer.protocol}, \text{referrer.host}, /, \langle \rangle, \perp \rangle$ 
         $\rightarrow$  Referrer stripped down to origin.
13:    let referrer.fragment :=  $\perp$ 
         $\rightarrow$  Browsers do not send fragment identifiers in the Referer header.
14:    let message.headers[Referer] := referrer
15:  let s'.pendingDNS[ $\nu_s$ ] :=  $\langle \text{reference}, \text{message}, \text{url} \rangle$ 
16:  stop  $\langle \langle s'.\text{DNSaddress}, a, \langle \text{DNSResolve}, \text{message.host}, \nu_s \rangle \rangle \rangle$ , s'

```

– 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 23) takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.
- The function CANCELNAV (Algorithm 24) 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 25) 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 65. *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

Algorithm 26 Web Browser Model: Navigate a window backward.

```

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

```

Algorithm 27 Web Browser Model: Navigate a window forward.

```

1: function NAVFORWARD( $\overline{w'}$ , s')
2:   if  $\exists \bar{j} \in \mathbb{N}$  such that s'. $\overline{w'}$ .documents. $\bar{j}$ .active  $\equiv \top$ 
      $\hookrightarrow \wedge s'.\overline{w'}$ .documents. $(\bar{j} + 1) \in \text{Documents}$  then
3:     let s'. $\overline{w'}$ .documents. $\bar{j}$ .active :=  $\perp$ 
4:     let s'. $\overline{w'}$ .documents. $(\bar{j} + 1)$ .active :=  $\top$ 
5:     let s' := CANCELNAV(s'. $\overline{w'}$ .nonce, s')
6:   stop  $\langle \rangle$ , s'

```

Algorithm 28 Web Browser Model: Execute a script.

```
1: function RUNSCRIPT( $\bar{w}, \bar{d}, a, s'$ )
2:   let  $tree := \text{Clean}(s', s'.\bar{d})$ 
3:   let  $cookies := \langle \{ \langle c.name, c.content.value \rangle \mid c \in {}^\diamond s'.cookies [s'.\bar{d}.origin.host] \} \rangle$ 
    $\hookrightarrow \wedge c.content.httpOnly \equiv \perp$ 
    $\hookrightarrow \wedge (c.content.secure \equiv \top \implies (s'.\bar{d}.origin.protocol \equiv S)) \rangle$ 
4:   let  $tlw \leftarrow s'.windows$  such that  $tlw$  is the top-level window containing  $\bar{d}$ 
5:   let  $sessionStorage := s'.sessionStorage [(s'.\bar{d}.origin, tlw.nonce)]$ 
6:   let  $localStorage := s'.localStorage [s'.\bar{d}.origin]$ 
7:   let  $secrets := s'.secrets [s'.\bar{d}.origin]$ 
8:   let  $R := \text{script}^{-1}(s'.\bar{d}.script)$  if possible; otherwise stop
9:   let  $in := \langle tree, s'.\bar{d}.nonce, s'.\bar{d}.scriptstate, s'.\bar{d}.scriptinputs, cookies,$ 
    $\hookrightarrow localStorage, sessionStorage, s'.ids, secrets \rangle$ 
10:  let  $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}}), cookies' \leftarrow \text{Cookies}' , localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}}),$ 
    $\hookrightarrow sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}}), command \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}}),$ 
    $\hookrightarrow out := \langle state', cookies', localStorage', sessionStorage', command \rangle$ 
    $\hookrightarrow$  such that  $out := out^\lambda[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$  with  $(in, out^\lambda) \in R$ 
11:  let  $s'.cookies [s'.\bar{d}.origin.host] :=$ 
    $\hookrightarrow \langle \text{CookieMerge}(s'.cookies [s'.\bar{d}.origin.host], cookies', s'.\bar{d}.origin.protocol) \rangle$ 
12:  let  $s'.localStorage [s'.\bar{d}.origin] := localStorage'$ 
13:  let  $s'.sessionStorage [(s'.\bar{d}.origin, tlw.nonce)] := sessionStorage'$ 
14:  let  $s'.\bar{d}.scriptstate := state'$ 
15:  let  $referrer := s'.\bar{d}.location$ 
16:  let  $referrerPolicy := s'.\bar{d}.headers[\text{ReferrerPolicy}]$ 
17:  let  $docorigin := s'.\bar{d}.origin$ 
18:  switch  $command$  do
19:    case  $\langle \text{HREF}, url, hrefwindow, noreferrer \rangle$ 
20:      let  $w' := \text{GETNAVIGABLEWINDOW}(\bar{w}, hrefwindow, noreferrer, s')$ 
21:      let  $reference := \langle \text{REQ}, s'.w'.nonce \rangle$ 
22:      let  $req := \langle \text{HTTPReq}, \nu_4, \text{GET}, url.host, url.path, url.parameters, \langle \rangle, \langle \rangle \rangle$ 
23:      if  $noreferrer \equiv \top$  then
24:        let  $referrerPolicy := noreferrer$ 
25:      let  $s' := \text{CANCELNAV}(reference, s')$ 
26:      call  $\text{HTTP\_SEND}(reference, req, url, \perp, referrer, referrerPolicy, a, s')$ 
27:    case  $\langle \text{IFRAME}, url, window \rangle$ 
28:      if  $window \equiv \_SELF$  then
29:        let  $w' := \bar{w}$ 
30:      else
31:        let  $w' := \text{GETWINDOW}(\bar{w}, window, s')$ 
32:        let  $req := \langle \text{HTTPReq}, \nu_4, \text{GET}, url.host, url.path, url.parameters, \langle \rangle, \langle \rangle \rangle$ 
33:        let  $w' := \langle \nu_5, \langle \rangle, \perp \rangle$ 
34:        let  $s'.w'.activedocument.subwindows := s'.w'.activedocument.subwindows + {}^\diamond w'$ 
35:        call  $\text{HTTP\_SEND}(\langle \text{REQ}, \nu_5 \rangle, req, url, \perp, referrer, referrerPolicy, a, s')$ 
```

→ Algorithm continues on next page.

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 26) and NAVFORWARD (Algorithm 27), 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 28) performs a script execution step of the script in the document $s'.\bar{d}$ (which is part of the window $s'.\bar{w}$). 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 29) 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 65. *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:

```

36:  case ⟨FORM, url, method, data, hrefwindow⟩
37:  if method ∉ {GET, POST} then
38:    stop
39:  let  $\bar{w}' := \text{GETNAVIGABLEWINDOW}(\bar{w}, hrefwindow, \perp, s')$ 
40:  let reference := ⟨REQ,  $s'.\bar{w}'$ .nonce⟩
41:  if method = GET then
42:    let body := ⟨⟩
43:    let parameters := data
44:    let origin :=  $\perp$ 
45:  else
46:    let body := data
47:    let parameters := url.parameters
48:    let origin := docorigin
49:  let req := ⟨HTTPReq,  $\nu_4$ , method, url.host, url.path, parameters, ⟨⟩, body⟩
50:  let  $s' := \text{CANCELNAV}(\text{reference}, s')$ 
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⟩ ∉(1)  $s'.\text{cibaBindingMessages}$  then
54:    stop
55:  if method ∉ {GET, POST} then
56:    stop
57:  let  $\bar{w}' := \text{GETNAVIGABLEWINDOW}(\bar{w}, hrefwindow, \perp, s')$ 
58:  let reference := ⟨REQ,  $s'.\bar{w}'$ .nonce⟩
59:  if method = GET then
60:    let body := ⟨⟩
61:    let parameters := data
62:    let origin :=  $\perp$ 
63:  else
64:    let body := data
65:    let parameters := url.parameters
66:    let origin := docorigin
67:  let req := ⟨HTTPReq,  $\nu_4$ , method, url.host, url.path, parameters, ⟨⟩, body⟩
68:  let  $s' := \text{CANCELNAV}(\text{reference}, s')$ 
69:  call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a,  $s'$ )
70:  case ⟨SETSCRIPT, window, script⟩
71:  let  $\bar{w}' := \text{GETWINDOW}(\bar{w}, window, s')$ 
72:  let  $s'.\bar{w}'$ .activedocument.script := script
73:  stop ⟨⟩,  $s'$ 
74:  case ⟨SETSCRIPTSTATE, window, scriptstate⟩
75:  let  $\bar{w}' := \text{GETWINDOW}(\bar{w}, window, s')$ 
76:  let  $s'.\bar{w}'$ .activedocument.scriptstate := scriptstate
77:  stop ⟨⟩,  $s'$ 
78:  case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
79:  if method ∈ {CONNECT, TRACE, TRACK} ∨  $\text{xhrreference} \notin V_{\text{process}} \cup \{\perp\}$  then
80:    stop
81:  if  $\text{url.host} \neq \text{docorigin.host} \vee \text{url.protocol} \neq \text{docorigin.protocol}$  then
82:    stop
83:  if method ∈ {GET, HEAD} then
84:    let data := ⟨⟩
85:    let origin :=  $\perp$ 
86:  else
87:    let origin := docorigin
88:  let req := ⟨HTTPReq,  $\nu_4$ , method, url.host, url.path, url.parameters, ⟨⟩, data⟩
89:  let reference := ⟨XHR,  $s'.\bar{d}$ .nonce, xhrreference⟩
90:  call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a,  $s'$ )
91:  case ⟨BACK, window⟩
92:  let  $\bar{w}' := \text{GETNAVIGABLEWINDOW}(\bar{w}, window, \perp, s')$ 
93:  call NAVBACK( $\bar{w}'$ ,  $s'$ )

```

→ Algorithm continues on next page.

```

94:   case ⟨FORWARD, window⟩
95:     let  $w' := \text{GETNAVIGABLEWINDOW}(\bar{w}, \text{window}, \perp, s')$ 
96:     call NAVFORWARD( $w', s'$ )
97:   case ⟨CLOSE, window⟩
98:     let  $w' := \text{GETNAVIGABLEWINDOW}(\bar{w}, \text{window}, \perp, s')$ 
99:     remove  $s'.w'$  from the sequence containing it
100:    stop  $\langle \rangle, s'$ 
101:   case ⟨POSTMESSAGE, window, message, origin⟩
102:     let  $w' \leftarrow \text{Subwindows}(s')$  such that  $s'.w'.nonce \equiv \text{window}$ 
103:     if  $\exists \bar{j} \in \mathbb{N}$  such that  $s'.w'.documents.\bar{j}.active \equiv \top$ 
104:        $\hookrightarrow \wedge (\text{origin} \neq \perp \implies s'.w'.documents.\bar{j}.origin \equiv \text{origin})$  then
105:         let  $s'.w'.documents.\bar{j}.scriptinputs := s'.w'.documents.\bar{j}.scriptinputs$ 
106:          $\hookrightarrow + \langle \rangle \langle \text{POSTMESSAGE}, s'.w'.nonce, \text{docorigin}, \text{message} \rangle$ 
107:     stop  $\langle \rangle, s'$ 

```

Definition 72. The pair $((\langle a, f, m \rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 30 (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 73 (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 74 (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 66) with the following properties:

- $s_0^p.\text{windows} \equiv \langle \rangle$
- $s_0^p.\text{ids} \subset \langle \rangle \mathcal{T}_{\mathcal{N}}$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- $s_0^p.\text{secrets} \in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- $s_0^p.\text{cookies} \equiv \langle \rangle$
- $s_0^p.\text{localStorage} \equiv \langle \rangle$
- $s_0^p.\text{sessionStorage} \equiv \langle \rangle$
- $s_0^p.\text{keyMapping} \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- $s_0^p.\text{sts} \equiv \langle \rangle$
- $s_0^p.\text{DNSaddress} \in \text{IPs}$ (note that this includes the possibility of using an attacker-controlled address)
- $s_0^p.\text{pendingDNS} \equiv \langle \rangle$
- $s_0^p.\text{pendingRequests} \equiv \langle \rangle$
- $s_0^p.\text{isCorrupted} \equiv \perp$
- $s_0^p.\text{cibaBindingMessages} \equiv \langle \rangle$
- $s_0^p.\text{tlskeys} \equiv \text{tlskeys}^p$ (see Appendix A-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 31): The state of a document contains a term, say *scriptinputs*, which records the input this document has obtained so far (via XHRs and postMessages). If the script of the document is activated, it will typically need to pick one input message from *scriptinputs* and record which input it has already processed. For this purpose, the function *CHOOSEINPUT*($s', \text{scriptinputs}$) 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).

Algorithm 29 Web Browser Model: Process an HTTP response.

```
1: function PROCESSRESPONSE(response, reference, request, requestUrl, a, f, s')
2:   if Set-Cookie  $\in$  response.headers then
3:     for each  $c \in \langle \rangle$  response.headers [Set-Cookie],  $c \in$  Cookies do
4:       let s'.cookies [request.host]
          $\hookrightarrow$  := AddCookie(s'.cookies [request.host], c, requestUrl.protocol)
5:   if Strict-Transport-Security  $\in$  response.headers  $\wedge$  requestUrl.protocol  $\equiv$  S then
6:     let s'.sts := s'.sts +  $\langle \rangle$  request.host
7:   if Referer  $\in$  request.headers then
8:     let referrer := request.headers[Referer]
9:   else
10:    let referrer :=  $\perp$ 
11:   if Location  $\in$  response.headers  $\wedge$  response.status  $\in$  {303, 307} then
12:     let url := response.headers [Location]
13:     if url.fragment  $\equiv$   $\perp$  then
14:       let url.fragment := requestUrl.fragment
15:     let method' := request.method
16:     let body' := request.body
17:     if Origin  $\in$  request.headers
          $\hookrightarrow$   $\wedge$  request.headers[Origin]  $\neq$   $\diamond$ 
          $\hookrightarrow$   $\wedge$  (url.host, url.protocol)  $\equiv$  (request.host, requestUrl.protocol)
          $\hookrightarrow$   $\vee$  (request.host, requestUrl.protocol)  $\equiv$  request.headers[Origin] then
18:       let origin := request.headers[Origin]
19:     else
20:       let origin :=  $\diamond$ 
21:     if response.status  $\equiv$  303  $\wedge$  request.method  $\notin$  {GET, HEAD} then
22:       let method' := GET
23:       let body' :=  $\langle \rangle$ 
24:     if  $\exists \bar{w} \in$  Subwindows(s') such that s'. $\bar{w}$ .nonce  $\equiv$   $\pi_2$ (reference) then  $\rightarrow$  Do not redirect XHRs.
25:       let req :=  $\langle$  HTTPReq,  $\nu_6$ , method', url.host, url.path, url.parameters,  $\langle \rangle$ , body'  $\rangle$ 
26:       let referrerPolicy := response.headers[RefererPolicy]
27:       call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s')
28:     else
29:       stop  $\langle \rangle$ , s'
30:   switch  $\pi_1$ (reference) do
31:     case REQ
32:       let  $\bar{w} \leftarrow$  Subwindows(s') such that s'. $\bar{w}$ .nonce  $\equiv$   $\pi_2$ (reference) if possible;
          $\hookrightarrow$  otherwise stop  $\rightarrow$  normal response
33:       if response.body  $\not\sim$   $\langle *, * \rangle$  then
34:         stop  $\langle \rangle$ , s'
35:       let script :=  $\pi_1$ (response.body)
36:       let scriptstate :=  $\pi_2$ (response.body)
37:       let d :=  $\langle \nu_7$ , requestUrl, response.headers, referrer, script, scriptstate,  $\langle \rangle$ ,  $\langle \rangle$ ,  $\top$   $\rangle$ 
38:       if s'. $\bar{w}$ .documents  $\equiv$   $\langle \rangle$  then
39:         let s'. $\bar{w}$ .documents :=  $\langle d \rangle$ 
40:       else
41:         let  $\bar{i} \leftarrow$   $\mathbb{N}$  such that s'. $\bar{w}$ .documents. $\bar{i}$ .active  $\equiv$   $\top$ 
42:         let s'. $\bar{w}$ .documents. $\bar{i}$ .active :=  $\perp$ 
43:         remove s'. $\bar{w}$ .documents. $(\bar{i} + 1)$  and all following documents
          $\hookrightarrow$  from s'. $\bar{w}$ .documents
44:         let s'. $\bar{w}$ .documents := s'. $\bar{w}$ .documents +  $\langle \rangle$  d
45:       stop  $\langle \rangle$ , s'
46:     case XHR
47:       let  $\bar{w} \leftarrow$  Subwindows(s'),  $\bar{d}$  such that s'. $\bar{d}$ .nonce  $\equiv$   $\pi_2$ (reference)
          $\hookrightarrow$   $\wedge$  s'. $\bar{d}$  = s'. $\bar{w}$ .activedocument if possible; otherwise stop
          $\rightarrow$  process XHR response
48:       let headers := response.headers – Set-Cookie
49:       let s'. $\bar{d}$ .scriptinputs := s'. $\bar{d}$ .scriptinputs +  $\langle \rangle$ 
          $\langle$  XMLHTTPREQUEST, headers, response.body,  $\pi_3$ (reference)  $\rangle$ 
50:       stop  $\langle \rangle$ , s'
```

Algorithm 30 Web Browser Model: Main Algorithm.

```
Input:  $\langle a, f, m \rangle, s$ 
1: let  $s' := s$ 
2: if  $s.isCorrupted \neq \perp$  then
3:   let  $s'.pendingRequests := \langle m, s.pendingRequests \rangle \rightarrow$  Collect incoming messages
4:   let  $m' \leftarrow d_V(s')$ 
5:   let  $a' \leftarrow$  IPs
6:   stop  $\langle \langle a', a, m' \rangle \rangle, s'$ 
7: if  $m \equiv$  TRIGGER then  $\rightarrow$  A special trigger message.
8:   let  $switch \leftarrow \{script, urlbar, reload, forward, back\}$ 
9:   if  $switch \equiv script$  then  $\rightarrow$  Run some script.
10:    let  $\bar{w} \leftarrow$  Subwindows( $s'$ ) such that  $s'.\bar{w}.documents \neq \langle \rangle$ 
11:     $\hookrightarrow$  if possible; otherwise stop  $\rightarrow$  Pointer to some window.
12:    let  $\bar{d} := \bar{w} +^{(\diamond)}$  activedocument
13:    call RUNSCRIPT( $\bar{w}, \bar{d}, a, s'$ )
14:  else if  $switch \equiv urlbar$  then  $\rightarrow$  Create some new request.
15:    let  $newwindow \leftarrow \{\top, \perp\}$ 
16:    if  $newwindow \equiv \top$  then  $\rightarrow$  Create a new window.
17:      let  $windownonce := \nu_1$ 
18:      let  $w' := \langle windownonce, \langle \rangle, \perp \rangle$ 
19:      let  $s'.windows := s'.windows +^{(\diamond)}$   $w'$ 
20:    else  $\rightarrow$  Use existing top-level window.
21:      let  $tlw \leftarrow \mathbb{N}$  such that  $s'.tlw.documents \neq \langle \rangle$ 
22:       $\hookrightarrow$  if possible; otherwise stop  $\rightarrow$  Pointer to some top-level window.
23:      let  $windownonce := s'.tlw.nonce$ 
24:      let  $protocol \leftarrow \{P, S\}$ 
25:      let  $host \leftarrow$  Doms
26:      let  $path \leftarrow \mathbb{S}$ 
27:      let  $fragment \leftarrow \mathbb{S}$ 
28:      let  $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$ 
29:      let  $body := \langle \rangle$ 
30:      let  $startciba \leftarrow \{\top, \perp\}$ 
31:      if  $startciba \equiv \top$  then
32:        let  $body[authServ] \leftarrow$  Doms
33:        let  $body[identity] \leftarrow s'.ids$ 
34:      let  $url := \langle URL, protocol, host, path, parameters, fragment \rangle$ 
35:      let  $req := \langle HTTPReq, \nu_2, GET, host, path, parameters, \langle \rangle, body \rangle$ 
36:      call HTTP_SEND( $\langle REQ, windownonce \rangle, req, url, \perp, \perp, \perp, a, s'$ )
37:    else if  $switch \equiv reload$  then  $\rightarrow$  Reload some document.
38:      let  $\bar{w} \leftarrow$  Subwindows( $s'$ ) such that  $s'.\bar{w}.documents \neq \langle \rangle$ 
39:       $\hookrightarrow$  if possible; otherwise stop  $\rightarrow$  Pointer to some window.
40:      let  $url := s'.\bar{w}.activedocument.location$ 
41:      let  $req := \langle HTTPReq, \nu_2, GET, url.host, url.path, url.parameters, \langle \rangle, \langle \rangle \rangle$ 
42:      let  $referrer := s'.\bar{w}.activedocument.referrer$ 
43:      let  $s' := CANCELNAV(s'.\bar{w}.nonce, s')$ 
44:      call HTTP_SEND( $\langle REQ, s'.\bar{w}.nonce \rangle, req, url, \perp, referrer, \perp, a, s'$ )
45:    else if  $switch \equiv forward$  then
46:      let  $\bar{w} \leftarrow$  Subwindows( $s'$ ) such that  $s'.\bar{w}.documents \neq \langle \rangle$ 
47:       $\hookrightarrow$  if possible; otherwise stop  $\rightarrow$  Pointer to some window.
48:      call NAVFORWARD( $\bar{w}, s'$ )
49:    else if  $switch \equiv back$  then
50:      let  $\bar{w} \leftarrow$  Subwindows( $s'$ ) such that  $s'.\bar{w}.documents \neq \langle \rangle$ 
51:       $\hookrightarrow$  if possible; otherwise stop  $\rightarrow$  Pointer to some window.
52:      call NAVBACK( $\bar{w}, s'$ )
53:  else if  $m \equiv FULLCORRUPT$  then  $\rightarrow$  Request to corrupt browser
54:    let  $s'.isCorrupted := FULLCORRUPT$ 
55:    stop  $\langle \rangle, s'$ 
56:  else if  $m \equiv CLOSECORRUPT$  then  $\rightarrow$  Close the browser
57:    let  $s'.secrets := \langle \rangle$ 
58:    let  $s'.windows := \langle \rangle$ 
59:    let  $s'.pendingDNS := \langle \rangle$ 
60:    let  $s'.pendingRequests := \langle \rangle$ 
61:    let  $s'.sessionStorage := \langle \rangle$ 
62:    let  $s'.cookies \subset^{(\diamond)}$  Cookies such that
63:     $\hookrightarrow (c \in^{(\diamond)} s'.cookies) \iff (c \in^{(\diamond)} s.cookies \wedge c.content.session \equiv \perp)$ 
64:    let  $s'.isCorrupted := CLOSECORRUPT$ 
65:    stop  $\langle \rangle, s'$ 
```

```

60: else if  $\exists \langle \text{reference}, \text{request}, \text{url}, \text{key}, f \rangle \in {}^\diamond s'.\text{pendingRequests}$  such that
     $\hookrightarrow \pi_1(\text{dec}_s(m, \text{key})) \equiv \text{HTTPResp}$  then  $\rightarrow$  Encrypted HTTP response
61:   let  $m' := \text{dec}_s(m, \text{key})$ 
62:   if  $m'.\text{nonce} \neq \text{request}.\text{nonce}$  then
63:     stop
64:   remove  $\langle \text{reference}, \text{request}, \text{url}, \text{key}, f \rangle$  from  $s'.\text{pendingRequests}$ 
65:   if  $\text{binding\_message} \in {}^\diamond m'.\text{body}$  then
66:     let  $s'.\text{cibaBindingMessages} := s'.\text{cibaBindingMessages} + {}^\diamond \langle \text{request}.\text{host}, m'.\text{body}[\text{binding\_message}] \rangle$ 
67:     call  $\text{PROCESSRESPONSE}(m', \text{reference}, \text{request}, \text{url}, a, f, s')$ 
68: else if  $\pi_1(m) \equiv \text{HTTPResp} \wedge \exists \langle \text{reference}, \text{request}, \text{url}, \perp, f \rangle \in {}^\diamond s'.\text{pendingRequests}$  such that
     $\hookrightarrow m.\text{nonce} \equiv \text{request}.\text{nonce}$  then  $\rightarrow$  Plain HTTP Response
69:   remove  $\langle \text{reference}, \text{request}, \text{url}, \perp, f \rangle$  from  $s'.\text{pendingRequests}$ 
70:   call  $\text{PROCESSRESPONSE}(m, \text{reference}, \text{request}, \text{url}, a, f, s')$ 
71: else if  $m \in \text{DNSResponses}$  then  $\rightarrow$  Successful DNS response
72:   if  $m.\text{nonce} \notin s.\text{pendingDNS} \vee m.\text{result} \notin \text{IPs}$ 
     $\hookrightarrow \forall m.\text{domain} \neq s.\text{pendingDNS}[m.\text{nonce}].\text{request}.\text{host}$  then
73:     stop
74:   let  $\langle \text{reference}, \text{message}, \text{url} \rangle := s.\text{pendingDNS}[m.\text{nonce}]$ 
75:   if  $\text{url}.\text{protocol} \equiv \text{S}$  then
76:     let  $s'.\text{pendingRequests} := s'.\text{pendingRequests}$ 
     $\hookrightarrow + {}^\diamond \langle \text{reference}, \text{message}, \text{url}, \nu_3, m.\text{result} \rangle$ 
77:     let  $\text{message} := \text{enc}_a(\langle \text{message}, \nu_3 \rangle, s'.\text{keyMapping}[\text{message}.\text{host}])$ 
78:   else
79:     let  $s'.\text{pendingRequests} := s'.\text{pendingRequests}$ 
     $\hookrightarrow + {}^\diamond \langle \text{reference}, \text{message}, \text{url}, \perp, m.\text{result} \rangle$ 
80:   let  $s'.\text{pendingDNS} := s'.\text{pendingDNS} - m.\text{nonce}$ 
81:   stop  $\langle \langle m.\text{result}, a, \text{message} \rangle \rangle, s'$ 
82: else if  $\exists m_{\text{dec}}, k, k', \text{inDomain}$  such that  $\langle m_{\text{dec}}, k \rangle \equiv \text{dec}_a(m, k') \wedge \langle \text{inDomain}, k' \rangle \in s.\text{tlskeys}$  then
     $\rightarrow$  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
83:   let  $n, \text{method}, \text{path}, \text{parameters}, \text{headers}, \text{body}$  such that
     $\hookrightarrow \langle \text{HTTPReq}, n, \text{method}, \text{inDomain}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$ 
     $\hookrightarrow$  if possible; otherwise stop
84:   if  $\text{path} \neq /start-ciba-authentication$  then stop
85:   let  $\text{newwindow} \leftarrow \{\top, \perp\}$ 
86:   if  $\text{newwindow} \equiv \top$  then  $\rightarrow$  Create a new window.
87:     let  $\text{windownonce} := \nu_1$ 
88:     let  $w' := \langle \text{windownonce}, \langle \rangle, \perp \rangle$ 
89:     let  $s'.\text{windows} := s'.\text{windows} + {}^\diamond w'$ 
90:   else  $\rightarrow$  Use existing top-level window.
91:     let  $\text{tlw} \leftarrow \mathbb{N}$  such that  $s'.\text{tlw}.\text{documents} \neq \langle \rangle$ 
     $\hookrightarrow$  if possible; otherwise stop  $\rightarrow$  Pointer to some top-level window.
92:     let  $\text{windownonce} := s'.\text{tlw}.\text{nonce}$ 
93:     let  $\text{url} := \text{body}[\text{ciba\_url}]$ 
94:     let  $\text{req} := \langle \text{HTTPReq}, \nu_{\text{ciba\_req}}, \text{POST}, \text{url}.\text{host}, \varepsilon, \langle \rangle, \langle \rangle, \text{body} \rangle$ 
95:     call  $\text{HTTP\_SEND}(\langle \text{REQ}, \text{windownonce} \rangle, \text{req}, \text{url}, \perp, \perp, \perp, a, s')$ 
96: stop

```

Algorithm 31 Function to retrieve an unhandled input message for a script.

```

1: function  $\text{CHOOSEINPUT}(s', \text{scriptinputs})$ 
2:   let  $iid$  such that  $iid \in \{1, \dots, |\text{scriptinputs}|\} \wedge iid \notin {}^\diamond s'.\text{handledInputs}$  if possible;
     $\hookrightarrow$  otherwise return  $(\perp, s')$ 
3:   let  $\text{input} := \pi_{iid}(\text{scriptinputs})$ 
4:   let  $s'.\text{handledInputs} := s'.\text{handledInputs} + {}^\diamond iid$ 
5:   return  $(\text{input}, s')$ 

```

Algorithm 32 Function to extract the first script input message matching a specific pattern.

```

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

```

b) *CHOOSEFIRSTINPUTPAT* (Algorithm 32): Similar to the function CHOOSEINPUT above, we define the function CHOOSEFIRSTINPUTPAT. This function takes the term *scriptinputs*, which as above records the input this document has obtained so far (via XHRs and postMessages, append-only), and a pattern. If called, this function chooses the first message in *scriptinputs* that matches *pattern* and returns it. This function is typically used in places, where a script only processes the first message that matches the pattern. Hence, we omit recording the usage of an input.

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

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

e) *SUBWINDOWS*: This function takes a term *tree* and a document nonce *docnonce* as input just as the function above. If *docnonce* is not a reference to a document contained in *tree*, then SUBWINDOWS(*tree*, *docnonce*) returns $\langle \rangle$. Otherwise, let $\langle \text{docnonce}, \text{location}, \langle \rangle, \text{referrer}, \text{script}, \text{scriptstate}, \text{scriptinputs}, \text{subwindows}, \text{active} \rangle$ denote the subterm of *tree* corresponding to the document referred to by *docnonce*. Then, 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 GETORIGIN(*tree*, *docnonce*) is used. This function searches for the document with the identifier *docnonce* in the (cleaned) tree *tree* of the browser's windows and documents. It returns the origin *o* of the document. If no document with nonce *docnonce* is found in the tree *tree*, \diamond is returned.

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

J. DNS Servers

Definition 75. 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

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

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 33.

Algorithm 33 Relation of a DNS server R^d .

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

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 76. A web system $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ 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 $p \in \text{Web} \cup \text{Net}$ is an attacker process for some set of sender addresses $A \subseteq \text{IPs}$. For a web attacker $p \in \text{Web}$, 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 $p' \neq p, p' \in \text{Hon} \cup \text{Web}$. 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 $p \in \text{Hon}$ 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 $\text{script}(s)$.

Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \text{TRIGGER} \rangle$ 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 77 (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: $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \text{IPs}$ (containing the IP address of a DNS server), $keyMapping \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$ (containing a mapping from domains to public keys), $tlskeys \in [\text{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $corrupt \in \mathcal{T}_{\mathcal{N}}$ (either \perp if the server is not corrupted, or an arbitrary term otherwise).

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

Let ν_{n0} and ν_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. We now define the default functions of the generic web server in Algorithms 34–38, and the main relation in Algorithm 39.

Algorithm 34 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** $s'.pendingDNS[\nu_{n0}] := \langle reference, message \rangle$
3: **stop** $\langle \langle s'.DNSaddress, a, \langle \text{DNSResolve}, message.host, \nu_{n0} \rangle \rangle \rangle, s'$

Algorithm 35 Generic HTTPS Server Model: Default HTTPS response handler.

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

Algorithm 36 Generic HTTPS Server Model: Default trigger event handler.

```

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

```

Algorithm 37 Generic HTTPS Server Model: Default HTTPS request handler.

```

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

```

Algorithm 38 Generic HTTPS Server Model: Default handler for other messages.

```

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

```

Algorithm 39 Generic HTTPS Server Model: Main relation of a generic HTTPS server

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

```

1: let  $s' := s$ 
2: if  $s'.\text{corrupt} \not\equiv \perp \vee m \equiv \text{CORRUPT}$  then
3:   let  $s'.\text{corrupt} := \langle \langle a, f, m \rangle, s'.\text{corrupt} \rangle$ 
4:   let  $m' \leftarrow d_V(s')$ 
5:   let  $a' \leftarrow \text{IPs}$ 
6:   stop  $\langle \langle a', a, m' \rangle \rangle, s'$ 
7: if  $\exists m_{\text{dec}}, k, k', \text{inDomain}$  such that  $\langle m_{\text{dec}}, k \rangle \equiv \text{dec}_a(m, k') \wedge \langle \text{inDomain}, k' \rangle \in \text{s.tlskeys}$  then
8:   let  $n, \text{method}, \text{path}, \text{parameters}, \text{headers}, \text{body}$  such that
    $\hookrightarrow \langle \text{HTTPReq}, n, \text{method}, \text{inDomain}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$ 
    $\hookrightarrow$  if possible; otherwise stop
9:   call PROCESS_HTTPS_REQUEST( $m_{\text{dec}}, k, a, f, s'$ )
10: else if  $m \in \text{DNSResponses}$  then  $\rightarrow$  Successful DNS response
11:   if  $m.\text{nonce} \notin s.\text{pendingDNS} \vee m.\text{result} \notin \text{IPs}$ 
    $\hookrightarrow \vee m.\text{domain} \neq s.\text{pendingDNS}[m.\text{nonce}].2.\text{host}$  then
12:     stop
13:   let  $\text{reference} := s.\text{pendingDNS}[m.\text{nonce}].1$ 
14:   let  $\text{request} := s.\text{pendingDNS}[m.\text{nonce}].2$ 
15:   let  $s'.\text{pendingRequests} := s'.\text{pendingRequests} + \langle \text{reference}, \text{request}, \nu_{n1}, m.\text{result} \rangle$ 
16:   let  $\text{message} := \text{enc}_a(\langle \text{request}, \nu_{n1} \rangle, s'.\text{keyMapping}[\text{request}.\text{host}])$ 
17:   let  $s'.\text{pendingDNS} := s'.\text{pendingDNS} - m.\text{nonce}$ 
18:   stop  $\langle \langle m.\text{result}, a, \text{message} \rangle \rangle, s'$ 
19: else if  $\exists \langle \text{reference}, \text{request}, \text{key}, f \rangle \in s'.\text{pendingRequests}$ 
    $\hookrightarrow$  such that  $\pi_1(\text{dec}_s(m, \text{key})) \equiv \text{HTTPResp}$  then  $\rightarrow$  Encrypted HTTP response
20:   let  $m' := \text{dec}_s(m, \text{key})$ 
21:   if  $m'.\text{nonce} \neq \text{request}.\text{nonce}$  then
22:     stop
23:   if  $m' \notin \text{HTTPResponses}$  then
24:     call PROCESS_OTHER( $m, a, f, s'$ )
25:   remove  $\langle \text{reference}, \text{request}, \text{key}, f \rangle$  from  $s'.\text{pendingRequests}$ 
26:   call PROCESS_HTTPS_RESPONSE( $m', \text{reference}, \text{request}, a, f, s'$ )
27: else if  $m \equiv \text{TRIGGER}$  then  $\rightarrow$  Process was triggered
28:   call PROCESS_TRIGGER( $a, s'$ )
29: else
30:   call PROCESS_OTHER( $m, a, f, s'$ )
31: stop

```

M. General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [18].

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

Definition 78 (Emitting Events). Given an atomic process p , an event e , and a finite run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \dots)$ we say that p emits e iff there is a processing step in ρ of the form

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

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

Definition 79. We say that a term t is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \rightarrow s_{i+1}$ of a run $\rho = (s_0, s_1, \dots)$) 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 80. We say that a set of processes P leaks a term t (in a processing step $s_i \rightarrow s_{i+1}$) to a set of processes P' if there exists a message m that is emitted (in $s_i \rightarrow s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \rightarrow s_{i+1}$. 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 81. We say that a DY process p created a message m in a processing step

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

of a run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ 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

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

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

Definition 82. 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 29).

Definition 83. 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 84. 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 $t = f(t_1, \dots, t_n)$, for $f \in \Sigma$ and $t_i \in \mathcal{T}_N(X)$ ($i \in \{1, \dots, n\}$), then $f = \text{pub}$ or for all t_i , k appears only as a public key in t_i .

Definition 85. We say that a script initiated a request r if a browser triggered the script (in Line 10 of Algorithm 28) 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 86. 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 39).

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

Lemma 43. If in the processing step $s_i \rightarrow s_{i+1}$ of a run ρ of \mathcal{MS} an honest browser b

- (I) emits an HTTPS request of the form

$$m = \text{enc}_a(\langle \text{req}, k \rangle, \text{pub}(k'))$$

- (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{MS} where any party except for u knows k' , thus no one except for u can decrypt m to obtain *req*.
- (2) If there is a processing step $s_j \rightarrow s_{j+1}$ where the browser b leaks k to $\mathcal{W} \setminus \{u, b\}$ there is a processing step $s_h \rightarrow s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ 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 $\text{pub}(k')$ in the browsers' keymapping $s_0.\text{keyMapping}$ (in its initial state).

- (4) If b accepts a response (say, m') to m in a processing step $s_j \rightarrow s_{j+1}$ 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 $\text{pub}(x)$, 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 $s_j \rightarrow s_{j+1}$ 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 30) 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 30). 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 30), a host header is always contained in HTTP requests by browsers. From Line 77 of Algorithm 30 we can see that the encryption key for the request req was chosen using the host header of the message. It is chosen from the *keyMapping* in the browser's state, which is never changed during ρ . This proves (3).

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

Corollary 1. In the situation of Lemma 43, 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 44. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\text{windows}$ with the origin $\langle \text{dom}, S \rangle$ where $\text{dom} \in \text{Domain}$, and $S_i(b).\text{keyMapping}[\text{dom}] \equiv \text{pub}(k)$ with $k \in \mathcal{K}$ 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 29) 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 29. 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 30. In particular, the request was an HTTPS request iff a (symmetric) key was added to the information in *pendingRequests*. When receiving the response to req , it is checked against that information and accepted only if it is encrypted with the proper key and contains the same nonce as the request (say, n). Only then the protocol part of the origin of the newly created document becomes S. The domain part of the origin (in our case dom) is taken directly from the *pendingRequests* and is thus guaranteed to be unaltered.

From Line 77 of Algorithm 30 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 $S_i(b).\text{keyMapping}[\text{dom}] = S_0(b).\text{keyMapping}[\text{dom}] \equiv \text{pub}(k)$ 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 43 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 45. If in a processing step $s_i \rightarrow s_{i+1}$ of a run ρ of \mathcal{US} an honest browser b issues an HTTP(S) request with the Origin header value $\langle \text{dom}, S \rangle$ where $S_i(b).\text{keyMapping}[\text{dom}] \equiv \text{pub}(k)$ with $k \in \mathcal{K}$ 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 29) 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 25), 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 28
- Line 90 of Algorithm 28
- Line 27 of Algorithm 29

■

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 44 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 29 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 43, but is applied to the generic HTTPS server (instead of the web browser).

Lemma 46. If in the processing step $s_i \rightarrow s_{i+1}$ of a run ρ of \mathcal{WS} an honest instance s of the generic HTTPS server model (I) emits an HTTPS request of the form

$$m = \text{enc}_a(\langle \text{req}, k \rangle, \text{pub}(k'))$$

- (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 $\text{enc}_a(\langle \text{req}', k_2 \rangle, \text{pub}(k'))$, 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 $\text{enc}_a(\langle \text{req}', k_2 \rangle, \text{pub}(k'))$, u uses the symmetric key k_2 only for symmetrically encrypting HTTP responses (and in particular, k_2 is not part of a payload of any messages sent out by u),

then all of the following statements are true:

- (1) There is no state of \mathcal{WS} where any party except for u knows k' , thus no one except for u can decrypt m to obtain req .
- (2) If there is a processing step $s_j \rightarrow s_{j+1}$ where some process leaks k to $\mathcal{W} \setminus \{u, s\}$, there is a processing step $s_h \rightarrow s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ or the process s is corrupted in s_j .
- (3) The value of the host header in req is the domain that is assigned the public key $\text{pub}(k')$ in $S^0(s).\text{keyMapping}$ (i.e., in the initial state of s).
- (4) If s accepts a response (say, m') to m in a processing step $s_j \rightarrow s_{j+1}$ and s is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ prior to s_j , then u created the HTTPS response m' to the HTTPS request m , i.e., the nonce of the HTTP request req is not known to any atomic process p , except for the atomic processes s and u .

PROOF. (1) follows immediately from the preconditions. The proof is the same as for Lemma 43:

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 $\text{pub}(x)$, the other processes can never derive k' .

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

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

The process s is honest in s_i . s emits HTTPS requests like m only in Line 18 of Algorithm 39:

- The message emitted in Line 3 of Algorithm 34 has a different message structure
- As s is honest, it does not send the message of Line 6 of Algorithm 39

- 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 39, 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 39, 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 39 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 39, 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 34, where a DNS request is sent, and in Line 18 of Algorithm 39, 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 39 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' .