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

PEDRAM HOSSEYNI, University of Stuttgart, Germany
RALF KÜSTERS, University of Stuttgart, Germany
TIM WÜRTELE, University of Stuttgart, Germany

FAPI 2.0 is a suite of Web protocols developed by the OpenID Foundation's FAPI Working Group (FAPI WG) for third-party data sharing and digital identity in high-risk environments. Even though the specifications are not completely finished, several important entities have started to adopt the FAPI 2.0 protocols, including Norway's national HelseID, Australia's Consumer Data Standards, as well as private companies like Authlete and Australia-based connectID; the predecessor FAPI 1.0 is in widespread use with millions of users.

The FAPI WG asked us to accompany the standardization of the FAPI 2.0 protocols with a formal security analysis to proactively identify vulnerabilities before widespread deployment and to provide formal security guarantees for the standards. In this paper, we report on our analysis and findings.

Our analysis is based on a detailed model of the Web infrastructure, the so-called Web Infrastructure Model (WIM), which we extend to be able to carry out our analysis of the FAPI 2.0 protocols including important extensions like FAPI-CIBA. Based on the (extended) WIM and formalizations of the security goals and attacker model laid out in the FAPI 2.0 specifications, we provide a formal model of the protocols and carry out a formal security analysis, revealing several attacks. We have worked with the FAPI WG to fix the protocols, resulting in several amendments to the specifications. With these changes in place, we have adjusted our protocol model and formally proved that the security properties hold true under the strong attacker model defined by the FAPI WG.

CCS Concepts: • **Security and privacy** → **Logic and verification**; **Web protocol security**; **Security protocols**; • **Networks** → *Network protocol design*; • **Information systems** → Browsers.

1 INTRODUCTION

Web-based authentication and authorization are ubiquitous. Many websites and applications can be used by logging in with a so-called Identity Provider, for instance, "Login with Google" or "Login with Facebook", generally dubbed "social login", or more generally, *Single Sign-On (SSO)*. There are typically three parties involved for SSO: (i) the user/the user's browser, (ii) the *identity provider (IdP)*, also called *authorization server (AS)*, and (iii) the *relying party (RP)*, also called *client*. For example, a user may log in at TripAdvisor (RP/client) with its account at Facebook (IdP/AS). In the following, we will use the terms "authorization server" and "client", in line with the FAPI 2.0 specifications. It is also possible to authorize applications, including websites, but also IoT devices, such as routers and smart TVs, to access resources managed by the AS. In this setting, a fourth party is involved: the *resource server (RS)*, which stores the actual resources. Such resources include email addresses [41, 73], documents/calendars/pictures/movies stored in the cloud [20, 42], YouTube accounts [43], or development repositories [41]. For example, a smart TV can be authorized to access a user's YouTube account. A widely used protocol family for these authorization and authentication use cases are the OAuth 2.0 and OpenID Connect protocols [101, 102].

While plain OAuth 2.0 and OpenID Connect are suitable for typical low-risk use cases like social login, many use cases have emerged in high-risk settings for both authorization and authentication scenarios: Third-party services can be authorized to, for example, access bank transaction histories

*Technical Report. The corresponding paper appears in *ACM Transactions on Privacy and Security*.

Authors' addresses: Pedram Hosseyni, University of Stuttgart, Germany, pedram.hosseyni@sec.uni-stuttgart.de; Ralf Küsters, University of Stuttgart, Germany, ralf.kuesters@sec.uni-stuttgart.de; Tim Würtele, University of Stuttgart, Germany, tim.wuertele@sec.uni-stuttgart.de.

for monitoring and feedback [9, 15], trigger financial transactions [4, 76], access cars [95] and medical records [17, 93], or perform health-related actions like managing electronic prescriptions [21]. In such high-risk use cases, attacks that enable malicious actors to access resources or impersonate end-users not only have more severe consequences than in classical low-risk settings, but such use cases also require more robust protocols and overall stronger security guarantees. For example, when using SSO to manage access to health records, it is important to not only recognize the same user again at a later point, which is typical for social network SSO, but to provide the user's full legal identity. Likewise, protocols for high-risk use cases should be robust, i.e., provide security even if some messages or relevant values leak to an attacker, e.g., through leaked server log files [77].

To provide security in such high-risk settings, the OpenID Foundation developed FAPI 1.0, which is based on OAuth 2.0 and OpenID Connect, but uses many additional mechanisms to increase their security, for instance, to guarantee authorization and authentication even if the attacker can misconfigure certain endpoints or certain TLS-protected messages leak to the attacker, e.g., via log files. Since FAPI 1.0 meets many ecosystems' needs, it is now in widespread use, e.g., as part of the UK's Open Banking standards [76] and in the Australian Government's *Consumer Data Standards* [13] which govern customer interaction with banks, energy, and telecommunications companies, with more industry sectors to follow. Other examples for the use of FAPI 1.0 include Brazil's Open Finance and Open Insurance programs [4, 74] as well as companies like Verimi [99] where participating banks act as trusted authorization servers that can be used by sites and apps to log in users, identify natural persons, sign legally binding contracts, and interact with authorities, e.g., to register cars. FAPI 1.0 is also employed by the US-based Financial Data Exchange FDX with more than 42 million users [40] and New Zealand's core payment clearing house payments.nz [80]. The high security goals the FAPI 1.0 standard aims to achieve have been formally analyzed [33], uncovering several attacks and proposing fixes for those.

Based on the experiences with FAPI 1.0, including interoperability and implementation aspects, the OpenID Foundation is currently standardizing a successor named FAPI 2.0, which not only comprises a completely new protocol (see Section 4.4) but also comes with new extension protocols to accommodate additional use cases.

The FAPI 2.0 Standards. FAPI 2.0 is a framework of specifications, with the core protocol specified in the *FAPI 2.0 Security Profile (FAPI 2.0 SP)* [29]. Another important specification for our purposes in this framework is the *FAPI 2.0 Attacker Model (FAPI 2.0 AM)* [27], which captures the security goals that the protocol aims to fulfill, along with assumptions on the attacker capabilities.[1] Overall, FAPI 2.0 AM assumes a very strong attacker, far exceeding standard attacker models for protocol analysis (see Section 2.7), owing to the high-risk environments FAPI 2.0 is supposed to be employed in.

To accommodate even more ecosystems' needs, the FAPI 2.0 framework contains additional specifications to extend the core protocol: In some high-stakes environments, operators may want to or may even be legally obliged to prove that some party has sent a certain message, e.g., a payment request. For this purpose, the *FAPI 2.0 Message Signing (FAPI 2.0 MS)* [39] specification draws from several existing mechanisms to provide non-repudiation properties for most of the FAPI 2.0 SP's messages. Furthermore, while the FAPI 2.0 SP protocol requires a user-agent, e.g., a user's browser, to initiate an authorization or authentication flow, and to forward messages between the authorization server and the client, in some scenarios, e.g., for payment authorization at point-of-sale terminals, there is no such user-agent. To cover these scenarios, the *FAPI-CIBA* [98] specification defines a profile of the OpenID Foundation's *Client Initiated Backchannel Authentication (CIBA)* [23] protocol.

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

Despite being a very recent standard, FAPI 2.0 is expected to be adopted soon in many important ecosystems, with several of the aforementioned FAPI 1.0 users already having committed to switching to FAPI 2.0, for example, the previously mentioned Australian Consumer Data Standards [22].

Given the importance of FAPI 2.0 and its current and future use in high-risk environments, the FAPI Working Group (FAPI WG), i.e., the body within the OpenID Foundation developing the FAPI 2.0 standards, has asked us to accompany the standardization process with a formal analysis, providing feedback early on and throughout the development of FAPI 2.0. We hence first performed a detailed formal security analysis of the FAPI 2.0 SP with the security goals specified in the FAPI 2.0 AM, and in a second step, performed a similar analysis of the FAPI 2.0 SP when used in conjunction with FAPI 2.0 MS, FAPI-CIBA, *Dynamic Client Registration (DCR)* [85], and *Dynamic Client Management (DCM)* [86]. While DCR and DCM are not part of the FAPI 2.0 framework, the FAPI WG suggested including them, since FAPI 2.0 protocols will often be used with DCR/DCM, hence justifying a combined analysis. Throughout the rest of the paper, by *FAPI 2.0++*, we denote the combination of the FAPI 2.0 protocols with FAPI-CIBA, DCR, and DCM.

The Web Infrastructure Model (WIM). Our analyses are based on the Web Infrastructure Model (WIM) [35], a symbolic Dolev-Yao-style model [19] of the Web infrastructure. In Dolev-Yao-style models, messages are formal terms with an algebraic equational theory to capture the meaning of cryptographic and other operators. For example, the equation

$$\mathsf{dec}_\mathsf{a}(\mathsf{enc}_\mathsf{a}(m, \mathsf{pub}(k)), k) = m$$

captures that if a message m encrypted under the public key $\mathsf{pub}(k)$ is decrypted under the corresponding private key k, then the resulting message is the message m. Regarding attackers, the WIM models both, Web attackers, who may control certain Web and DNS servers or browsers, and network attackers, who control the complete communication network in addition to corrupted servers and browsers. Both types of attackers can *derive* new messages from their knowledge, e.g., decrypt ciphertexts if they know the key, but they cannot break cryptographic primitives like encryption and signature schemes.

Besides the basic infrastructure, such as Web and DNS servers, the WIM features a detailed model of Web browsers. This browser model captures many Web features, such as the handling of DNS, HTTP, and HTTPS messages, a detailed structure of windows and documents, an abstract model of JavaScript, Web storage and cookies, Web messaging (postMessage) and asynchronous HTTP communication (XMLHTTPRequest/AJAX), a rich set of HTTP headers (e.g., (Set-)Cookie, Location, Origin, Referer, Strict Transport Security, and Authorization), and HTTP redirections as well as security policies for cross-window navigation and access.

As such, the WIM is the most comprehensive and detailed model of the Web infrastructure to date and has been successfully applied to several Web standards, to uncover previously unknown attacks and to prove security properties [18, 33, 34, 35, 36, 38, 52, 53] (see Section 5 for a discussion of related work and Section 4.1 for more details on the WIM). Such a detailed model is necessary to: 1) faithfully model the FAPI 2.0++ protocols since they make use of several Web-specific technologies, and 2) achieve meaningful positive security results, since the details of different Web technologies and how they interact may lead, and in fact have led, to attacks [34, 36, 44].

The WIM is a pen-and-paper framework, thus, analyses based on the WIM are done manually, which, on the one hand, has the benefit that analyses are not limited by the capabilities of mechanized tools, but on the other hand, makes proofs tedious to verify and, in case of protocol changes, require manual re-verification. So far, no mechanized analysis framework has such a comprehensive model of the Web. Mechanizing such a 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 this paper, we use the WIM in its most recent published version [65] and extend it in several places to accommodate the technologies used by the FAPI 2.0++ protocols (see below).

Formal Analysis of FAPI 2.0++: Attacks, Fixes, and Impact. For the analysis of FAPI 2.0++, we build a formal model of the FAPI 2.0 SP, FAPI 2.0 MS, FAPI-CIBA, DCR, and DCM, and formalize the security properties stated in FAPI 2.0 MS as well as the FAPI 2.0 AM and incorporate the assumptions on the attacker laid out therein.

We have coordinated these steps with the FAPI WG to ensure faithful modeling. In the process of formally proving the security properties within our FAPI model, we have revealed several attacks that break the goals of the protocol. We propose fixes and improvements to the specifications, which the FAPI WG has appreciated and amalgamated into the official specifications, resulting in substantial changes of the standard.

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

Our analysis of the FAPI 2.0 SP was first published in an extended abstract [53]. Here, in this journal version, besides providing more details of the analysis in [53], we greatly extend our analyses by four more protocols (FAPI 2.0 MS, FAPI-CIBA, DCR, and DCM). Our analysis captures the fact that all of these protocols, i.e., FAPI 2.0 SP, FAPI 2.0 MS, FAPI-CIBA, DCR, and DCM, can run in parallel. So, not only is the resulting model considerably larger and the number of security properties more than doubled but all previously formulated security properties—and hence their proofs—changed as well due to the parallel composition of all these protocols.

Contributions. In summary, our contributions are as follows:

- We extend the WIM's browser and communication model with a push-message channel, a model of HTTP Message Signatures, and several HTTP headers, like DPoP, Content-Digest, and Signature. All of these extensions are of interest also independently of our work on FAPI 2.0 and can be used in future analyses of other protocols based on the WIM.
- We provide the first formal model of a FAPI 2.0 ecosystem, covering not only the core Security Profile, but also the FAPI 2.0 MS and FAPI-CIBA specifications, along with a formalization of the security goals set forth in the FAPI 2.0 AM and FAPI 2.0 MS specifications.
- Furthermore, our model of a FAPI 2.0 ecosystem features Dynamic Client Registration and the first formal model of Dynamic Client Management, both of which are also used outside the FAPI 2.0 context, and hence, are of independent interest.
- Our analysis has uncovered several attacks, i.e., violations of the security goals under the attacker model defined by the FAPI WG.
- We propose fixes and improvements and worked with the FAPI WG to incorporate them, resulting in significantly modified and improved FAPI 2.0++ specifications.
- We adapted our formal model to reflect the improved specifications and were then able to prove the formalized security goals.
- We have accompanied the development of the FAPI 2.0++ specifications from an early stage and were able to support the standardization process with security recommendations before the widespread deployment of the FAPI 2.0++ protocols in high-risk environments with tens of millions of users.

Structure of This Paper. We first give a detailed description of the FAPI 2.0++ protocols, security goals, and attacker model in Section 2 and then present the attacks that we have discovered in the process of our formal analysis in Section 3. We describe our formal model, including extensions to the WIM, and formal security theorem in Section 4. The full formal model and proofs are given in the appendix. Related work is discussed in Section 5. We conclude in Section 6.

2 PROTOCOLS AND SECURITY GOALS

Here, we describe the protocols and security goals as of the start of our analysis efforts, i.e., as of June 1st, 2022 for FAPI 2.0 SP and FAPI 2.0 AM, and as of May 4th, 2023 for FAPI 2.0 MS and FAPI-CIBA. The specifications for DCR and DCM are not part of the FAPI 2.0 framework and have been finalized prior to our work, we therefore model these final versions. For all specifications, we discuss changes made since the onset of our work in Section 3. We begin with an overview of the FAPI 2.0 SP (Section 2.1), introducing the protocol roles and core protocol flow, followed by a detailed description thereof in Section 2.2. In Section 2.3, we describe the FAPI 2.0 MS extension, followed by a description of the FAPI-CIBA protocol in Section 2.4, and the DCR and DCM protocols in Section 2.5, before detailing the security goals and assumptions on the attacker in Sections 2.6 and 2.7, as outlined in the FAPI 2.0 AM and FAPI 2.0 MS specifications.

2.1 Overview of the FAPI 2.0 Security Profile

In a nutshell, FAPI 2.0 SP 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 for managing 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 SP may be used to grant an account aggregation service (client) access to a user's account balance at various banks (RSs), with services of these banks (ASs) managing such access (such services exist today, e.g., [4, 9, 15, 76]).

On a high level, a FAPI 2.0 SP 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 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.

2.2 The FAPI 2.0 Security Profile in Detail

In the following, we describe a FAPI 2.0 SP protocol flow in detail (see 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).

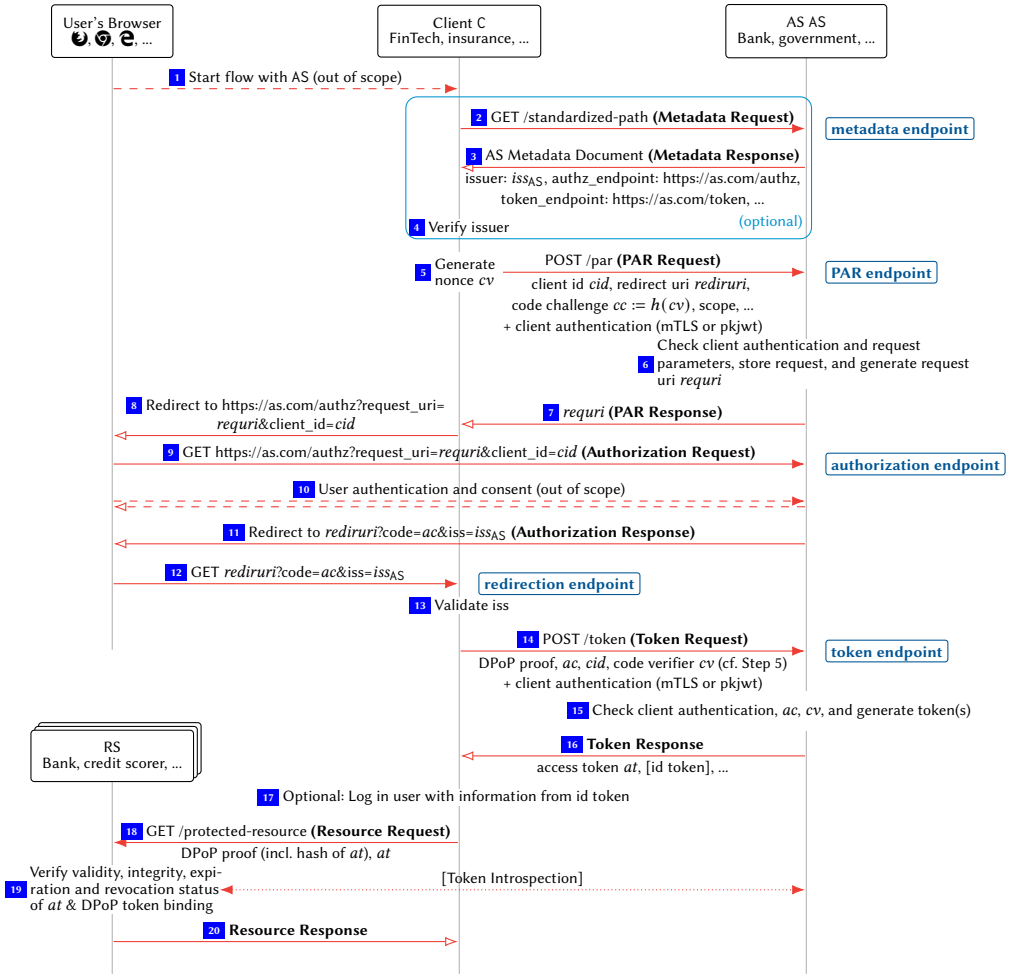

Fig. 1. FAPI 2.0 Security Profile protocol flow (with DPoP sender constraining)

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 [103] and consists of an HTTPS URL without query or fragment components. However, to complete a FAPI 2.0 flow, the client needs additional knowledge on AS, e.g., endpoint URLs (which in general are different to the issuer identifier). If C does not yet know all necessary values (e.g., via configuration), it can proceed by fetching so-called *Authorization Server Metadata* [61, 91] from AS by sending a corresponding request to the URL iss_{AS}, with a standardized path appended to it (Step 2). However, this step is optional. Like all other communication in FAPI 2.0++, this exchange is done via HTTPS, i.e., is protected by TLS. The metadata returned by AS includes URIs of the relevant endpoints, supported cryptographic algorithms, and similar information, along with the issuer identifier of AS (Step 3). Once the client acquired the metadata response, it verifies that the issuer value contained in the response equals the value that it used to create the metadata request.

Once the required values are available, C assembles a *Pushed Authorization Request* (PAR) [71] 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) [89]. 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 the so-called *request uri requri*, which is essentially a nonce (the format of the uri is not specified, however, the value must contain a non-guessable part). AS then stores the requested scope, *cid*, $cc := h(cv)$, *rediruri*, and *requri* (Step [6]); *requri* will be used as a reference to the PAR data in Step [9] and is therefore sent to C in the PAR response (Step [7]). Client C then redirects the user's browser to AS, adding *requri* and *cid* as request parameters (Step [8]). Following that redirect, the user's browser visits AS and in doing so, forwards *requri* and *cid*, hence providing information on the user's context (i.e., the current flow) to AS (Step [9]). The user now authenticates at AS and reviews the access requested by C (Step [10]), the exact details of this step are up to the AS and out of scope of FAPI 2.0. If the user consents, AS generates a random *authorization code ac* and stores it with the PAR data from Step [5]. AS then redirects the user's browser back to the *rediruri* of C (stored in Step [6]), and includes *ac* as well as an *iss* value [103] (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 in that request by comparing it to the issuer identifier of the AS to which the client sent the PAR request in Step [5], i.e., iss_{AS}, to prevent mix-up attacks [34, 70, 75, 103] which exploit the interaction between ASs and clients via user's browsers. If the issuer 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 (explained below), 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 the 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 [79]: token introspection [83], and structured access tokens, which contain the necessary information and are typically signed by the AS [8, 57]. For 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 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 [10] 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` [90], the client adds a signed *JSON Web Token (JWT)* [57, 59, 60] to its messages. This JWT contains, among other things, the client's id at the AS, the issuer identifier of the AS, and a fresh 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). This mechanism, called *access token sender constraining*, is chosen independently of the client authentication mechanism (see also the explanation below). 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)* [32], which is shown in Figure 1, and mTLS [10]. In both cases, the access token is bound to a client key pair, e.g., by including a hash of the DPoP public key or mTLS certificate in the token, and the client has to include a proof of possession of the corresponding 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 (without parameters and fragment components), a nonce chosen by the client, and a public verification key $\mathsf{pub}(k)$ (of the client's choice). *dpopJWT* is signed using the corresponding private key k. The AS then binds the access token to $\mathsf{pub}(k)$. 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 again that client authentication and access token sender constraining mechanisms are chosen independently of each other, and that sender constraining is not meant to identify the client towards the RS. E.g., a client that 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.

2.3 FAPI 2.0 Message Signing

With FAPI 2.0 SP alone, one cannot expect to achieve accountability properties. However, in the context of high-stakes applications, parties may be interested in or even legally required to be able to prove that some other party sent a certain message, e.g., a payment order. To accommodate such applications, the FAPI 2.0 framework features a standard called *FAPI 2.0 MS* [39], which aims to add non-repudiation properties to FAPI 2.0 ecosystems.

Similar to FAPI 2.0 SP, FAPI 2.0 MS combines existing standards and adds some requirements to enhance security and interoperability. Furthermore, FAPI 2.0 MS is somewhat modular in that a given deployment does not necessarily have to use all of the below profiles, but can select which of them to use, depending on the specific deployment's requirements. All of the profiles have in common that they add sender signatures to one or more of the messages sent in a flow of FAPI 2.0 SP. The available profiles are:

Signed Authorization Requests For signed authorization requests, the *OAuth 2.0 JWT Secured Authorization Request (JAR)* [92] is employed. In the context of FAPI 2.0 SP, the relevant message to be signed (by the client) is the pushed authorization request (Step 5 in Figure 1).

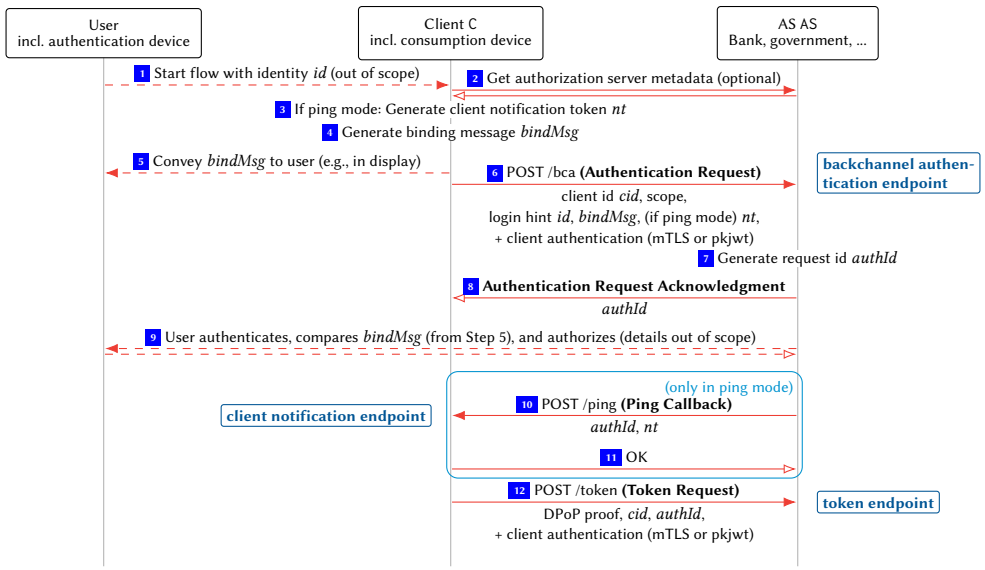

Fig. 2. FAPI-CIBA protocol flow (with DPoP sender constraining) – the remaining steps after the token request are the same as in Figure 1.

Signed Authorization Responses Signed authorization responses are implemented using the *JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)* [68] (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* [72] specification is applied.

Signed HTTP Messages To sign resource requests and responses (Steps 18 and 20 in Figure 1), *HTTP Message Signatures* [3] are used. This profile allows signing both, request and response, or only one of them.

2.4 FAPI-CIBA

The FAPI-CIBA profile [98] of the CIBA [23] authentication flow covers use cases of FAPI 2.0 in which the user aims to authorize a *consumption device (CD)* of a client but uses a different device, the *authentication device*, often a smartphone, to authenticate and provide consent, for example when authorizing a payment at a point-of-sale terminal.

Figure 2 depicts a FAPI-CIBA authentication flow, where we assume that the CD is subsumed by the client as the communication between the CD and client is not specified by CIBA or FAPI-CIBA (the CD is "the device that helps the user consume the service" [23], thus, can essentially be seen as the interface of the client towards the user). The flow is initiated by the user providing an identity *id* to the client C (Step 1), e.g., by entering their phone number at a point-of-sale terminal. Note that the authorization server AS used by C in the following is either fixed for a given CD or derived from *id*. As in FAPI 2.0 SP, C may now fetch AS metadata (Step 2). The next step depends on the *delivery mode* chosen by C: in the *poll* mode, the client polls the AS for the requested tokens, whereas in the *ping* mode, the client waits for the AS to notify it before sending a token request. In the case of the ping mode, C generates a *client notification token nt* in Step 3 that the AS will use later when notifying C of completed authorization by the user (see Step 10). Next, C generates a *binding message bindMsg* (Step 4), a short human-readable string used to link the user's interaction

with the CD to the user's interaction with the authentication device. This binding message is then conveyed to the user (Step 5), e.g., by showing it on a display of the CD.

Meanwhile, C sends an *authentication request* to AS (Step 6) with its client id, the requested scope, and client authentication data as in the PAR request in Section 2.2. In addition to these values, the authentication request contains 1) the *id* from Step 1 as a so-called *login hint* which AS later needs to determine which user to contact, 2) the binding message *bindMsg*, and, if the ping mode is used, 3) the client notification token *nt*. After validation of the authentication request, AS generates a (random) reference *authId* (Step 7) and includes it in its response to C (Step 8). In poll mode, C now starts to send repeated token requests to AS.

Subsequently, AS contacts the user identified by *id*, e.g., via a push notification to their phone (Step 9), asking them to not only authenticate to AS with *id* and authorize C's request but to also compare the binding message sent by AS to the one the user received from the CD in Step 5. As before, the details of this interaction between user and AS are out of scope.

In ping mode, AS now sends a *ping callback* to C's notification endpoint with *authId* and *nt* to inform C about the completed authorization (Step 10), which C acknowledges in Step 11. The notification endpoint is part of the data a client needs to register with an AS (if that client wants to use the ping mode).

Finally, C sends a token request with information for access token sender constraining (see Section 2.2), its client id, the reference *authId*, and client authentication information to the AS (Step 12). The remaining steps, i.e., token response and retrieving resources, are the same as for the FAPI 2.0 SP flow described in Section 2.2. In particular, the access token is sender-constrained, either using DPoP or mTLS.

User Identity Hint. In Step 6, the client has to include a hint on the user's identity. CIBA defines three options for this hint, an authentication request must contain exactly one of them: 1) a *login hint* as described above contains a global user identifier, e.g., a user's email address or phone number, 2) a *login hint token* is an application-specific data structure with information to identify the user, and 3) an *id token hint* contains a previously issued (possibly expired) id token.

2.5 Dynamic Client Registration and Management

As mentioned before, a FAPI 2.0 client needs to be registered with a FAPI 2.0 AS which includes registering client keys, e.g., for client authentication and access token sender constraining, a client id issued by the AS, but also client metadata like the types of access token sender constraining supported by the client and the client notification endpoint (see Section 2.4). While this registration can be done out-of-band, e.g., via manual configuration, this is not feasible in many real-world use cases, such as single-page applications. Hence, the OAuth 2.0 Dynamic Client Registration Protocol [85] 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 [86] allows clients to change their configuration at an AS.

For DCR, the AS offers an additional HTTPS endpoint to which clients send their desired configuration, to which the AS replies with the values it registered (which may differ from the ones the client wished to register, e.g., the AS might restrict the possible FAPI-CIBA delivery modes). 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. How a client obtains such an initial access token is, 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.

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

2.6 Security Goals

Along with the actual protocol specification, the FAPI WG developed the FAPI 2.0 Attacker Model [27] which outlines security goals and assumptions on attackers under which these goals are expected to hold for FAPI 2.0 SP. As before, we describe the state as of June 1st, 2022 here and discuss changes made since then in Section 3. Furthermore, the FAPI 2.0 MS specification adds security goals related to non-repudiation which have not changed since the onset of our work. Since the FAPI-CIBA, DCR, and DCM specifications do not state any explicit security goals, we consulted with the FAPI WG and agreed to expect the same level of security as for FAPI 2.0 SP, i.e., the same security goals apply. The formalized security properties and modeling of attacker assumptions are presented in Section 4.

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

Authentication. 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. 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. 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.

Non-repudiation Properties. For signed authorization requests, non-repudiation means that if an honest AS accepts a pushed authorization request it expects to be signed, then that request is signed. Furthermore, if the signature is valid for a key registered with the AS by an honest client, then that client cannot deny having signed the request. Note that since we model an ecosystem in which flows with and without signatures can be used in parallel, the receiver has to decide whether it expects a given message to be signed. In practice, this decision may for example depend on the message content or the (claimed) sender.

Non-repudiation for signed authorization responses, signed introspection responses, signed resource requests, and signed resource responses is defined accordingly.

Note that the distribution of verification keys is out of scope of the FAPI 2.0 specifications, but ASs will typically learn clients' verification keys as part of client registration, whereas ASs and RSs typically make use of the existing TLS PKI by providing a TLS-protected so-called *JWKs endpoint* where they publish their verification keys.

2.7 Attacker Model

In the following, we summarize the attacker assumptions laid out in FAPI 2.0 AM. We stress that these (strong) assumptions are part of and justified by the specification, e.g., to account for leaked server logs or leaked browser history.

$\mathcal{A}$**1**. The attacker controls the network, i.e., can intercept, block, and tamper with all messages sent over the network. In particular, the attacker can also reroute, reorder, and create (from its knowledge) new messages. However, the attacker cannot break cryptography unless it learns the respective keys. From a network point of view, the attacker can pose as any party (and any network participant) in the protocol. In addition, the attacker can also send links to (honest) users which are then visited by these users. See [28, Sec. A1, A1a, A2].
$\mathcal{A}$**2**. The attacker can read authorization requests in plain (Step 9 in Figure 1). See [28, Sec. A3a].
$\mathcal{A}$**3**. The attacker can read authorization responses in plain (Step 12). See [28, Sec. A3b].
$\mathcal{A}$**4**. The attacker can trick the client into using an attacker-controlled token endpoint URL (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 that do not use the AS metadata mechanism. See [28, Sec. A5].
$\mathcal{A}$**5**. Resource requests (Step 18) leak to the attacker in plain. See [28, Sec. A7].
$\mathcal{A}$**6**. Resource responses (Step 20) leak to the attacker in plain. See [28, Sec. A7].
$\mathcal{A}$**7**. The attacker can modify resource responses (Step 20), i.e., can replace an honest RS' resource response with its own message without the client noticing, even though that response is protected by TLS. Note that this does *not* give the attacker the ability to replace arbitrary messages in TLS connections, but is limited to resource responses (see [28, Sec. A8]). The specification mentions a compromised reverse proxy in front of the RS as a possible reason for this attacker ability.

Note that attacker assumption $\mathcal{A}1$ corresponds to the capabilities of a standard network attacker (with the additional ability to prompt users to open arbitrary links) and does not imply the other assumptions, as FAPI 2.0++ requires the use of TLS for all messages. E.g., $\mathcal{A}2$ enables the attacker to learn the request uri value of an authorization request even if the request is sent between honest parties (which the attacker cannot learn by $\mathcal{A}1$ due to TLS).

3 ATTACKS

We formally modeled the FAPI 2.0++ specifications mentioned above and then formalized and tried to prove the security goals laid out therein under the attacker assumptions outlined in FAPI 2.0 AM. In the course of this analysis, we have uncovered several attacks, i.e., violations of the security goals. We have discussed these findings with the FAPI WG and worked with them to resolve the issues, resulting in a number of changes to the specifications which we explain here. The formal model presented in Section 4, for which we prove security, incorporates these changes. While, to the best of our knowledge, Attacker Token Injection, the Client Impersonation attacks, and DPoP Proof Replay are completely new attacks, interestingly, for the other attacks (Browser Swapping, Cuckoo's Token, Cross-Device Consent Phishing, Authorization Request Leak), similar attack patterns have been reported for related protocols [14, 33, 55, 69]. This emphasizes the importance of systematic, formal analysis, as even seasoned experts overlook known attack patterns for complex protocols.

3.1 Attacker Token Injection

This two-phased attack violates both session integrity goals and requires attacker assumptions $\mathcal{A}4$ (token endpoint misconfiguration) and $\mathcal{A}5$ (resource requests leak, see Section 2.7). In the first phase of this attack, the attacker, posing as a user, completes a flow with an honest client $\mathsf{C}_{\mathsf{hon}}$ and honest AS $\mathsf{AS}_{\mathsf{hon}}$. During this flow, the attacker uses $\mathcal{A}5$ to obtain at_{att}, i.e., an access token bound to keys of $\mathsf{C}_{\mathsf{hon}}$, issued by $\mathsf{AS}_{\mathsf{hon}}$ for resources of the attacker. For the second phase, the attacker uses $\mathcal{A}4$, such that $\mathsf{C}_{\mathsf{hon}}$ uses an attacker-controlled token endpoint (instead of $\mathsf{AS}_{\mathsf{hon}}$'s token endpoint). Hence, when an honest user u starts a flow with $\mathsf{C}_{\mathsf{hon}}$ and $\mathsf{AS}_{\mathsf{hon}}$, the attacker receives $\mathsf{C}_{\mathsf{hon}}$'s token request, to which the attacker answers with at_{att} and an id token constructed by the attacker for an

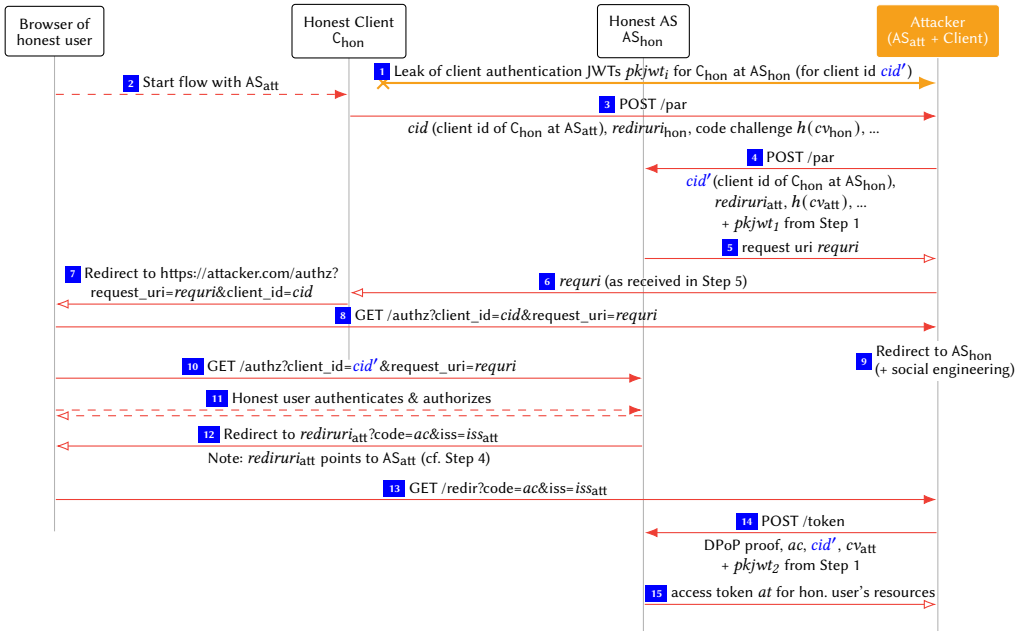

Fig. 3. Client impersonation attack

attacker identity. Client C_{hon} then logs u in under the attacker's identity and uses attacker resources in a session with u.

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

3.2 Client Impersonation Attacks

This attack violates the authorization goal, requires attacker assumption $\mathcal{A}4$ (token endpoint manipulation, see Section 2.7), and targets an honest client C_{hon} that uses the `private_key_jwt` method (see Section 2.2) to authenticate itself to an honest AS AS_{hon}. Using $\mathcal{A}4$, the attacker modifies C_{hon}'s configuration such that C_{hon} uses an attacker-controlled token endpoint (instead of AS_{hon}'s).

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

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

With such valid client authentication JWTs for C_{hon} at AS_{hon}, the attacker can proceed as depicted in Figure 3: An honest user u starts a flow with C_{hon} and expresses its wish to use an AS AS_{att}, identified by issuer identifier iss_{att}, which happens to be controlled by the attacker (Step 2). Hence, C_{hon} sends a PAR request to AS_{att}, containing its client id cid at AS_{att}, a redirect uri, a code challenge

$h(cv_{\mathsf{hon}})$ for a code verifier cv_{hon} chosen by $\mathsf{C}_{\mathsf{hon}}$, etc. as described in Section 2.2 (Step 3). Instead of replying to this request immediately, the attacker now poses as $\mathsf{C}_{\mathsf{hon}}$ towards $\mathsf{AS}_{\mathsf{hon}}$: the attacker sends its own PAR request to $\mathsf{AS}_{\mathsf{hon}}$, assembled from an attacker-chosen code challenge $h(cv_{\mathsf{att}})$, a redirect uri $redir uri_{\mathsf{att}}$ pointing to a URL of $\mathsf{AS}_{\mathsf{att}}$, the client id cid' of $\mathsf{C}_{\mathsf{hon}}$ at $\mathsf{AS}_{\mathsf{hon}}$, and $pkjwt_1$ (Step 4). Upon receiving the attacker's PAR request, $\mathsf{AS}_{\mathsf{hon}}$ validates the request and replies with *requri* (Step 5), which the attacker forwards to $\mathsf{C}_{\mathsf{hon}}$ (now again in the role of $\mathsf{AS}_{\mathsf{att}}$ towards $\mathsf{C}_{\mathsf{hon}}$) in Step 6 in response to $\mathsf{C}_{\mathsf{hon}}$'s original PAR request from Step 3.

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

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

We stress that u authenticates at an honest, trusted AS and authorizes not only an honest and trusted client, but also exactly the client u expected to authorize. There are some variants of this attack with slightly different preconditions, but similar outcomes, which we describe in Appendix A.1. These attacks emerged when we tried to prove that a client authentication JWT for authentication of an honest client at an honest AS $\mathsf{AS}_{\mathsf{hon}}$ (i.e., the JWT contains the issuer identifier of $\mathsf{AS}_{\mathsf{hon}}$) cannot leak to an attacker (Lemma 12).

Fix. Since the possible misconfiguration of an honest client's token endpoint is the root cause for both the attacker token injection and the client impersonation attacks and FAPI 2.0 ASs are already required to serve a metadata document, our proposed fix of mandating clients to request and use this metadata was adopted by the FAPI WG [30, 48, 50]. Recall $\mathcal{A}4$: such token endpoint misconfiguration is only considered for clients which do not use the AS metadata mechanism.

3.3 DPoP Proof Replay

With attacker assumption $\mathcal{A}5$ (resource requests leak) from Section 2.7, the attacker can read resource requests in plain and hence can try to replay them at the RS (cf. Step 18 in Figure 1), thus violating the authorization goal (see Section 2.7) if the RS accepts the replayed request.

When DPoP sender constraining is used, the attacker can indeed replay the client's DPoP proof (using the attacker's TLS keys for the underlying connection): neither FAPI 2.0++, nor DPoP [32] itself, nor the specifications on which DPoP is built [56, 57, 90] mandate for DPoP proofs to be strictly one-time use. Hence, the specifications do not mandate the RS to reject a replayed proof.

Our initial attempts to prove the authorization property (Definition 3) revealed a second variant, which also violates the authorization goal: with $\mathcal{A}1$ (network attacker), the attacker can additionally block the honest client's request, i.e., from the point of view of the RS, the attacker's resource request is not even a replay, and hence, replay protection cannot prevent this attack.

Note that neither variant of this attack is possible with mTLS sender constraining since the attacker cannot even establish an mTLS connection to the RS for the client's mTLS key (to which the access token is bound when mTLS sender constraining is used).

Fix. In our discussions with the FAPI WG, it became clear that the FAPI WG formulated attacker assumption $\mathcal{A}5$ with leaks of RS server log files in mind. Hence, FAPI 2.0 AM was changed to clarify that resource requests leak *after* processing by the RS [26]. However, this only resolves the problem resulting from the attacker blocking the honest client's request, but does not prevent the replay attack. Hence, we proposed to fix this attack by mandating the use of *resource server-provided nonces* with strict one-time use enforcement by the RS [32, Sec. 9]. The server-provided nonce mechanism is an optional part of DPoP in which—in a challenge-response manner—the client first requests a nonce from the RS which the client then has to include in its DPoP proof. We validated the effectiveness of this fix in our formal model.

The FAPI WG acknowledged the attack [49], and added a description of the attack, as well as several options to fix it, to the specification.

3.4 Browser Swapping Attack

In this attack, the attacker violates the authorization and authentication goals by combining attacker assumptions $\mathcal{A}1$ (network attacker) and $\mathcal{A}3$ (authorization responses leak, see Section 2.7).

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

Due to space constraints, we refer to Appendix A.2 for a detailed description.

At the heart of this attack is the lack of a strong connection between the sessions user–client and user–AS. We discovered it when trying to prove the authorization property (see Definition 3): while proving that in a flow between an honest client, honest AS, and honest user, the client does not leak the user's resources, we have to prove, among others, that the authorization code associated with the flow cannot be sent to the client's redirection endpoint by the attacker.

Fix. The FAPI WG acknowledged this attack and after several discussions, there was consensus that this attack cannot be fixed with currently deployed methods [31, 87]. This decision is documented and explained along with the attack in the specifications [27, Sec. 6.5.7]. Consistent with this decision, the FAPI WG also removed attacker assumption $\mathcal{A}3$, i.e., FAPI 2.0 no longer claims to fulfill its security goals when authorization responses leak.

3.5 Cuckoo's Token Attack

In this attack, the attacker leverages attacker assumption $\mathcal{A}5$ (resource requests leak) to violate the authorization goal (see Section 2.7).

Said attacker assumption allows the attacker to obtain an access token at_{hon} from an honest flow, i.e., at_{hon} was issued by an honest AS $\mathsf{AS}_{\mathsf{hon}}$ for an honest client $\mathsf{C}_{\mathsf{hon}}$ on behalf of an honest user u (and thus at_{hon} is bound to keys of $\mathsf{C}_{\mathsf{hon}}$). By then injecting at_{hon} into a flow between the attacker as user, $\mathsf{C}_{\mathsf{hon}}$, and an attacker-controlled AS, the attacker (as user of $\mathsf{C}_{\mathsf{hon}}$) gains access to u's resources.

Figure 4 shows the attack in detail: the attacker first acquires an access token at_{hon} from an honest flow (Step 1). Such a token may, for example, be obtained by the attacker through observing a resource request (see $\mathcal{A}5$). Due to access token sender constraining, the attacker cannot use at_{hon} directly at an RS (recall: at_{hon} is bound to keys of $\mathsf{C}_{\mathsf{hon}}$). Instead, the attacker, posing as a user towards $\mathsf{C}_{\mathsf{hon}}$, now starts a flow with $\mathsf{C}_{\mathsf{hon}}$, selecting an AS $\mathsf{AS}_{\mathsf{att}}$ controlled by the attacker (Step 2). $\mathsf{C}_{\mathsf{hon}}$ initiates the flow as usual with PAR (Step 3), followed by instructing the user, i.e., attacker, to visit $\mathsf{AS}_{\mathsf{att}}$ (Step 4). Since $\mathsf{AS}_{\mathsf{att}}$ is controlled by the attacker, the authentication and authorization steps can be skipped and the attacker (posing as a user) immediately "redirects" itself to $\mathsf{C}_{\mathsf{hon}}$ with

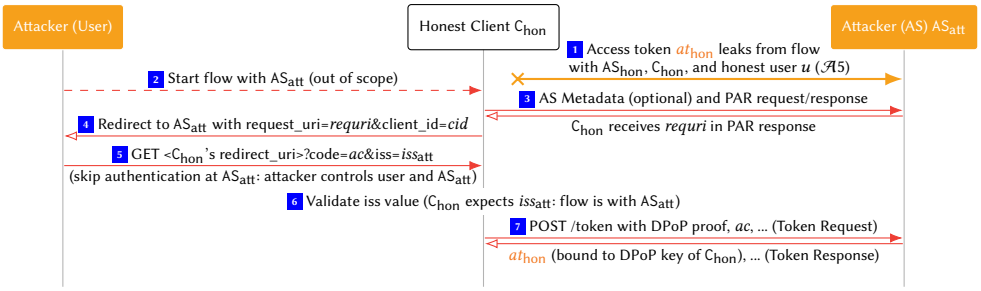

Fig. 4. Cuckoo's token attack

issuer identifier iss_{att} (which identifies AS_{att}) and an arbitrary authorization code *ac* (Step 5). Upon receiving that message, C_{hon} validates the issuer identifier (Step 6), which succeeds: C_{hon} is and wants to be in a flow with AS_{att}. As usual, C_{hon} now sends a token request to AS_{att}, which responds with the previously acquired at_{hon} (Step 7). Recall that at_{hon} was issued for and is bound to keys of C_{hon} to access resources of u. Hence, C_{hon} can use at_{hon} at an honest RS. But since C_{hon} associates at_{hon} with the session between C_{hon} and the attacker, this gives the attacker access to u's resources.

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

Fix. We proposed to fix this attack by mandating the client to include an AS issuer identifier in each resource request (which would be the attacker AS' identifier in the example above). The RS can then compare this issuer identifier sent by the client with the actual issuer of the access token (which, in our example, would be some different, honest AS). Note that in order to verify the token's validity, the RS already needs to determine which AS originally issued the token. The FAPI WG acknowledged the attack [88] and added a description of the attack, as well as our proposed fix, to the specification [51]. However, the FAPI WG decided that the "preconditions for this attack do not apply to many ecosystems and require a powerful attacker" [29, Sec. 5.6.5] and hence made implementation of a fix optional. Note that while this fix seems to be a small change, there is no standardized way to send the issuer identifier of the AS, as well as no standardized way for the RS to get a value to compare against [29, Sec. 5.6.5]. Hence, mandating such a change would require substantial standardization efforts.

3.6 Cross-Device Consent Phishing Attack on FAPI-CIBA

FAPI-CIBA is a cross-device authentication and authorization protocol. A known weakness of such protocols is the lack of authentication of the CD towards the user, which is exploited by so-called *cross-device consent phishing attacks* [14, 55, 62] to violate the authorization and authentication goals. Specifically, in Step 9 of Figure 2, besides the binding message, the user only has the information shown by the AS to determine which client they are currently authorizing. With FAPI-CIBA, the binding message is supposed to mitigate this problem by "enabling the end-user to ensure that the action taken on the authentication device is related to the request initiated by the consumption device." [23, Sec. 7.1]. However, as the attack in Figure 5 shows, this is not the case:

An honest user u starts a flow at a malicious CD, using one of u's identities *id* at an honest AS AS_{hon} (Step 1). The attacker forwards this to an honest client C_{hon} (Step 2), i.e., from the point of view of C_{hon}, the attacker is a regular user. Consequently, C_{hon} generates a binding message *bindMsg* (Step 3) and conveys *bindMsg* to C_{hon}'s user, i.e., the attacker (Step 4). While the attacker (in its role

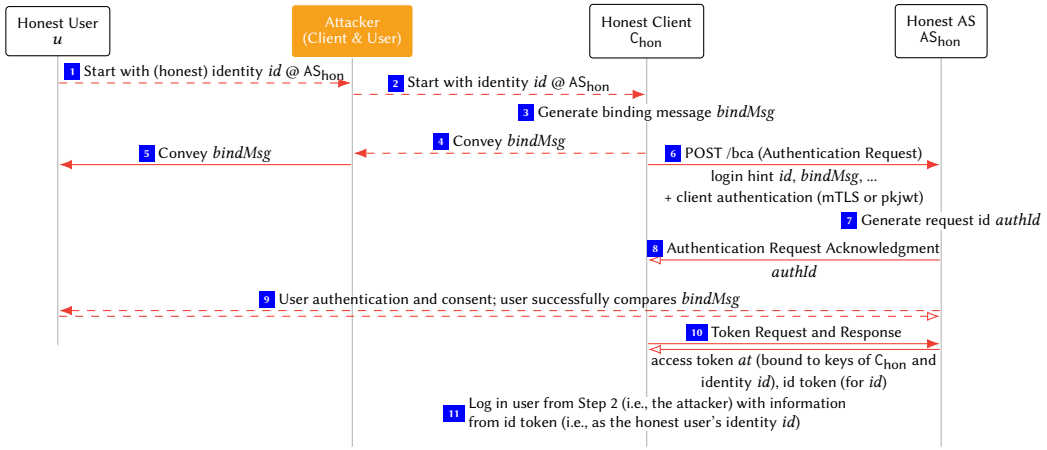

Fig. 5. Cross-device consent phishing attack on FAPI-CIBA (malicious client variant)

as a CD) conveys *bindMsg* to the honest user (Step 5), C_{hon} sends an authentication request with login hint *id* to AS_{hon} (Step 6), which generates a request id (Step 7) and sends it to C_{hon} (Step 8). Next, AS_{hon} contacts the user associated with *id*, i.e., *u*, and asks them to authenticate, compare the binding message *bindMsg*, and to authorize C_{hon}'s request (Step 9). Note that the binding message presented by AS_{hon} in Step 9 indeed matches the one presented by the malicious CD (see Step 5). Furthermore, note that *u* is asked to authorize a request by C_{hon}, whereas *u* started the flow with and obtained *bindMsg* from the malicious CD – this is where the lack of authentication of the (in this case malicious) CD towards the user comes into play: the attacker CD can claim to be C_{hon} towards *u* and *u* has no way to detect this deception. In practice, this is worsened by the fact that the client information shown by the AS to the user in Step 9 often only consists of a manufacturer name and logo.

Following *u*'s authorization, C_{hon} obtains access and id tokens linked to *id* (Step 10), and logs its user, i.e., the attacker, in under identity *id* (Step 11), violating the authentication goal. Similarly, C_{hon} uses the access token to access resources of *u*, which C_{hon} associates with its session with the attacker, thus violating the authorization goal. We describe another variant of this attack that does not require the user to initiate a flow with the attacker in Appendix A.3.

Fix. As explained above, this attack relies on the client device not being authenticated towards the user in such a way that the user can reliably compare the client device he/she is using against the client they are authorizing. A possible fix is therefore to require the consumption device to authenticate itself towards the user. However, such (cryptographic) authentication is problematic, not only due to usability constraints but also due to the limited input/output capabilities of client devices as such devices are the motivation for the CIBA flow in the first place. We reported our findings, in particular about the binding message not preventing this attack, to the FAPI WG and discussed possible mitigations. There now is an ongoing effort by the OAuth Working Group to give guidance on how to mitigate such attacks [62], e.g., by establishing proximity between the client device and the user/authentication device using technologies like NFC or Bluetooth Low Energy. As of this writing, the FAPI WG is working on updating the specifications with our findings on the binding message and references to [62] for mitigations [97].

3.7 Authorization Request Leak Attack

A combination of $\mathcal{A}1$ (network attacker) and $\mathcal{A}2$ (authorization request leaks, see Section 2.7) allows the attacker to violate both session integrity goals: the attacker starts by intercepting ($\mathcal{A}2$) and blocking ($\mathcal{A}1$) the authorization request (Step 8 in Figure 1) of an otherwise honest flow between an honest user u, an honest client $\mathtt{C}_{\mathsf{hon}}$, and honest AS $\mathtt{AS}_{\mathsf{hon}}$. Next, the attacker continues that flow by visiting $\mathtt{AS}_{\mathsf{hon}}$'s authorization endpoint with the leaked authorization request, and logs in using its own (attacker) identity. $\mathtt{AS}_{\mathsf{hon}}$ then answers with an authorization response, containing an authorization code ac, which $\mathtt{AS}_{\mathsf{hon}}$ associates with the attacker's identity and the PAR of the initial, honest flow. From the authorization response, the attacker assembles a link pointing to $\mathtt{C}_{\mathsf{hon}}$'s redirection endpoint with ac and the `iss` parameter as received from $\mathtt{AS}_{\mathsf{hon}}$, and sends this link to u. Once u follows this link, $\mathtt{C}_{\mathsf{hon}}$ exchanges ac for tokens, logs u in under the attacker's identity, and accesses resources belonging to the attacker in a session with u, hence violating both session integrity goals.

Fix. Upon our notification, the FAPI WG acknowledged the attack [25]. After some discussion, there was consensus that there is no currently deployed technology that can prevent such attacks. However, the FAPI WG wanted to keep $\mathcal{A}2$ in their attacker model. Therefore, the FAPI WG decided to document the attack, as well as (optional) mitigations, which do not prevent, but harden against the attack in practice, in the specifications [29, Sec. 5.6.6]. For our analysis, we formulated the session integrity properties such that they only apply to flows in which the authorization request does not leak (see Appendix E)

3.8 Inconsistencies in Attacker Model

In addition to the attacks described above, we discovered inconsistencies in FAPI 2.0 AM, i.e., assumptions on the attacker that immediately violate one or more of the security goals (see Section 2.7).

$\mathcal{A}6$ **Violates Authorization Goal.** As resource responses contain users' resources, this attacker assumption (leak of resource responses) immediately violates the authorization goal. When we pointed this out to the FAPI WG, it was decided to drop this attacker assumption [26].

$\mathcal{A}7$ **Violates Session Integrity for Authorization Goal.** If the attacker can tamper with resource responses, honest users can be forced to use attacker resources by replacing (honest) resources in resource responses with attacker resources. This violates the session integrity for authorization goal. As above, the FAPI WG decided to drop attacker assumption $\mathcal{A}7$ once we pointed this inconsistency out [26].

3.9 User Identifier Mix-up in FAPI-CIBA

Another problematic scenario for FAPI-CIBA violates both session integrity goals if the user in its first message to an honest client device (Step 1 in Figure 2) mistypes their identity, e.g., instead of `alice@as.com`, they enter `malice@as.com` (note that both identities may belong to an honest AS, and they do not necessarily need to belong to the same AS). The client then sends the usual authentication request to the AS of the mistyped identity, and that AS contacts the associated user, i.e., the attacker. If the attacker now authorizes the user's device, the flow continues as usual, with the user's device logging the user in under the attacker's identity and using attacker resources, hence violating both session integrity goals.

When discussing this issue with the FAPI WG, it became apparent that FAPI-CIBA is not intended to prevent or mitigate this flow due to its reliance on the user to enter the wrong identity.

Nonetheless, the security considerations section of the FAPI-CIBA specification provides guidance on how to get more reliable login hints [98, Sec. 5.2].

4 FORMAL ANALYSIS

In this section, we describe our formal analysis. We start with a primer on the WIM and how we extend it, followed by an overview of our formal model of FAPI 2.0++, including its limitations. Subsequently, we discuss the most important differences and technical challenges of our work compared to prior work on authorization protocols using the WIM, before describing our formal model of FAPI 2.0++ and the formalized security properties, including our security theorem. We refer to the appendix for the full formal model and full proofs of all security properties.

4.1 The Web Infrastructure Model

In the following, we give a high-level overview of the WIM closely following the summary in [34], with the full model given in Appendix G: the WIM is designed independently of a specific Web application and closely mimics published (de-facto) standards and specifications for the Web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The WIM defines a general communication model, and, based on it, Web systems consisting of Web browsers, DNS servers, and Web servers as well as Web and network attackers.

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

As usual in DY models (see, e.g., [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

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

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

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

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

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

A *(DY) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (more formally, *derived* in the usual DY style, e.g., by application of function symbols and the equational theory) 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 description of browsers below). Similarly to an attacker process, the so-called *attacker script* may output everything that is derivable from its input.

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

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

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

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

4.2 Extensions of the WIM

To model the FAPI 2.0++ protocols, we extend the WIM with models for the very recent HTTP Message Signatures [3] standard, as well as a channel for push messages to users, like the ones sent by the AS to the user's authentication device in FAPI-CIBA (see Section 2.4). While created in the context of analyzing FAPI 2.0, these extensions are not tailored towards FAPI 2.0 and hence of independent interest, e.g., for [47, 70, 84]. For example, the extensions for HTTP Message Signatures simply follow the respective specifications, including parts not used by our FAPI 2.0++ model, e.g., the option to bind a signed response to the corresponding signed request. We briefly sketch how we model these extensions and refer to the appendix for full details.

HTTP Message Signatures. With HTTP Message Signatures [3], HTTP clients and servers can sign their HTTP requests and responses, while accounting for common transformations applied to HTTP messages during transmission, e.g., headers added by proxies. Our model of HTTP Message Signatures closely follows the specification: We add HTTP headers for the signature input (specifying signature parameters and which parts of the HTTP message in which order form the signature base), the signature value, and the Content-Digest header for message digests of the HTTP message body. The model for these new headers includes their values' internal structure, e.g., for specifying signature parameters, and precisely specifies how they are processed by servers and browsers. We define generic functionality to validate signed HTTP messages which we use in our server and browser models. Note that while [52] contains a limited model of HTTP Message Signatures in the WIM, the model there is tailored towards the analyzed GNAP protocol and, more importantly, models an early draft of the HTTP Message Signatures specification with a known vulnerability, which has recently been fixed. We model the fixed version, which is currently in the final editing stages for publication as an internet standard.

Channel for Push Messages to Users. Since the WIM subsumes the user in its browser model, we extend the WIM's browser with 1) augmented input event processing to enable receiving pushed messages and 2) generic functions that can be instantiated by the application model to process pushed messages appropriately. And 3), we add DNS names and corresponding TLS keys for browsers. These extensions allow any other process in a Web system to send push messages to any browser, both encrypted and in plain text. See Algorithms 30, 31, and 32, as well as Definition 80.

HTTP Headers. In addition to the HTTP headers for HTTP Message Signing, we extend the WIM's HTTP message model with the headers required for modeling *OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)* [32] (the `DPoP` header), and *JWT Response for OAuth Token Introspection* [72] whereby an RS can request a signed introspection response by setting the `Accept` header appropriately.

4.3 Overview of FAPI 2.0++ Model

We build our application-specific model in the (extended) WIM, based on the FAPI 2.0++ specifications, including specifications referenced there, e.g., [3, 32, 68, 71, 72, 89, 92], totaling over 1,000 pages even when excluding low-level specifications, e.g., on message encoding. For this purpose, we instantiate the generic HTTPS server model provided by the WIM to create detailed formal

models for ASs, clients, and RSs, including models of scripts. Furthermore, we instantiate the WIM's browser model to process pushed messages and handle the binding message for FAPI-CIBA flows.

In addition to DCR and DCM, our formal model covers all essential mechanisms used by FAPI 2.0++, e.g., AS Metadata, PAR, PKCE, the mTLS and `private_key_jwt` client authentication methods, DPoP and mTLS access token sender constraining, ID tokens, and the signing mechanisms for FAPI 2.0 MS, i.e., JAR, JARM, the JWT introspection response mode, and HTTP Message Signatures. Our model also covers both structured and opaque access tokens—which can be different for each flow (the token type is chosen non-deterministically for each token request)—as well as token introspection. The final model reflects the FAPI 2.0++ specifications with all changes/fixes described in Section 3.

Using client authentication and access token sender constraining as examples, we exemplify how our model closely follows the specifications and show (parts of) some concrete messages within our model. We then continue with details on how we incorporated the attacker model and fixes described in Section 3 into our formal model, followed by a discussion of the limitations of our model. More details specific to the AS, client, RS, and browser models are given in Appendix B and we provide the full formal models in Appendix C.

Client Authentication. Within the model, the ASs only accept PAR and token requests if they are client-authenticated, as mandated by the specifications [29, Sec. 5.3.1.2. No. 4 and Sec. 5.3.1.1. No. 6], and clients always add client authentication when sending such requests [71, Sec. 2], [29, Sec. 5.3.2.1. No. 2]. For this, our model supports both mTLS and `private_key_jwt` authentication [29, Sec. 5.3.1.1. No. 6 and Sec. 5.3.2.1. No. 2]. To illustrate how we model the `private_key_jwt` method, we show a client authentication JWT. Let *cid* be the client identifier of client `C` at AS `AS`, and let iss_{AS} be the issuer identifier of `AS`. Furthermore, let sk_{C} be a private signing key of `C` such that the corresponding public key $\mathsf{pub}(sk_{\mathsf{C}})$ is registered with `AS` for `C` under *cid*. With these values, a client authentication JWT is—closely following the specification [90, Sec. 9]—represented by the term $\mathsf{sig}([\texttt{iss:}\ cid, \texttt{sub:}\ cid, \texttt{aud:}\ iss_{\mathsf{AS}}], sk_{\mathsf{C}})$, where the `iss` field denotes the issuer of the JWT, i.e., `C`, `sub` is the subject that is being authenticated, and `aud` is the audience value (i.e., the AS for which the JWT was created).

Access Token Sender Constraining. Our AS model returns access tokens that are sender-constrained by mTLS or DPoP [29, Sec. 5.3.1.1. No. 4, 5], and the client model sends the token request to the AS and the resource request to the RS with the corresponding proof-of-possession [29, Sec. 5.3.2.1. No. 1]. As mandated by FAPI 2.0, our RS model requires access tokens to be sender-constrained using one of these methods [29, Sec. 5.3.3. No. 5]. We illustrate how we model a DPoP proof in a token request, where a client `C` adds a DPoP proof to the HTTP headers of its request [32, Sec. 4.1]: $headers[\texttt{DPoP}] := dpopProof$, where *dpopProof* is a signed JWT as defined in [32, Sec. 4.2] (not to be confused with a client authentication JWT): $dpopProof := \mathsf{sig}(dpopJwt, sk_{\mathsf{C}})$ with $dpopJwt := [\texttt{headers:}\ [\texttt{jwk:}\ \mathsf{pub}(sk_{\mathsf{C}})], \texttt{payload:}\ [\texttt{htm:}\ \texttt{POST}, \texttt{htu:}\ tokenEndpoint]]$. The `jwk` value identifies the public key that the receiver can use for verifying the signature,[2] the `htm` value is the HTTP method of the request in which the DPoP proof is included, e.g., `GET` or `POST`, and the `htu` value is the URL of that request (without parameters and fragment).

4.3.1 Optional Fixes. As explained in Section 3, the FAPI WG decided to make some fixes resulting from our attacks optional. In order to prove FAPI 2.0++ secure, these fixes are required, hence, our results only apply to implementations with these fixes in place. Here we describe how we deal with them in the model.

[2]Note that the AS/RS checks whether this public key is indeed registered for the client and rejects the request if not, i.e., the `jwk` value is only a hint.

Cuckoo's Token Attack. As described in Section 3.5, the working group added (optional) countermeasures for preventing the attack. However, none of these fixes are mandatory, and each of them might mask other attacks. Thus, we decided to model a minimal fix that specifically targets this attack: right before sending the resource request to an RS, the client model checks whether the requested resource is managed by the AS from which it got the access token.

DPoP Proof Replay. To prevent this attack, we modeled replay protection using server-provided DPoP nonces (see also Section 3.3). We require these nonces to be one-time use only, i.e., the RS model invalidates them after one use.

Cross-Device Consent Phishing Attack. In Section 3.6, we mention that this attack can only be fixed if the CD authenticates itself toward the user, and explain why such authentication is problematic in practice. However, there is an ongoing effort from the OAuth Working Group to standardize mitigations [62], e.g., by ensuring close proximity between the user's authentication device and the CD. Assuming that the user and their authentication device are "close" to at most one CD at a time, this mitigation basically yields CD authentication. Hence, we assume the mitigations to be effective and use an authenticated channel for conveying the binding message from the CD to the user (Step 5 in Figure 2), i.e., the user learns the CD's identity, which the user can compare to the identity displayed by the AS during user authentication (Step 9 in Figure 2).

4.3.2 FAPI 2.0 Attacker Model. We initially modeled all attacker assumptions (see Section 2.7), but as described in Section 3, the FAPI WG decided to remove some of them in response to our analysis. We describe how we modeled the final assumptions (under which our proofs hold) and refer to Appendix B.5 for the removed ones:

$\mathcal{A}1$ (network attacker) is part of the WIM, see Section 4.1.

$\mathcal{A}2$ (authorization requests leak) is modeled by leakage at the client: After creating the authorization request URI (to which the client redirects the browser in Step 8 of Figure 1), the client non-deterministically decides whether to leak it.[3] In the leak case, it non-deterministically chooses an IP address and sends a copy of URI to this IP (in plain), which the attacker, being a network attacker, can read. Thus, the client model leaks the client identifier and the request URI value of the authorization request.

$\mathcal{A}5$ (resource requests leak) was initially modeled by leaking the resource request at the RS after receiving the request (and in the case of opaque access tokens, before receiving the introspection response). As described in Section 3.3, this enables the attacker to replay DPoP proofs. In line with changes to FAPI 2.0 AM (see Section 3.3), we adapted the model such that the resource request now leaks *after* the RS sent the corresponding resource response.

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

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

Modeled Grant Types. The OAuth 2.0 framework [46], as well as OpenID Connect [90], define various grant types, such as the implicit grant, the hybrid grant, and the authorization code grant. FAPI 2.0 explicitly excludes all of them, except for the client credentials grant and the authorization

[3]Recall from Section 3.7 that we show session integrity only for protocol flows in which this request does not leak.

code grant, where only the latter is required to be supported by FAPI 2.0 ASs and clients. In the client credentials grant, the client takes the role of the user, hence, removing the additional interaction between user and client. So, this grant is subsumed by the authorization code grant, which is why we only model and analyze the latter.

Refresh Tokens. Even though FAPI 2.0 allows for refresh tokens (see [46]) to be used, we do not model them. Instead, we model regular access (and id) tokens as having an indefinite lifetime, i.e., they never expire, which only increases the attack surface.

Native Clients with Loopback Redirect. FAPI 2.0 ASs must reject PAR requests with a redirect uri using the http scheme, i.e., where the client's redirection endpoint lacks TLS protection (cf. Step 5 in Figure 1); except if the client is a native client running on the same device as the browser and is using loopback interface (i.e., device-local) redirection, it may use an http redirect uri (see [16] and [29, Sec. 5.3.1.2 No. 8]). In our AS model, we do not allow http redirect uris.

4.4 Comparison to WIM Analyses of Related Protocols

We here describe specific differences between our work and previous WIM analyses of FAPI 1.0 [33], OAuth 2.0 [34], and OpenID Connect [38]. Since FAPI 2.0++ comprises completely redesigned and new protocols, obviously, previous analyses do not apply to FAPI 2.0++. The differences are also apparent by the new vulnerabilities and attacks we have found on FAPI 2.0++.

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

Similarly, while FAPI 1.0, the predecessor of FAPI 2.0 SP, has been formally analyzed in [33], FAPI 2.0 SP is a very different protocol: in comparison to FAPI 1.0, several security mechanisms have been removed, most notably *OAuth Token Binding* [58] and *JWT Secured Authorization Response Mode (JARM)* [68] (although JARM is used in FAPI 2.0 MS, it is not part of FAPI 2.0 SP). On the other hand, FAPI 2.0 SP adds new mechanisms, such as DPoP and PAR, which have not undergone any formal treatment so far. Also, FAPI 2.0 SP mandates client authentication, removes support for public clients as well as the hybrid flow, and FAPI 2.0 AM introduces a stronger attacker model compared to FAPI 1.0. Furthermore, FAPI-CIBA and FAPI 2.0 MS serve additional use cases that were not even considered for FAPI 1.0 (or OAuth 2.0 or OpenID Connect, for that matter).
In the following, we briefly discuss further details and differences.

DCR and DCM. While previous work on OpenID Connect [38] considered DCR in the context of OpenID Connect, the model in [38] only supports client authentication using *client secrets* issued by the AS upon registration, whereas our model (in line with the FAPI 2.0++ specifications) requires public-private-key client authentication, i.e., clients have to register their public keys. As a result, client authentication in our model depends on the client's registration request, whereas in [38], client authentication only depends on values selected by the AS (client secret and client id). Furthermore, our model also covers DCM, which was not considered in [38]. Finally, the attacker model in [38] is a standard DY attacker, i.e., considerably weaker than ours.

Handling of Access Tokens by the RS. Previous work on OAuth 2.0 and OpenID Connect does not consider the RS at all, and while prior work on FAPI 1.0 does, token introspection is modeled only on a very abstract level: instead of querying the AS, the RS uses an idealized introspection oracle. In contrast, our AS and RS models contain a detailed representation of token introspection as defined in RFC 7662 [83], including authentication of RSs to the ASs during introspection, as mandated by the specification. Hence, we reason about additional messages as well as their integrity, authentication, and secrecy. Likewise, prior work does not contain a model of structured access

tokens and their use. Note that besides the additional options for flows introduced by structured access tokens, modeling them also requires additional keys to be distributed.

Number of ASs Supported by an RS. As mentioned above, prior work on OAuth 2.0 and OpenID Connect did not model the RS. In the FAPI 1.0 analysis in [33], the RS supports one fixed AS, whereas our RS model supports an arbitrary set of ASs, some of which may be corrupted. Hence, our analysis considers a broader set of cases and potential mix-ups.

DPoP Access Token Sender Constraining. The DPoP [32] mechanism has only recently been standardized, and hence, has not been part of any formal analysis to the best of our knowledge. With its additional signatures and subtle details regarding the signed data, in particular, the exact contents of the `htu` value (see [32, Sec. 4.2]), DPoP requires careful modeling of the AS, the RS, and the client. In addition, DPoP requires key material to be distributed between client, AS, and RS (as well as additions to the token introspection and structured access token models).

PAR. Similar to DPoP above, PAR [71] has only recently been standardized and has thus not been part of any formal analysis. While PAR does not require additional key material, it adds more options and messages to the FAPI 2.0 flows, hence necessitating reasoning about them.

JWT Client Authentication. While the `public_key_jwt` client authentication method has been standardized as an optional part of OpenID Connect [90], it has not received any formal security analysis so far, including WIM analyses of OpenID Connect and FAPI 1.0 (FAPI 1.0 is based on OpenID Connect). Hence, our work is the first to consider this kind of client authentication.

Attacker Model. Compared to prior WIM work on OAuth 2.0, OpenID Connect, and FAPI 1.0, our analysis considers a significantly stronger attacker model. Where [34] and [38] consider a fairly standard Dolev-Yao attacker (i.e., similar to $\mathcal{A}1$, see Section 2.7), [33] considers a stronger attacker: the attacker in [33] is a network attacker similar to our $\mathcal{A}1$, which also may access authorization requests (cf. $\mathcal{A}2$) and authorization responses (cf. $\mathcal{A}3$) in plain, as well as force honest clients to use attacker-controlled token endpoints (cf. $\mathcal{A}4$). In addition, the authors of [33] consider access tokens leaking to the attacker, which is something that is subsumed by our attacker assumption $\mathcal{A}5$ (resource request leak). However, our attacker model is even stronger in that (1) $\mathcal{A}5$ not only leaks the access token, but also all other values in the resource request, in particular DPoP proofs, (2) $\mathcal{A}6$ leaks resource responses, and (3) $\mathcal{A}7$ even allows for the attacker to modify resource responses. To the best of our knowledge, neither (1), nor (2), nor (3) have previously been considered in any formal analysis of authorization protocols.

Security Properties. While prior works on OAuth 2.0 and OpenID Connect do consider session integrity properties similar to ours, these are only proven in a setting with *Web attackers*, i.e., the attacker may corrupt any party, but does not control the network and cannot spoof sender addresses. Similar properties have also been proven for FAPI 1.0 *with* a network attacker. However, their proofs rely on the use of *OAuth 2.0 Token Binding* [58], a mechanism to bind authorization codes and access tokens to TLS connections, which is quite different from DPoP and mTLS access token sender constraining. Furthermore, our analysis is the first WIM analysis that formalizes and proves non-repudiation properties.

4.5 FAPI 2.0 Web System

As outlined in Section 4.1, a Web system formalizes the overall system covered by our analysis. We call $(\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ a *FAPI Web system with network attacker*, or short $\mathcal{FAPI}$, if the components of the Web system are defined as follows: $\mathcal{W}$ contains the network attacker process, as well as an arbitrary, but finite number of browsers, ASs, clients, and RSs; $\mathcal{S}$ contains login and index "websites"

of clients and ASs, with script mapping them to strings. Finally, E^0 contains an infinite number of trigger events for all process addresses (see Section 4.1).

Note that we prove security properties about all such FAPI Web systems, and hence, systems with an arbitrary number of browsers, clients, ASs, and RSs, and in which each party can have an arbitrary number of parallel, interleaving protocol sessions. Also recall from Section 4.1 that the attacker can corrupt honest parties at any time.

4.6 Formal Security Properties

As described in Section 2.6, the FAPI WG wants the protocols to meet authorization, authentication, and session integrity goals, with additional non-repudiation properties for FAPI 2.0 MS. In the following, we describe our formalized authorization, authentication, and one of the non-repudiation security properties, capturing the corresponding security goals. The remaining properties for session integrity for authorization and authentication, as well as non-repudiation are presented in Appendix E. Note that whenever possible we give full formal definitions. However, some definitions require the full formal model, which we cannot state here in full detail due to space constraints. For such definitions, we provide simplified and more abstract descriptions (explicitly marked as such), which, nevertheless, should suffice to understand the rest of the paper. As mentioned, for full details we refer to the appendix.

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

For our authorization property, we need the notion of an access token t being bound to a public key k, an authorization server $\mathtt{AS}$, a client id *cid*, and an identity *id*:

Definition 1 (Access Token Bound to Key, AS, Client Id, Identity). *Let $\mathcal{FAPI}$ be a FAPI Web system with network attacker, $k \in \mathcal{T}_{\mathcal{N}}$ a term, $\mathtt{AS} \in \mathsf{AS}$ an AS, cid a client identifier, and $id \in \mathsf{ID}$ a user identity.*[4] *We say that a term t is an* access token bound to k, $\mathtt{AS}$, *cid*, and *id in configuration (S, E, N) of a run ρ of $\mathcal{FAPI}$, if there is an entry $rec \in^{\langle\rangle} S(\mathtt{AS}).\mathtt{records}$ (i.e., in the state of $\mathtt{AS}$ in (S, E, N)) such that $rec[\mathtt{access\_token}] \equiv t$ and $rec[\mathtt{subject}] \equiv id$ and $rec[\mathtt{client\_id}] \equiv cid$ and $(rec[\mathtt{cnf}] \equiv [\mathtt{jkt}\colon \mathtt{hash}(k)]) \vee (rec[\mathtt{cnf}] \equiv [\mathtt{x5t\#S256}\colon \mathtt{hash}(k)])$.*

Informally, this means that t is stored in the state of $\mathtt{AS}$, together with the identity *id*, the client identifier *cid*, and the hash of k. If the key is stored under the name $\mathtt{jkt}$, then the token is bound via DPoP, otherwise, it is bound via mTLS.

Furthermore, we need the following definition:

Definition 2 (Client Identifier Issued to Client by AS (simplified)). *We say that a* client identifier *cid* has been issued to $\mathtt{C}$ by $\mathtt{AS}$ in processing step P in a run ρ *(of a FAPI Web system $\mathcal{FAPI}$), if during P, $\mathtt{AS}$ responds to a registration request regReq with a registration response containing cid, and regReq was emitted by $\mathtt{C}$ during a processing step Q prior to P in ρ.*

We can now define our authorization property:

[4] ID is a set of terms of the form $\langle name, domain\rangle$, where *name* is a string, the user name, and *domain* is a domain (usually of an AS). We also define a mapping $\mathrm{ownerOfID}\colon \mathsf{ID} \to \mathsf{B}$ which maps an identity to the browser whose user owns the identify (users are modeled as part of their browser). Likewise, we define a mapping $\mathrm{governor}\colon \mathsf{ID} \to \mathsf{AS}$ mapping identities to "their" AS. See Appendix C.2 and Appendix C.4 for details.

Definition 3 (Authorization Property (simplified)). *We say that a FAPI Web system with network attacker $\mathcal{FAPI}$* is secure w.r.t. authorization *iff for all runs $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of $\mathcal{FAPI}$, every RS* $\mathsf{RS} \in \mathsf{RS}$ *that is honest in S^n, every identity id for which* RS *manages resources,*[5] *if $b = \mathsf{ownerOfID}(id)$ is an honest browser in S^n, we have that whenever* RS *provides access to, i.e., emits, a resource r associated with id that is managed by an honest AS* AS*, during a processing step Q (in ρ), then all of the following hold true:*

(i) RS *has received a request for accessing the resource r with an access token at_{hon} in Q (if the token at_{hon} is structured and can be verified by the RS immediately) or in a previous processing step (if the token at_{hon} is opaque to the RS and it thus performed token introspection).*

(ii) The token at_{hon} is bound to some key k, AS*, a client identifier cid, and the user identity id (see Definition 1).*

(iii) If there is a client C *such that cid has been issued to* C *by* AS *(see Definition 2) during a processing step prior to Q in ρ, and* C *is honest in S^n, then the attacker cannot derive the resource r.*

4.6.2 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 state, under the `sessions` 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.

Once again, we first need an additional definition, capturing that a client logged in an identity *id* (managed by AS) under a service session id:

Definition 4 (Service Sessions). *We say that there is a* service session identified by a nonce *n* for a user identity *id* at a client C *in a configuration (S, E, N) of a run ρ of a FAPI Web system with network attacker $\mathcal{FAPI}$, iff there exists a session id x (an internal reference in the client model to identify a protocol session, not to be confused with a service session id) and a domain $d_{\mathsf{AS}} \in \mathsf{dom}(\mathsf{governor}(id))$ (where $\mathsf{dom}(p)$ is the set of domains owned by process p) such that $S(\mathsf{C}).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d_{\mathsf{AS}}, id\rangle$ and $S(\mathsf{C}).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$.*

With this, we can define the authentication property:

Definition 5 (Authentication Property). *We say that a FAPI Web system with network attacker $\mathcal{FAPI}$* is secure w.r.t. authentication *iff for every run ρ of $\mathcal{FAPI}$, every configuration (S, E, N) in ρ, every* $\mathsf{C} \in \mathsf{C}$ *that is honest in S, every identity id* $\in \mathsf{ID}$ *with* $\mathsf{AS} = \mathsf{governor}(id)$ *being an honest AS in S and with $b = \mathsf{ownerOfID}(id)$ being an honest browser in S, every service session identified by some nonce n for id at* C*, n is not derivable from the attackers knowledge in S.*

4.6.3 Non-Repudiation for Signed Authorization Requests. Informally, the non-repudiation for signed authorization requests security goal states that if an honest AS accepts a pushed authorization request it expects to be signed, then that request is signed, and if the signature is valid for a key registered with the AS by an honest client, then that client cannot deny having signed the request.

Definition 6 (Non-Repudiation for Signed Authorization Requests (simplified)). *Let $\mathcal{FAPI}$ be a FAPI Web system with a network attacker. We say that $\mathcal{FAPI}$* is secure w.r.t. non-repudiation

[5]Each RS RS in our model is responsible for resources of an arbitrary set of identities. Note that we do not make any assumptions on this set, and in particular, that multiple RSs may hold resources for a given identity.

for signed authorization requests *iff for every run* ρ *of* $\mathcal{FAPI}$*, every configuration* (S^n, E^n, N^n) *in* ρ*, every process* $\mathsf{AS} \in \mathsf{AS}$ *that is honest in* S^n*, every request uri requestUri, we have that if*

$$S^n(\mathsf{AS}).\mathtt{authorizationRequests}[\mathit{requestUri}][\mathtt{signed\_par}] \equiv \top$$

(i.e., AS *expected a signature on the pushed authorization request that* AS *refers to by requestUri) then all of the following hold true:*

(I) There exists a processing step Q prior to (S^n, E^n, N^n) *in* ρ*, such that* AS *accepts a pushed authorization request signedPAR during Q and issues requestUri to identify that PAR. Technically speaking, this means that during Q,* AS *adds requestUri to the* authorizationRequests *subterm in its state.*

(II) The message signedPAR is of the form $\mathrm{sig}(\mathit{par}, \mathit{signKey})$*, i.e., signed with some key signKey.*

(III) If there is a (client) process $\mathsf{C} \in \mathsf{C}$ *which is honest in* S^n*, and a configuration* (S^i, E^i, N^i) *in* ρ *with* $S^i(\mathsf{C}).\mathtt{asAccounts}[\mathit{selectedAS}][\mathtt{sign\_key}] \equiv \mathit{signKey}$*, i.e.,* C *registered signKey with* AS*, then there is a processing step P in* ρ *prior to Q during which* C *signed par.*

4.6.4 Security Theorem. As described in Section 2.6, the protocols aim to fulfill authorization, authentication, and session integrity properties, along with non-repudiation properties for FAPI 2.0 MS. Thus, our overall security theorem is the conjunction of all these properties:

Theorem 1. *Every FAPI Web system with network attacker* $\mathcal{FAPI}$ *fulfills all of the following properties:*

- *Authorization (Definition 19),*
- *Authentication (Definition 21),*
- *Session integrity for authentication in authorization code flows (Definition 29),*
- *Session integrity for authentication in CIBA flows (Definition 30),*
- *Session integrity for authorization in authorization code flows (Definition 31),*
- *Session integrity for authorization in CIBA flows (Definition 32),*
- *Non-repudiation for signed authorization requests (Definition 33),*
- *Non-repudiation for signed authorization responses (Definition 34),*
- *Non-repudiation for signed introspection responses (Definition 35),*
- *Non-repudiation for signed resource requests (Definition 36), and*
- *Non-repudiation for signed resource responses (Definition 37).*

We highlight that we prove this theorem for the powerful attacker described in Section 4.3.2 within a faithful formal model that includes the fixes described in Section 3 and Section 4.3.1, with all analyzed protocols (FAPI 2.0 SP & MS, FAPI-CIBA, DCR, DCM) running in parallel, hence accounting for all potential interferences. 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,[6] several security-critical headers, as well as subtleties of various cookie attributes, which, for example, could result in vulnerable session management, and in-browser communication using postMessages, just to name a few of the Web features considered in our analysis. Thus, our analysis excludes attacks that arise from features of the Web infrastructure. In addition to stronger results, the features provided by the WIM make it easy to faithfully model the functional aspects of FAPI 2.0 in detail, since features used in FAPI 2.0 like HTTP redirects and HTTP headers, e.g., for various types of cookies (used for state management), and the authorization header are built into the WIM, including how browsers deal with such features. Also, without modeling such features, even the most basic kinds of common

[6]For example, FAPI 2.0 excludes code 307 redirects, as they would cause attacks similar to [34].

Web attacks, such as CSRF attacks, would go unnoticed if FAPI 2.0 did not properly defend against them. Our proof of Theorem 1, which, due to space limitations, we give in Appendix F, consists of more than 40 lemmas with over 50 pages of proofs and of course reasons about the full formal model that we provide in Appendix C.

4.7 Proofs: Overview of Selected Lemmas

In this section, we give an impression of the proofs by informally discussing important intermediate statements that we identified and proved. We refer to Appendix F for the full proof.

(1) **Private signature keys of clients do not leak.** We show that private signature keys used by a client for the `private_key_jwt` authentication method and DPoP sender constraining never leak to the attacker as long as the client is honest.[7] This is easy to show, as initially, clients store only fresh nonces as private keys when registering at an AS using DCR. The same is true whenever clients update their configuration at an AS using DCM. Furthermore, except for sending the corresponding public keys, clients only use the private keys for generating signatures, from which the private key cannot be extracted. (See Lemma 4 for details.)

(2) **mTLS private keys of clients do not leak.** In our model, clients use different mTLS keys for different ASs and can update these keys using DCM. We show that within the model, the client always chooses fresh nonces as private keys and only uses them for decrypting messages. The only terms related to these keys that the client sends out are the corresponding public keys, from which the private key cannot be extracted. (See Lemma 6 for details.)

(3) **Registration access tokens do not leak.** Recall from Section 2.5 that ASs issue registration access tokens that clients have to use to modify their configuration via DCM. We prove that registration access tokens issued by ASs for clients do not leak as long as both parties are honest: Registration access tokens are fresh nonces chosen by the AS when processing the registration request. We show that the client sends this token to the same AS for DCM and that neither party leaks this token during DCR and DCM. (See Lemma 16 for details.)

(4) **Client keys stored at ASs do not leak.** Statements 1 and 2 state that keys stored at the client do not leak. Here, this property is stated from the perspective of the AS, i.e., the corresponding private keys of the mTLS and JWT public keys that an AS stores under a client identifier cannot be derived by any party other than the client c to which the identifier was initially issued to (Definition 2), as long as the client and AS are honest. The proof relies on Statements 1 and 2, but also on 3. We show that initially, the AS stores the corresponding public keys from the DCR request, which was created by c as the corresponding client identifier was issued to c in this step. For this case, the property follows (mostly) from Statements 1 and 2. These public keys stored by the AS can only be changed via DCM. From Statement 3, it follows that the corresponding registration access token cannot leak, i.e., whenever the AS updates the keys, the corresponding DCM request was created by c. The property follows again (mostly) from Statements 1 and 2. (See Lemma 17 for details.)

(5) **Client authentication.** We show that requests sent to the PAR, token, or backchannel authentication endpoints of an honest AS were indeed created by the client c that authenticated via mTLS or JWS client assertion (more precisely, the client to which the client identifier in the request was issued as in Definition 2, if this client is honest). For this, we show the effectiveness of the two client authentication mechanisms required by the specifications (see Section 2.2), i.e., if the checks done by the AS are successful, the request is successfully

[7] Within the model, we assume that clients pre-register their DPoP keys.

authenticated via mTLS or contains a JWS client assertion that only c and the AS can derive. The proofs of both cases are based on Statement 4. (See Lemma 13 for details.)

(6) **Access tokens can only be used by the client to which the token is bound.** This is an important lemma due to the assumption that access tokens can leak to the attacker (see Section 4.3.2), i.e., without an effective token binding mechanism, the attacker could get access to the resources of honest users. We show that whenever an honest resource server rs provides access to a resource of an honest identity id (see Step 20 of Figure 1), and if the resource is managed by an honest AS AS, then (i) rs previously received a resource request (see Step 18 of Figure 1) which contains an access token at created by AS bound to some key, AS, some client identifier cid, and id (Definition 1), and (ii) if c is an honest client to which cid has been issued to, then c created the resource request. For the proof, we distinguish whether the token is structured (the RS can directly respond to the resource request) or opaque (the RS first needs to perform token introspection at AS, as shown in Figure 1). For structured access tokens, we show that the token was created by AS if the RS verifies the signature of the token successfully. The token contains the identity id and information on the key to which the token is bound (which both follow as AS is honest according to the precondition of the statement). The token endpoint (at the AS) requires client authentication, and we show that the AS associates the authenticated client identifier with the token, and if this client identifier was issued to an honest client, then the corresponding DPoP signing key or mTLS private key is only known to c (recall from Section 2.2 that the token request must contain proof of possession of a key pair via DPoP or mTLS for token binding). For DPoP sender constraining, we show that the DPoP proof verified by the RS cannot be known by any party other than c and the RS (recall from Section 4.3.2 that we assume that resource requests, including DPoP proofs that they contain, leak after the RS sends the resource response; in combination with resource server-provided nonces, we can rule out replay attacks as we describe in Section 3.3), and conclude that the resource request was sent by c. For this proof, we use Statement 1 again. For mTLS token binding, we show that there is a private mTLS key that only c knows (by using Statement 4), and show that the proof of possession checks done by the RS are sufficient to conclude that the resource request was created by c. The reasoning for the case of token introspection (instead of a structured access token) is similar and again relies essentially on the checks done by the AS at the token endpoint and by the RS when processing the resource request and introspection response. (See Lemma 18 for details.)

(7) **Authorization code secrecy.** We show that authorization codes stored at an honest AS for a flow with an honest client c and honest user identity id do not leak to any party other than AS, the client, and the browser of the user. Note that we show this only for valid authorization codes (recall from Section 2.2 that the AS invalidates authorization codes received at the token endpoint). We start by showing that authorization codes are fresh nonces chosen by the AS after processing the authorization request, more precisely, right before creating the authorization response (Step 11 of Figure 1). The AS associates this code with id, thus, we can show that the browser of id sent the login request (Step 10 of Figure 1), and that the authorization response is sent to this browser. We show that the authorization response contains a location redirect to a URL of c (essentially, because the AS retrieved this URL from a PAR request that the AS processed previously. Due to client authentication – we again use Statement 5 – it follows that c created the PAR request, and as c is honest, this must be a redirect URL of c). Then, we show that c uses the code contained in the authorization response only within token requests sent to the same AS that issued the code. The proof also considers the correctness of the token endpoint by reasoning on the AS metadata retrieved

by c (i.e., the token endpoint that the client stored for the AS is indeed the token endpoint of the AS). We then conclude the proof by showing that the AS does not store or send out the code that is part of the token request. (See Lemma 22 for details.)

To give a complete impression of a significant part of the proofs, we now give a brief description of the authorization proof for the authorization code flow case: On a high-level view, we first show that if the resource server provides access to some resource of an honest user u and if the resource is managed by an honest authorization server AS, then the access token contained in the resource request is bound to a key, AS, some client identifier cid, and u (see Definition 1), and if cid has been issued to an honest client c, then the RS provides access to the resource only to c, which follows essentially from Statement 6. However, this is not sufficient as the client gives the end-user involved in the flow access to the resource. Thus, in a second step, we show that the authorization response that c receives at the redirection URI (see Step 12 of Figure 1) is not sent by the attacker (the client provides the sender of this message access to the resource). We prove this by contradiction: Assume that the authorization response is created by the attacker, i.e., the attacker can derive the authorization code ac contained in the response. The client exchanges ac for the access token that it uses for the resource request (see Step 14 of Figure 1), and as described in Section 4.3.1, the client checks whether it receives the access token from the AS that manages the resource, i.e., AS (which prevents the Cuckoo's Token Attack, see also Section 3.5). As this is an honest AS (see the pre-conditions of the property), and as the AS returns an access token, it follows that ac was created by AS and is associated with c an u. However, this contradicts Statement 7, i.e., such an authorization code cannot leak to the attacker.

5 RELATED WORK

We first discuss related work regarding models of the Web infrastructure and then regarding the analysis of FAPI 2.0++ and related protocols.

Models of the Web Infrastructure. Given the importance of the Web, there is surprisingly little work on generic formal models of the Web infrastructure for protocol analysis. Instead, Web protocols are often analyzed in isolation or with ad-hoc models that only capture parts of the Web infrastructure [11, 67, 78] – both approaches are prone to missing attacks introduced by seemingly unrelated Web features as demonstrated, e.g., by the discovery of the *307 Redirect Attack* on OAuth 2.0 by Fett et al. [34] after several formal analyses of OAuth 2.0 had already been conducted and missed this attack (see below). An early work by Groß et al. [45] focused on defining a limited browser model based on state machines with pen-and-paper proofs to analyze authentication protocols. However, their browser model lacks crucial parts like models for cookies and JavaScript. Later work by Akhawe et al. [2] employs the Alloy model checker and introduces a machine-checkable, albeit limited, model of the Web infrastructure and shows how it can be used to find vulnerabilities in a Kerberos-based SSO system. Their model not only lacks many security-critical Web features but is also limited to a very small number of protocol sessions. Further models of the Web and its infrastructure, such as those by Pai et al. [78], Kumar [63, 64], or Bansal et al. [5] have similar drawbacks in that their models are very abstract or completely miss key parts of the Web infrastructure, as they are limited by the tools upon which they are built. In recent work by Veronese et al. [100], the authors propose a model of Web browsers based on the Coq proof assistant and the Z3 theorem prover and apply it to discover attacks on browser security mechanisms. While capturing the browser in great detail, their model is focused on browser-side security mechanisms and cannot be used to analyze Web protocols involving other parties like Web servers. Finally, our work is based on the Web Infrastructure Model proposed by Fett et al. in [35], which as described

above is a symbolic Dolev-Yao style pen-and-paper model and the most comprehensive and detailed model of the Web infrastructure to date.

Related Protocols. While there has not been much work on the relatively recent and partly still under development FAPI 2.0++ standards and specifications so far, standards like OAuth 2.0 and OpenID Connect, on which FAPI is based, have received quite some attention from security researchers. Besides many studies on implementations and deployments [94, 96, 101, 102], OAuth 2.0 has been formally analyzed: Pai et al. [78] built a limited model of OAuth 2.0, lacking many generic Web features, for the Alloy finite-state model checker and showed that with their approach, known weaknesses can be found. Chari et al. [11] analyzed the authorization code flow in the UC model and found no attacks, but their model omits many Web features. Two more comprehensive formal analyses have been conducted by Bansal et al. [5, 6], using their WebSpi library and ProVerif, in which they modeled various settings of OAuth 2.0, e.g., with CSRF vulnerabilities in ASs and clients. Bansal et al. uncovered several previously unknown attacks on various popular OAuth 2.0 implementations. However, their work's main focus was on finding attacks, rather than proving security. The latter was the focus of a formal analysis by Fett et al. [34], in which, despite prior formal analysis efforts, they discovered several new attacks, and were only able to prove security after developing fixes for them. As in our work, the analysis by Fett et al. is based on the WIM, i.e., includes a comprehensive formal model of the Web infrastructure. Likewise, OpenID Connect has been analyzed before [66, 75], including an analysis based on the WIM [38].

As described in more detail in Section 4.4 where we also discuss prior work on FAPI 1.0, FAPI 2.0++ comprises completely redesigned as well as new protocols, and hence, previous analyses on related protocols do not apply. Finally, as already mentioned, FAPI 2.0 and other protocols and specifications considered here, like DCM and CIBA, have not undergone any formal security analysis so far, let alone being analyzed in combination.

6 CONCLUSION

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

Performing such an analysis in a meaningful model like the WIM ensures that attacks based on many threats and features of the Web and their complex interactions can be found and ruled out (see also our discussion at the end of Section 4.6). As mentioned in the introduction, our analysis is a pen-and-paper analysis, as creating a mechanized model of the Web is a challenge by itself and left for future work. Hence, our proofs are not mechanized and, given the complexity of the protocol, are inherently lengthy and tedious to verify. Nevertheless, our formal and systematic analysis has helped improve the standards and gain more insights and confidence in their security. We consider mechanized analyses of Web protocols as interesting future work.

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

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

ACKNOWLEDGMENTS

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

REFERENCES

[1] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. In *ACM POPL*, 104–115.
[2] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. 2010. Towards a Formal Foundation of Web Security. In *CSF 2010*, 290–304. DOI: 10.1109/CSF.2010.27.
[3] Annabelle Backman, Justin Richer, and Manu Sporny. 2023. HTTP Message Signatures. Internet-Draft draft-ietf-httpbis-message-signatures-19. Internet Engineering Task Force. 118 pp.
[4] Banco Central do Brasil. [n. d.] Open Finance. https://www.bcb.gov.br/en/financialstability/open_finance.
[5] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2014. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22, 4, 601–657. Lieven Desmet, Martin Johns, Benjamin Livshits, and Andrei Sabelfeld, editors. DOI: 10.3233/jcs-140503.
[6] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. 2012. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *IEEE CSF*. DOI: 10.1109/csf.2012.27.
[7] Robin Berjon et al., eds. 2014. HTML5, W3C Recommendation. (2014). http://www.w3.org/TR/html5/.
[8] Vittorio Bertocci. 2021. JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens. RFC 9068. (2021).
[9] 2022. Budget tracker & planner. https://mint.intuit.com/.
[10] Brian Campbell, John Bradley, Nat Sakimura, and Torsten Lodderstedt. 2020. OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. RFC 8705. (2020).
[11] Suresh Chari, Charanjit Jutla, and Arnab Roy. 2011. Universally Composable Security Analysis of OAuth v2.0. Cryptology ePrint Archive, Paper 2011/526. https://eprint.iacr.org/2011/526. (2011). https://eprint.iacr.org/2011/526.
[12] Lily Chen, Steven Englehardt, Mike West, and John Wilander. 2021. Cookies: HTTP State Management Mechanism. Internet-Draft draft-ietf-httpbis-rfc6265bis-09. Internet Engineering Task Force. 59 pp.
[13] Commonwealth of Australia. 2022. Consumer Data Standards. https://consumerdatastandards.gov.au/.
[14] Bobby Cooke. 2021. The Art of the Device Code Phish. https://0xboku.com/2021/07/12/ArtOfDeviceCodePhish.html.
[15] Credit Sense. [n. d.] Bank Account Aggregation Services. https://www.creditsense.com.au/bank-account-aggregation.
[16] William Denniss and John Bradley. 2017. OAuth 2.0 for Native Apps. RFC 8252. (2017).
[17] 2021. Die elektronische Patientenakte. https://www.bundesgesundheitsministerium.de/elektronische-patientenakte.
[18] Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, Nils Wenzler, and Tim Würtele. 2022. A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification. In *IEEE S&P*, 215–234. DOI: 10.1109/SP46214.2022.9833681.
[19] Danny Dolev and Andrew Yao. 1983. On the Security of Public-Key Protocols. *IEEE Trans. Inf. Theory*, 29, 2, 198–208.
[20] Dropbox Platform Team. 2020. OAuth Guide. https://developers.dropbox.com/oauth-guide.
[21] 2022. Electronic Prescription Service - FHIR API. https://digital.nhs.uk/developer/api-catalogue/electronic-prescription-service-fhir.
[22] 2021. FAPI 2.0 Profile Transition. https://github.com/ConsumerDataStandardsAustralia/future-plan/issues/47.
[23] Gonzalo Fernandez, Florian Walter, Axel Nennker, Dave Tonge, and Brian Campbell. 2021. OpenID Connect Client-Initiated Backchannel Authentication Flow - Core 1.0. OpenID Foundation, (2021). https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html.
[24] Daniel Fett. 2018. *An Expressive Formal Model of the Web Infrastructure*. PhD thesis.
[25] Daniel Fett. 2022. Authorization Request Leaks lead to CSRF. https://bitbucket.org/openid/fapi/issues/534.
[26] Daniel Fett. 2022. Change attacker model to reflect formal model. https://bitbucket.org/openid/fapi/pull-requests/381.
[27] Daniel Fett. 2022. FAPI 2.0 Attacker Model. Second implementer's draft. OpenID Foundation, (2022). https://openid.net/specs/fapi-2_0-attacker-model-02.html.

[28] Daniel Fett. 2022. FAPI 2.0 Attacker Model, Commit 209f58a. OpenID Foundation, (2022). https://bitbucket.org/openid/fapi/src/209f58afbd41fb20ab3ed65ca4e2f67ffd5dda77/FAPI_2_0_Attacker_Model.md.

[29] Daniel Fett. 2022. FAPI 2.0 Security Profile. Second implementer's draft. OpenID Foundation, (2022). https://openid.net/specs/fapi-2_0-security-profile-ID2.html.

[30] Daniel Fett. 2022. Improve attacker model description after introduction of metadata. https://bitbucket.org/openid/fapi/pull-requests/371.

[31] Daniel Fett. 2022. Reduced attacker model. https://bitbucket.org/openid/fapi/pull-requests/377.

[32] Daniel Fett, Brian Campbell, John Bradley, Torsten Lodderstedt, Michael B. Jones, and David Waite. 2023. OAuth 2.0 Demonstrating Proof of Possession (DPoP). RFC 9449. (2023).

[33] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. 2019. An Extensive Formal Security Analysis of the OpenID Financial-grade API. In *IEEE S&P*, 1054–1072. DOI: 10.1109/SP.2019.00067.

[34] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *ACM CCS*. DOI: 10.1145/2976749.2978385.

[35] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2014. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *IEEE S&P*, 673–688.

[36] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2015. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. In *ESORICS*. Vol. 9326, 43–65.

[37] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2015. SPRESSO: a secure, privacy-respecting single sign-on system for the web. In *ACM CCS*. DOI: 10.1145/2810103.2813726.

[38] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2017. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. In *IEEE CSF*.

[39] Daniel Fett and Dave Tonge. 2023. FAPI 2.0 Message Signing. Commit 67246ac. OpenID Foundation, (2023). https://bitbucket.org/openid/fapi/src/67246ac44d2ee136c184789b9757ba44df57c7b8/fapi-2_0-message-signing.md.

[40] [n. d.] Financial Data Exchange (FDX). https://financialdataexchange.org/.

[41] GitHub, Inc. 2022. Scopes for OAuth Apps. https://docs.github.com/en/developers/apps/building-oauth-apps/scopes-for-oauth-apps.

[42] Google. 2022. OAuth 2.0 Scopes for Google APIs. https://developers.google.com/identity/protocols/oauth2/scopes.

[43] Google. 2022. YouTube Data API Overview. https://developers.google.com/youtube/v3/getting-started.

[44] Thomas Groß. 2003. Security analysis of the SAML single sign-on browser/artifact profile. In *CSAC 2003*, 298–307. DOI: 10.1109/CSAC.2003.1254334.

[45] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. 2005. Browser Model for Security Analysis of Browser-Based Protocols. In *ESORICS 2005*, 489–508. ISBN: 978-3-540-31981-8. DOI: 10.1007/11555827_28.

[46] Dick Hardt. 2012. The OAuth 2.0 Authorization Framework. RFC 6749. (2012).

[47] Dick Hardt, Aaron Parecki, and Torsten Lodderstedt. 2024. The OAuth 2.1 Authorization Framework. Internet-Draft draft-ietf-oauth-v2-1-11. Internet Engineering Task Force. 96 pp.

[48] Joseph Heenan. 2022. Discovery should be mandated for clients. https://bitbucket.org/openid/fapi/issues/536.

[49] Joseph Heenan. 2022. Dpop & resource leaks. https://bitbucket.org/openid/fapi/issues/533.

[50] Joseph Heenan. 2022. FAPI2SP: Add requirement for RP to use discovery. https://bitbucket.org/openid/fapi/pull-requests/363.

[51] Joseph Heenan. 2022. FAPI2SP: Add security consideration for cuckoo's token attack. https://bitbucket.org/openid/fapi/pull-requests/364.

[52] Florian Helmschmidt, Pedram Hosseyni, Ralf Küsters, Klaas Pruiksma, Clara Waldmann, and Tim Würtele. 2023. The Grant Negotiation and Authorization Protocol: Attacking, Fixing, and Verifying an Emerging Standard. In *ESORICS 2023*. DOI: 10.1007/978-3-031-51479-1_12.

[53] Pedram Hosseyni, Ralf Küsters, and Tim Würtele. 2024. Formal Security Analysis of the OpenID FAPI 2.0: Accompanying a Standardization Process. In *CSF 2024*. To appear.

[54] Pedram Hosseyni, Ralf Küsters, and Tim Würtele. 2022. Formal Security Analysis of the OpenID Financial-grade API 2.0. Tech. rep. WP 1(b). OpenID Foundation. https://openid.net/wordpress-content/uploads/2022/12/Formal-Security-Analysis-of-FAPI-2.0_FINAL_2022-10.pdf.

[55] Jenko Hwong. 2021. New Phishing Attacks Exploiting OAuth Authentication Flows. DEF CON 29. https://www.youtube.com/watch?v=9slRYvpKHp4.

[56] Michael Jones, John Bradley, and Nat Sakimura. 2015. JSON Web Signature (JWS). RFC 7515. (2015).

[57] Michael Jones, John Bradley, and Nat Sakimura. 2015. JSON Web Token (JWT). RFC 7519. (2015).

[58] Michael Jones, Brian Campbell, John Bradley, and William Denniss. 2018. OAuth 2.0 Token Binding. Internet-Draft draft-ietf-oauth-token-binding-08. Internet Engineering Task Force. 30 pp.

[59] Michael Jones, Brian Campbell, and Chuck Mortimore. 2015. JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. RFC 7523. (2015).

[60] Michael Jones, Brian Campbell, Chuck Mortimore, and Yaron Y. Goland. 2015. Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants. RFC 7521. (2015).

[61] Michael Jones, Nat Sakimura, and John Bradley. 2018. OAuth 2.0 Authorization Server Metadata. RFC 8414. (2018).

[62] Pieter Kasselman, Daniel Fett, and Filip Skokan. 2023. Cross-Device Flows: Security Best Current Practice. Internet-Draft draft-ietf-oauth-cross-device-security-04. Internet Engineering Task Force. 53 pp.

[63] Apurva Kumar. 2014. A Lightweight Formal Approach for Analyzing Security of Web Protocols. In *Research in Attacks, Intrusions and Defenses*. Springer International Publishing, 192–211. DOI: 10.1007/978-3-319-11379-1_10.

[64] Apurva Kumar. 2012. Using automated model analysis for reasoning about security of web protocols. In *ACM ACSAC*. DOI: 10.1145/2420950.2420993.

[65] Ralf Küsters, Guido Schmitz, and Daniel Fett. 2022. The Web Infrastructure Model (WIM). Technical Report. Version 1.0. https://www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf.

[66] Wanpeng Li and Chris J. Mitchell. 2016. Analysing the Security of Google's Implementation of OpenID connect. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 357–376. DOI: 10.1007/978-3-319-40667-1_18.

[67] Xinyu Li, Jing Xu, Zhenfeng Zhang, Xiao Lan, and Yuchen Wang. 2020. Modular Security Analysis of OAuth 2.0 in the Three-Party Setting. In *EuroS&P 2020*, 276–293. DOI: 10.1109/EuroSP48549.2020.00025.

[68] Thorsten Lodderstedt and Brian Campbell. 2018. Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM). OpenID Foundation, (2018). https://openid.net/specs/openid-financial-api-jarm.html.

[69] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. 2019. OAuth 2.0 Security Best Current Practice. (2019). https://datatracker.ietf.org/meeting/105/materials/slides-105-oauth-sessa-oauth-security-topics-00.pdf.

[70] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. 2021. OAuth 2.0 Security Best Current Practice. Internet-Draft. Internet Engineering Task Force. 52 pp.

[71] Torsten Lodderstedt, Brian Campbell, Nat Sakimura, Dave Tonge, and Filip Skokan. 2021. OAuth 2.0 Pushed Authorization Requests. RFC 9126. (2021).

[72] Torsten Lodderstedt and Vladimir Dzhuvinov. 2021. JWT Response for OAuth Token Introspection. Internet-Draft draft-ietf-oauth-jwt-introspection-response-12. Internet Engineering Task Force. 19 pp.

[73] Meta. 2022. email – Graph API. https://developers.facebook.com/docs/permissions/reference/email.

[74] Ministério da Economia do Brasil. [n. d.] Open Finance. https://www.gov.br/susep/pt-br/assuntos/open-insurance.

[75] Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. 2015. On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect. *CoRR*, abs/1508.04324v2. arXiv: 1508.04324v2 [cs.CR].

[76] 2022. Open Banking UK. https://www.openbanking.org.uk/.

[77] OWASP Top 10. 2021. Security Misconfiguration. https://owasp.org/Top10/A05_2021-Security_Misconfiguration/.

[78] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika Pai, and Sanjay Singh. 2011. Formal Verification of OAuth 2.0 using alloy framework. In *IEEE CSNT*. DOI: 10.1109/csnt.2011.141.

[79] Aaron Parecki. 2020. OAuth 2.0 Simplified: Token Introspection Endpoint. https://www.oauth.com/oauth2-servers/token-introspection-endpoint/.

[80] Payments NZ. [n. d.] Governance of NZ payment systems. https://www.paymentsnz.co.nz/.

[81] Julian Reschke. 2015. The 'Basic' HTTP Authentication Scheme. RFC 7617. (2015).

[82] Eric Rescorla and Tim Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. (2008).

[83] Justin Richer. 2015. OAuth 2.0 Token Introspection. RFC 7662. (2015).

[84] Justin Richer and Fabien Imbault. 2024. Grant Negotiation and Authorization Protocol. Internet-Draft draft-ietf-gnap-core-protocol-20. Internet Engineering Task Force. 234 pp.

[85] Justin Richer, Michael Jones, John Bradley, Maciej Machulak, and Phil Hunt. 2015. OAuth 2.0 Dynamic Client Registration Protocol. RFC 7591. (2015).

[86] Justin Richer, Michael B. Jones, John Bradley, and Maciej Machulak. 2015. OAuth 2.0 Dynamic Client Registration Management Protocol. RFC 7592. (2015).

[87] Nat Sakimura. 2022. Browser swap attack explained on 2022-09-28. https://bitbucket.org/openid/fapi/issues/543.

[88] Nat Sakimura. 2022. Decide on what to do for A. Cuckoo's Token Attack. https://bitbucket.org/openid/fapi/issues/525.

[89] Nat Sakimura, John Bradley, and Naveen Agarwal. 2015. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636. (2015).

[90] Nat Sakimura, John Bradley, M. Jones, B. de Medeiros, and C. Mortimore. 2014. OpenID Connect Core 1.0 incorporating errata set 1. OpenID Foundation, (2014). http://openid.net/specs/openid-connect-core-1_0.html.

[91] Nat Sakimura, John Bradley, M. Jones, and E. Jay. 2014. OpenID Connect Discovery 1.0 incorporating errata set 1. OpenID Foundation, (2014). http://openid.net/specs/openid-connect-discovery-1_0.html.

[92] Nat Sakimura, John Bradley, and Michael Jones. 2021. The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR). RFC 9101. (2021).

[93] 2018. Services at Helsenorge. https://www.helsenorge.no/om-tjenestene/slik-brukes-tjenestene-paa-helsenorge.
[94] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. 2015. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 239–260. DOI: 10.1007/978-3-319-20550-2_13.
[95] Smartcar. 2022. Car API platform for connected vehicle data. https://smartcar.com/.
[96] San-Tsai Sun and Konstantin Beznosov. 2012. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *ACM CCS*. DOI: 10.1145/2382196.2382238.
[97] Dave Tonge. 2023. CIBA - Make clear limitation of binding message. https://bitbucket.org/openid/fapi/issues/609.
[98] Dave Tonge. 2023. FAPI Client Initiated Backchannel Authentication Profile. Commit f3390ad. OpenID Foundation, (2023). https://bitbucket.org/openid/fapi/src/f3390ad/Financial_API_WD_CIBA.md.
[99] 2023. verimi. https://verimi.de/en/.
[100] Lorenzo Veronese, Benjamin Farinier, Pedro Bernardo, Mauro Tempesta, Marco Squarcina, and Matteo Maffei. 2023. WebSpec: Towards Machine-Checked Analysis of Browser Security Mechanisms. In *SP 2023*, 2761–2779. DOI: 10.1109/SP46215.2023.10179465.
[101] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. 2016. Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations. In *ASIA CCS*. DOI: 10.1145/2897845.2897874.
[102] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *USENIX*, 495–510.
[103] Karsten zu Selhausen and Daniel Fett. 2022. OAuth 2.0 Authorization Server Issuer Identification. RFC 9207. (2022).

A ADDITIONAL ATTACKS AND ATTACK DETAILS

In this appendix, we give additional attacks, as well as additional details on the attacks presented in Section 3.

A.1 Variants of the Client Impersonation Attack

We discovered and reported two additional variants of this attack: Recall that in the original attack described above, the user authenticated at an honest, trusted AS and authorized not only an honest and trusted client but also exactly the client that the user expected to authorize, after starting a flow with $\mathsf{C}_{\mathsf{hon}}$, i.e., an honest client of the user's choice, for which the attacker needs leaked client authentication JWTs.

In the first variant and additionally assuming that users do not pay close attention to the client they are authorizing, the attack becomes much more viable in practice: the attacker can now select an arbitrary (but honest) client for which it can obtain client authentication JWTs. We note that the additional assumption about the user's behavior is not a particularly strong one: even if the user pays attention, the AS also has to gather and show enough (reliable) information about the client, which can be quite difficult in practice, especially in dynamic environments, in which the information an AS has on its clients are provided by the clients themselves, e.g., via DCR (see Section 2.5).

In yet another variant of the attack, depicted in Figure 6, the attacker once again uses leaked client authentication JWTs (Step 1) to act as $\mathsf{C}_{\mathsf{hon}}$ towards $\mathsf{AS}_{\mathsf{hon}}$, but sends a PAR without waiting for the user to initiate a flow (Step 2). As before, the attacker then constructs a link and uses social engineering to make the user click on that link (Steps 4 and 5). The remainder of the attack is the same as above, resulting in a violation of the authorization goal. We note that this attack variant assumes that the attacker can convince the user to not only click a link and authenticate at $\mathsf{AS}_{\mathsf{hon}}$ but also to authorize $\mathsf{C}_{\mathsf{hon}}$ even though the user did not start a flow at all (however, $\mathsf{AS}_{\mathsf{hon}}$ and $\mathsf{C}_{\mathsf{hon}}$ may be entities the user knows and trusts).

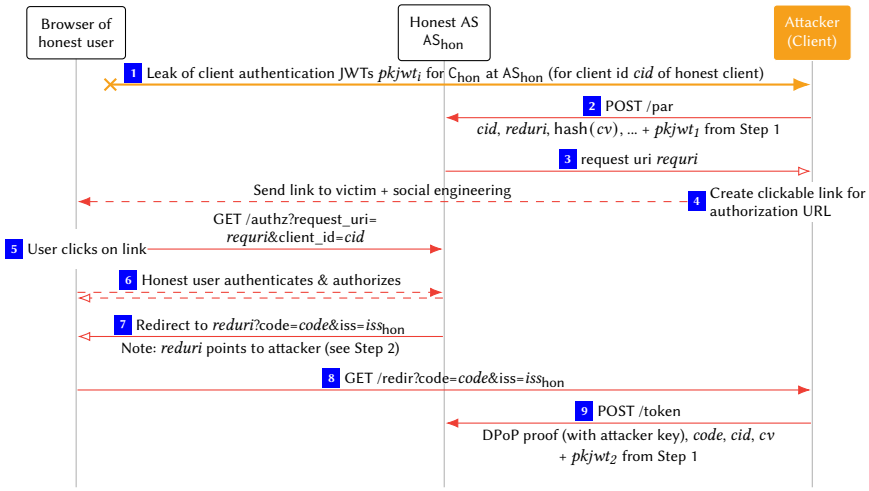

Fig. 6. Variant of client impersonation attack

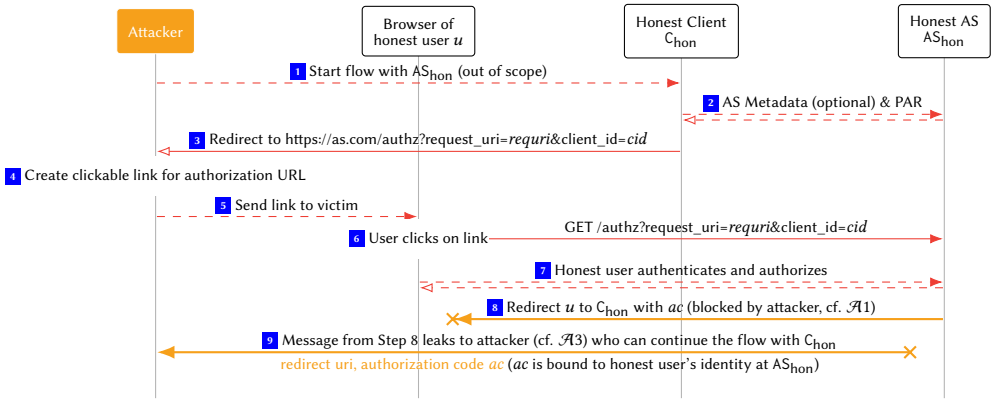

Fig. 7. Browser swapping attack

A.2 The Browser Swapping Attack in Detail

As described in Section 3.4, this attack violates the authorization and authentication goals by combining attacker assumptions $\mathcal{A}1$ (network attacker) and $\mathcal{A}3$ (authorization responses leak, see Section 2.7).

The detailed attack flow is depicted in Figure 7: in Step 1, the attacker initiates a flow at $\mathsf{C}_{\mathsf{hon}}$ with $\mathsf{AS}_{\mathsf{hon}}$. Then, $\mathsf{C}_{\mathsf{hon}}$ and $\mathsf{AS}_{\mathsf{hon}}$ exchange the PAR as described in Section 2.2 (Step 2). Following this, $\mathsf{C}_{\mathsf{hon}}$ instructs its user, i.e., the attacker, to visit the authorization endpoint of $\mathsf{AS}_{\mathsf{hon}}$ with the request uri *requri* from Step 2 and $\mathsf{C}_{\mathsf{hon}}$'s client id *cid*. However, instead of following this redirect, the attacker creates a clickable link, pointing to $\mathsf{AS}_{\mathsf{hon}}$'s authorization endpoint, with *requri* and *cid* as parameters (Step 4). The attacker then sends this link to u in Step 5, e.g., in an email or as part of an attacker website.

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

Now recall $\mathcal{A}3$: through a leaking authorization response, the attacker can obtain the authorization code *ac* (Step 9). With *ac*, the attacker visits $\mathsf{C}_{\mathsf{hon}}$'s redirect uri, so $\mathsf{C}_{\mathsf{hon}}$ can subsequently exchange *ac* for an access (and id) token at $\mathsf{AS}_{\mathsf{hon}}$. From $\mathsf{C}_{\mathsf{hon}}$'s point of view, these tokens are associated with $\mathsf{C}_{\mathsf{hon}}$'s session with the attacker. However, due to Step 7, these tokens are issued for the (honest user) u's resources (and identity).

A.3 Variant of Cross-Device Consent Phishing Attack on FAPI-CIBA

Recall the cross-device consent phishing attack on FAPI-CIBA from Section 3.6 where an attacker violates the authentication and authorization goals by posing as a client towards an honest user while simultaneously posing as that user towards an honest client. In the attack described there, the attacker has to wait for an honest user to initiate a protocol flow with an attacker-controlled CD.

In a variant of this attack, the attacker does not wait for the honest user to initiate a flow and instead uses social engineering to convey the binding message and a fabricated explanation for the honest AS' request to authorize something, e.g., via an email supposedly sent by $\mathsf{AS}_{\mathsf{hon}}$ saying that the user's account is about to be disconnected from $\mathsf{C}_{\mathsf{hon}}$ if the user does not confirm the connection.

Note that the user may have indeed connected $\mathsf{C}_{\mathsf{hon}}$ to their account at $\mathsf{AS}_{\mathsf{hon}}$ before and that the authorization granted during the attack is indeed for $\mathsf{C}_{\mathsf{hon}}$.

B DETAILS ON APPLICATION-SPECIFIC MODEL

In this appendix, we continue the overview of our formal model from Section 4.3 with additional details and references to the respective specifications, demonstrating how our model closely follows the specifications.

B.1 Authorization Server Model

Pushed Authorization Request. We model the PAR endpoint as part of the AS (mandated by FAPI 2.0 [29, Sec. 5.3.1.2. No. 2]). The endpoint model stores all values and returns a (freshly chosen) request URI value. In particular, as mandated by FAPI 2.0, the model requires a redirect URI and a PKCE code challenge value in the PAR request [29, Sec. 5.3.1.2. No. 5, 6].

Token Introspection. The introspection endpoint part of the AS model expects an opaque access token and returns whether the token is active (and thus, if the token was issued by this AS at all), information on the key to which the token is bound, and a subject identifier, i.e., the identity of the user whose login at the AS lead to the AS issuing the token. As required in [83, Sec. 2.1], the AS model requires successful authentication of the RS that sent the request. The exact authentication mechanism for RSs at ASs is out of scope for FAPI 2.0++. In our model, we use the HTTP Basic Authentication mechanism suggested by [83].

Login Script. Upon receiving the authorization request for a FAPI 2.0 SP flow, the AS model responds with a script that models the login page for the user. As a result of executing the script, the browser sends a POST request to the AS with the login credentials (for the current AS) of a user that is using the browser.

A similar script is used for FAPI-CIBA flows. We describe this script in Appendix B.4.

Dynamic Client Registration. For DCR, our AS model includes an endpoint that expects client registration messages. These messages must be HTTPS POST requests [85, Sec. 3], containing client metadata like the client's redirect URIs and public keys [85, Sec. 2]. When processing such a request, the AS selects a fresh client id, registration access token (see Section 2.5 and [86, Sec. 1.2, 1.3(D)]), the supported grant types [85, Sec. 2.1], and other values, registers the client with these values, and sends a response with these values back to the client [85, Sec. 3.2.1].

Dynamic Client Management. Similar to DCR (see above), DCM defines an additional AS endpoint [86, Sec. 2] to which clients send requests to change their configuration at an AS. Such requests must be HTTPS PUT [86, Sec. 2.2] (to change client configuration) or HTTP DELETE [86, Sec. 2.3] (to unregister a client) requests. Furthermore, these request must include the registration access token (see Section 2.5 and [86, Sec. 1.2, 1.3(D)]), and, for configuration updates, the new values [86, Sec. 2.2].

Similar to DCR, the AS processes these requests and either responds with the updated information in case of an update request [86, Sec. 3] or deactivates the client's registration [86, Sec. 2.3].

Further Endpoints. The AS model also comprises an endpoint for server metadata that returns information about the server such as the different endpoint URLs. This endpoint is mandated by FAPI 2.0 [29, Sec. 5.3.1.1. No. 1]. The authorization endpoint of the AS model requires a request URI value as mandated in [29, Sec. 5.3.1.2. No. 3]. Furthermore, the model has a JWKS endpoint, where the server responds with the public signature verification key, which is strongly recommended by FAPI 2.0 [29, Sec. 5.6.3].

B.2 Client Model

Configurations. As noted above, the client authenticates either via mTLS or the `private_key_jwt` method, and supports sender constraining by either mTLS or DPoP. In the model, the client can have different combinations of client authentication and sender constraining methods for different ASs and the configuration for a given AS may change due to DCM.

Starting a FAPI 2.0 SP Flow. For starting a FAPI 2.0 SP flow, the client model provides a script for browsers which triggers a POST request to the client. In our model, this request must contain the domain of an AS, modeling a user selecting an AS. When starting a flow for the first time with an AS, the client fetches the server metadata from the AS (as required by FAPI 2.0 [29, Sec. 5.3.2.1. No. 9]), and registers itself at that AS through DCR (with fresh keys for signatures, client authentication, and access token sender constraining).

Starting a FAPI-CIBA Flow. For starting a FAPI-CIBA flow, the client model expects to receive an HTTPS request with an AS domain and an identity to its `/start-ciba` endpoint. If the client did not interact with the selected AS before, it fetches the AS metadata and registers with the AS through DCR (with fresh keys for signatures, client authentication, and access token sender constraining).

Dynamic Client Management. Similar to how browsers handle trigger events (see Section 4.1), the client model non-deterministically performs one of several actions when receiving a trigger event. One of those actions is `CHANGE_CLIENT_CONFIG`, in which the client non-deterministically chooses one of the ASs it is registered with, selects the appropriate registration access token (see Section 2.5 and [86, Sec. 1.2, 1.3(D)]), non-deterministically decides whether to update or delete its registration, and sends an appropriate request to the AS. In case of an update request, the client includes fresh public keys for signatures, client authentication, and access token sender constraining.

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

Further Client Aspects. Furthermore, as FAPI 2.0 SP mandates, the client model always uses pushed authorization requests and PKCE, and always checks the *iss* issuer identifier in the redirection request (at the redirection endpoint) [29, Sec. 5.3.2.2. No. 2, 3, 4].

Another important detail concerns the handling of id tokens: FAPI 2.0 uses OpenID Connect id tokens, which means that id tokens are signed by their issuer. This of course raises the question of why the honest client in the attacker token injection attack accepts an attacker-constructed id token (see Section 3.1). The reason is that OpenID Connect [90, Sec. 3.1.3.7 No. 6] (and thus FAPI 2.0) allows clients to skip signature verification on id tokens if the token is received directly from the token endpoint over a TLS-protected connection – these conditions are always fulfilled with FAPI 2.0. In the attacker token injection attack, the client contacts an attacker-controlled token endpoint (via a TLS-protected connection) and thus does not verify the id token signature.

B.3 Resource Server Model

Verification of Access Token. A request for a resource must contain an access token in the HTTP headers [29, Sec. 5.3.3 No. 1]. The RS model verifies that the token is valid, identifies the resource owner for whose resources the token was issued, and the key to which the token is bound as follows. If the token is a structured token, the RS checks the signature of the token using the public verification key of the AS responsible for the requested resource and retrieves the resource owner information, as well as the key to which the token is bound from the token. Otherwise, the RS model sends the token and RS authentication information to the introspection endpoint of the AS that manages the requested resource. The token introspection response then contains the necessary information.

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

B.4 Browser Model

Push Message Endpoint. The authentication device in FAPI-CIBA needs to be able to receive pushed messages from the AS to initiate user authentication and consent (see Step 9 in Figure 2). Since we model the user as being subsumed by the browser model and the actual authentication and consent is based on browser-AS interaction, we model the authentication device (and its user) as a modified WIM browser. Namely, we extend the WIM's browser model such that it 1) gets assigned a URL and 2) can process HTTP and HTTPS requests.

Said URL is linked to a user's identity at an AS and known to that AS (see Definition 14). Push messages sent by the AS are then modeled as HTTPS requests sent to that URL. Note that such linking of user identity to a URL and sending push messages to that URL is also how (browser) notifications work in the real world.

While our extension of the WIM's browser to receive and process HTTP(S) requests is generic (see Section 4.2), we of course instantiate it for FAPI-CIBA (see Algorithm 31). Upon receiving an encrypted push message, our browser model decrypts it, extracts the contained URL, non-deterministically chooses whether to visit the URL contained in an open or a new browser window, and finally sends a GET request to that URL. In an honest protocol flow, this URL points to a website of the AS at which the user then authenticates (similar to a FAPI 2.0 SP flow).

Initiating Flows. Any FAPI 2.0 flow in our model is initiated by a user, i.e., the browser model subsuming that user, we slightly adapt the WIM's browser model to accommodate FAPI-CIBA flows. Specifically, the WIM's browser model already comes with a non-deterministic choice of actions upon receiving a trigger event (see Section 4.1). One of these actions is to visit a non-deterministically chosen URL. This is already sufficient to initiate FAPI 2.0 SP flows, where the browser initiates a flow by visiting a client website, i.e., sending an HTTPS GET request to path / at that client (where the user then selects the AS to use).

However, for FAPI-CIBA, this initial request (modeling the user's interaction with the CD) needs to already contain a user identity and an AS. Hence, we extend the aforementioned non-deterministic action in the browser model with the non-deterministically activated option to initiate a FAPI-CIBA flow. If that option is active, the browser model non-deterministically selects a domain (intended to be an AS domain) and one of its identities and includes them in the request sent to a non-deterministically chosen URL. Note that this initial message does not contain any secrets.

Handling of FAPI-CIBA's Binding Message. As explained in Section 2.4, the binding message is a random value chosen by the CD that the CD shows to the user and sends to the AS. Later, when the user authenticated at the AS and chooses whether to authorize the client's request, the AS shows this binding message and the user is supposed to compare it against the value shown by the CD. As explained above, the user's initial interaction with the CD is modeled as an HTTPS request from the browser (subsuming the user) to the CD. In the CD's response to that request, it includes that binding message. Hence, we slightly extend the WIM's browser model to extract the binding message from such responses and store it, along with the CD's identity (recall that we model this channel to be authenticated, see Section 4.3.1).

To then compare the stored binding message to the one shown by the AS, we extend the WIM's browser by an additional *script action*. Script actions model user actions like clicking the "back" button and JavaScript APIs such as sending a postMessage. Our new script action is called by a script that the AS returns once the browser visits the AS' authentication website (see "Initiating Flows" above). This action takes as input a client identity and a binding message (both provided by the AS) and checks whether it previously stored such a tuple. If this is the case, the browser sends the user's login credentials to the AS. Otherwise, the browser aborts processing the script.

B.5 Attacker Model

In Section 4.3.2, we describe how we model the attacker assumptions that remained in FAPI 2.0 AM. However, as explained in Section 3, FAPI 2.0 AM originally contained an even stronger attacker model (see Section 2.7), but some of the attacker assumptions where dropped as a result of our findings.

Here, we describe how we modeled these ultimately dropped attacker assumptions (see Section 4.3.2 for the others):

$\mathcal{A}3$ (authorization responses leak) was initially modeled by leakage of the authorization response at the AS, which sent the response in plain to a non-deterministically chosen IP address (in addition to sending the response to the client). As described in Section 3.4, we had to remove this leak according to the changed attacker model.
$\mathcal{A}4$ (attacker can trick client into using an attacker-controlled token endpoint) was initially modeled by the client non-deterministically choosing whether to use the correct token endpoint or a non-deterministically chosen endpoint. After the FAPI WG made it mandatory for clients to use AS metadata, we removed this attacker capability (see Section 3.2).
$\mathcal{A}6$ and $\mathcal{A}7$. As mentioned in Section 3.8, the FAPI WG dropped these attacker assumptions after we reported inconsistencies in the attacker model resulting from these assumptions. Thus, they are not part of our final model anymore.

C 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 instantiate the WIM's browser, how we model resources, and details on how *OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens* [10] is

modeled. We continue with the formal definitions of additional HTTP headers for the WIM and a helper function for HTTP Message Signing, and conclude the section with our formal models of the FAPI 2.0 clients (Appendix C.10), the FAPI 2.0 ASs (Appendix C.11), and the FAPI 2.0 RSs (Appendix C.12).

C.1 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 C.11. RS is the set of processes representing resource servers, described in Appendix C.12. C is the set of processes representing clients, described in Appendix C.10. Finally, B is the set of processes representing browsers, including their users. They are described in Appendix G.7.

C.2 Identities

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

Definition 7. *An* identity i *is a term of the form* $\langle name, domain\rangle$ *with* $name \in \mathbb{S}$ *and* $domain \in \mathsf{Doms}$. *Let* ID *be the finite set of identities. We say that an id is* governed *by the DY process to which the domain of the id belongs. This is formally captured by the mappings* $\mathsf{governor}\colon \mathsf{ID} \to \mathcal{W}$, $\langle name, domain\rangle \mapsto \mathsf{dom}^{-1}(domain)$ *and* $\mathsf{ID}^y := \mathsf{governor}^{-1}(y)$.

C.3 Keys and Secrets

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

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

These sets are used as follows:

- The set N contains the nonces that are available for the DY processes
- The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $tlskeys^p = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(p)\}\rangle$ (i.e., a sequence of pairs).
- The set K_{sign} contains the keys that will be used by ASs for signing id and access tokens, and by clients and RSs to sign HTTP messages. Let $\mathsf{signkey}\colon \mathsf{AS} \times \mathsf{C} \times \mathsf{RS} \to K_{\text{sign}}$ 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 $\mathsf{secretOfID}\colon \mathsf{ID} \to \mathsf{Passwords}$ 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 $\mathsf{secretOfRS}\colon \mathsf{Doms} \times \mathsf{Doms} \rightharpoonup \mathsf{RScredentials}$ be a partial mapping, assigning a secret to some of the RS–AS pairs (with the function arguments in that order).

C.4 Passwords

Definition 8. *Let* $\mathsf{ownerOfSecret}\colon \mathsf{Passwords} \to \mathsf{B}$ *be a mapping that assigns to each password a browser which* owns *this password. Similarly, we define* $\mathsf{ownerOfID}\colon \mathsf{ID} \to \mathsf{B}$ *as* $i \mapsto$

ownerOfSecret(secretOfID(i)), *which assigns to each identity the browser that owns this identity (i.e., this identity belongs to the browser).*

C.5 Web Browsers

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

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

Definition 9 (Initial Web Browser State for FAPI). *The initial state of a Web browser process $b \in \mathsf{B}$ follows the description in Definition 80, with the following additional constraints:*

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

$$s_0^b.\mathtt{secrets} \equiv \left\langle\left\{\langle\langle d, \mathsf{S}\rangle, \langle\mathsf{Secrets}^{b,p}\rangle\rangle \;\middle|\; \exists p, d\colon p \in \mathsf{AS} \cup \mathsf{C} \cup \mathsf{RS} \wedge d \in \mathsf{dom}(p)\right\}\right\rangle$$

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

C.6 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 10. *For each $rs \in \mathsf{RS}$, let* $\mathsf{resourceURLPath}^{rs} \subseteq \mathbb{S}$ *be a finite set of strings. These are the URL paths identifying the resources managed by the RS.*[8]

Definition 11. *For each $rs \in \mathsf{RS}$, let* $\mathsf{supportedAuthorizationServer}^{rs} \subseteq \mathsf{AS}$ *be a finite set of ASs. These are the ASs supported by the RS.*

Definition 12. *For each $rs \in \mathsf{RS}$, let* $\mathsf{authorizationServerOfResource}^{rs}\colon \mathsf{resourceURLPath}^{rs} \to \mathsf{supportedAuthorizationServer}^{rs}$ *be a mapping that assigns an AS to each resource URL path suffix of resources managed by the RS.*

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

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

C.7 Modeling mTLS

OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens (mTLS) [10] 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*[9], 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 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.[10]

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 [33]. An overview of the mTLS model is shown in Figure 8. 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 [82, 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.

[9] As noted in Section 7.2 of [10], this extension supports all TLS versions with certificate-based client authentication.
[10] The RS can read this information either directly from the access token if the access token is a signed document, or uses token introspection to retrieve the data from the AS.

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.

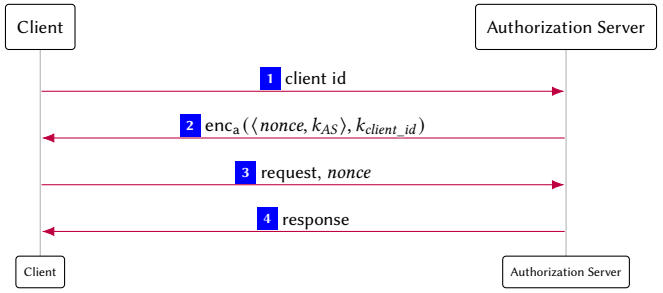

Fig. 8. Overview of mTLS model

All messages are sent by the generic HTTPS server model (Appendix G.12), 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 9 shows the complete messages, i.e., with the encryption used for transmitting the messages.

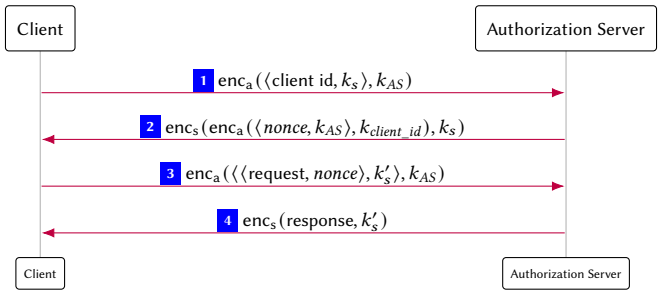

Fig. 9. Detailed view on mTLS model

C.8 Additional HTTP Headers

In order to model FAPI 2.0, we extend the list of headers of Definition 54 with the following headers:

- For DPoP, we add the header $\langle \texttt{DPoP}, p \rangle$ where $p \in \mathcal{T}_{\mathcal{N}}$ is (for honest senders) a DPoP proof (i.e., a signed JWT).
- The `Authorization` header can also take on values $\langle \texttt{Bearer}, t \rangle$ where $t \in \mathcal{T}_{\mathcal{N}}$ is usually a bearer token.
- We add the header $\langle \texttt{Accept}, s \rangle$ with $s \in \mathbb{S}$.
- For HTTP Message Signatures, we add the following headers
  - $\langle \texttt{Signature-Input}, \mathit{inputs} \rangle$ where inputs is a dictionary of elements $\mathit{label}\colon t$ with $t \in \mathcal{T}_{\mathcal{N}}$, $\mathit{label} \in \mathbb{S}$. 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.,

$$\left[\texttt{label1}\colon \left\langle \langle \langle \texttt{@status}, \langle\rangle \rangle, \langle \texttt{content-digest}, \langle \texttt{req} \rangle \rangle \right\rangle, [\texttt{keyid}\colon \texttt{some\_id}] \right\rangle\right]$$

- $\langle$`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.
- $\langle$`Content-Digest`, *digest*$\rangle$ where *digest* $\in \mathcal{T}_{\mathcal{N}}$ is – for honest senders – a hash of the message body.

C.9 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)   → request may be empty (◇)
2:      let componentName := component.1
3:      let componentParam := component.2
4:      let knownComponents := {@method, @target-uri, @status, authorization, content-digest, dpop}
5:      if componentName ∉ knownComponents then
6:          return ⊥
7:      let componentValue := ◇
8:      if componentParam ≡ ⟨⟩ then    → Compare against component value from m
9:          if componentName ≡ @method then
10:             let componentValue := m.method
11:         if componentName ≡ @target-uri then
12:             let componentValue := ⟨URL, m.protocol, m.host, m.path, m.parameters, ⊥⟩
13:         if componentName ≡ @status then
14:             let componentValue := m.status
15:         if componentName ≡ authorization then
16:             let componentValue := m.headers[Authorization]
17:         if componentName ≡ content-digest then
18:             let componentValue := m.headers[Content-Digest]
19:         if componentName ≡ dpop then
20:             let componentValue := m.headers[DPoP]
21:     else if componentParam ≡ ⟨req⟩ then    → Compare against component value from request
22:         if componentName ≡ @method then
23:             let componentValue := request.method
24:         if componentName ≡ @target-uri then
25:             let componentValue := ⟨URL, request.protocol, request.host, request.path, request.parameters, ⊥⟩
26:         if componentName ≡ content-digest then
27:             let componentValue := request.headers[Content-Digest]
28:         if componentName ≡ dpop then
29:             let componentValue := request.headers[DPoP]
30:     else
31:         return ⊥    → Unsupported component parameter
32:     if componentValue ≡ signerSignatureBase[component] then
33:         return ⊤
34:     else
35:         return ⊥
```

C.10 Clients

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

Definition 13. *A* state $s \in Z^c$ of a client c *is a term of the form* $\langle$*DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, sessions, oauthConfigCache, jwksCache, asAccounts, mtlsCache, pendingCIBARequests, resourceASMapping, dpopNonces, jwk, rsSigKeys*$\rangle$ *with* $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{TLS}]$ *(all former components as in Definition 83),* $\mathit{sessions} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{oauthConfigCache} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{jwksCache} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{asAccounts} \in \left[\mathsf{Doms} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$, $\mathit{mtlsCache} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{pendingCIBARequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{resourceASMapping} \in [\mathsf{Doms} \times [\mathbb{S} \times \mathsf{Doms}]]$, $\mathit{dpopNonces} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{jwk} \in K_{sign}$, *and* $\mathit{rsSigKeys} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$.
An initial state s_0^c of c *is a state of* c *with*

- $s_0^c.\mathtt{DNSaddress} \in \mathsf{IPs}$,
- $s_0^c.\mathtt{pendingDNS} \equiv \langle\rangle$,
- $s_0^c.\mathtt{pendingRequests} \equiv \langle\rangle$,
- $s_0^c.\mathtt{corrupt} \equiv \bot$,
- $s_0^c.\mathtt{keyMapping}$ *being the same as the keymapping for browsers,*
- $s_0^c.\mathtt{tlskeys} \equiv \mathit{tlskeys}^c$ *(see Appendix C.3),*
- $s_0^c.\mathtt{sessions} \equiv \langle\rangle$,
- $s_0^c.\mathtt{oauthConfigCache} \equiv \langle\rangle$,
- $s_0^c.\mathtt{jwksCache} \equiv \langle\rangle$,
- $s_0^c.\mathtt{asAccounts} \equiv \langle\rangle$,
- $s_0^c.\mathtt{mtlsCache} \equiv \langle\rangle$,
- $s_0^c.\mathtt{pendingCIBARequests} \equiv \langle\rangle$ *(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),*
- $s_0^c.\mathtt{resourceASMapping}[\mathit{domRS}][\mathit{resourceID}] \in \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(\mathit{resourceID}))$, $\forall\, \mathit{rs} \in \mathsf{RS}$ *and* $\forall\, \mathit{domRS} \in \mathsf{dom}(\mathit{rs})$ *and* $\forall\, \mathit{resourceID} \in \mathsf{resourceURLPath}^{rs}$ *(a domain of the AS managing the resource stored at rs identified by resourceID),*
- $s_0^c.\mathtt{dpopNonces} \equiv \langle\rangle$,
- $s_0^c.\mathtt{jwk} \equiv \mathsf{signkey}(c)$ *(used for HTTP message signing, see Appendix C.3), and*
- $s_0^c.\mathtt{rsSigKeys} \equiv \mathit{rsk}$ *such that* $\mathit{rsk}[\mathit{domRS}] = \mathsf{pub}(\mathsf{signkey}(\mathit{rs}))$ *for all* $\mathit{domRS} \in \mathsf{dom}(\mathit{rs})$ *for all* $\mathit{rs} \in \mathsf{RS}$ *(see [39, 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 G.12). 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 39).

The script that is used by the client on its index page is specified in Algorithm 10. This script uses the GETURL(*tree*, *docnonce*) function to to extract the current URL of a document. We define this function as follows: It searches for the document with the identifier *docnonce* in the (cleaned) tree *tree* of the browser's windows and documents. It then returns the URL u of that document. If no document with nonce *docnonce* is found in the tree *tree*, $\diamond$ is returned.

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

function PROCESS_HTTPS_REQUEST(m, k, a, f, s') → **Process an incoming HTTPS request.** Other message types are handled in separate functions. m is the incoming message, k is the encryption key for the response, a is the receiver, f the sender of the message. s' is the current state of the atomic DY process c.
  if m.path ≡ / **then** → Serve index page (start flow).
    let m' := enc$_s$(⟨HTTPResp, m.nonce, 200, *headers*, ⟨script_client_index, ⟨⟩⟩⟩, k) → Reply with *script_client_index*.
    stop ⟨⟨f, a, m'⟩⟩, s'
  else if m.path ≡ /startLogin ∧ m.method ≡ POST **then** → Start a new FAPI 2.0 flow (see script_client_index)
    if m.headers[Origin] ≢ ⟨m.host, S⟩ **then**
      stop → Check the Origin header for CSRF protection to prevent attacker from starting a flow in the background (as this would trivially violate the session integrity property).
    let *selectedAS* := m.body
    let *sessionId* := ν_1 → Session id is a freshly chosen nonce.
    let s'.sessions[*sessionId*] := [startRequest: [message: m, key: k, receiver: a, sender: f], selected_AS: *selectedAS*, cibaFlow: ⊥]
    call PREPARE_AND_SEND_INITIAL_REQUEST(*sessionId*, a, s') → Start authorization flow with the AS (Algorithm 8)
  else if m.path ≡ /redirect_ep **then** → User is being redirected after authentication to the AS.
    let *sessionId* := m.headers[Cookie][⟨__Host, sessionId⟩]
    if *sessionId* ∉ s'.sessions **then**
      stop
    let *session* := s'.sessions[*sessionId*] → Retrieve session data.
    let *selectedAS* := *session*[selected_AS]
    if *session*[requested_signed_authz_response] ≡ ⊤ **then**
      if checksig(m.parameters[response], s'.jwksCache[*selectedAS*]) ≢ ⊤ **then**
        stop → Invalid or missing signature on authorization response, see JARM [68, Sec. 2.4]
      let *authzResponse* := extractmsg(m.parameters[response])
      if *authzResponse*[aud] ≢ *session*[client_id] **then**
        stop → Wrong/missing audience value, see JARM [68, Sec. 2.4]
      let m.parameters := *authzResponse* → Remove signature (so we always store a "plain" message below)
    else
      let *authzResponse* := m.parameters
    if code ∉ *authzResponse* ∨ iss ∉ *authzResponse* **then**
      stop
    let *code* := *authzResponse*[code]
    let *issuer* := *authzResponse*[iss]
    if *issuer* ≢ *selectedAS* **then** → Check issuer parameter (RFC 9207 [103]).
      stop
    → Store browser's request for use in CHECK_ID_TOKEN (Algorithm 7) and PROCESS_HTTPS_RESPONSE (Algorithm 3)
    let s'.sessions[*sessionId*][redirectEpRequest] := [message: m, key: k, receiver: a, sender: f]
    call SEND_TOKEN_REQUEST(*sessionId*, *code*, a, s') → Retrieve a token from AS's token endpoint.
  else if m.path ≡ /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)
    let *selectedAS* := m.body[authServ]
    let *identity* := m.body[identity]
    let *sessionId* := ν_6 → Session id is a freshly chosen nonce.
    let *bindingMessage* := $\nu_{\text{bindingMsg}}$
    let s'.sessions[*sessionId*] := [selected_AS: *selectedAS*, selected_identity: *identity*, binding_message: *bindingMessage*, start_polling: ⊥, cibaFlow: ⊤]
    → Store record for continuing the flow later upon receiving a trigger message
    let s'.pendingCIBARequests := s'.pendingCIBARequests +$^{\langle\rangle}$ ⟨*sessionId*, a⟩
    let *headers* := [Set-Cookie: [⟨__Host, sessionId⟩: ⟨*sessionId*, ⊤, ⊤, ⊤⟩]]
    let *body* := [binding_message: *bindingMessage*]
    let m' := enc$_s$(⟨HTTPResp, m.nonce, 200, *headers*, *body*⟩, k)
    stop ⟨⟨f, a, m'⟩⟩, s'
  else if m.path ≡ /ciba_notif_ep **then** → CIBA notification endpoint
    let *receivedNotificationToken* := m.headers[Authorization].2
    let *receivedAuthReqId* := m.body[auth_req_id]
    let *sessionId* **such that** *sessionId* ∈ s'.sessions ∧s'.sessions[*sessionId*][client_notification_token] ≡ *receivedNotificationToken* ∧s'.sessions[*sessionId*][client_notification_token] ≢ ⟨⟩ ∧s'.sessions[*sessionId*][auth_req_id] ≡ *receivedAuthReqId* **if possible; otherwise stop**
    call SEND_CIBA_TOKEN_REQUEST(*sessionId*, a, s') → Send a token request

→ Algorithm continues on next page.

```
51:  else if m.path ≡ /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 ∉ s'.sessions then
54:      stop
55:    let session := s'.sessions[sessionId]    → Retrieve session data.
56:    if session[cibaFlow] ≡ ⊥ 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 ∉ session ∧ resource ∉ session then    → User authentication/authorization not finished yet
59:      stop
60:    let headers := [ ]
61:    let body := [ ]
62:    if serviceSessionId ∈ session then
63:      let serviceSessionId := session[serviceSessionId]
64:      let headers[Set-Cookie] := [serviceSessionId: ⟨serviceSessionId, ⊤, ⊤, ⊤⟩]
65:    if resource ∈ session then
66:      let body := session[resource]
67:    let m' := enc_s(⟨HTTPResp, m.nonce, 200, headers, body⟩, k)
68:    stop ⟨⟨f, a, m'⟩⟩, 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] ≡ MTLS then    → Client received an mTLS nonce (see Appendix C.7)
3:          let m_dc, k′ such that m_dc ≡ dec_a(m.body, k′) ∧ selectedAS ∈ s′.asAccounts ∧ s′.asAccounts[selectedAS][tls_key] ≡ k′
                ↪ if possible; otherwise stop
4:          let mtlsNonce, serverPubKey such that m_dec ≡ ⟨mtlsNonce, serverPubKey⟩ if possible; otherwise stop
5:          if serverPubKey ≡ s′.keyMapping[request.host] then    → Verify sender of mTLS nonce
6:              let clientId := reference[client_id]    → Note: If client_id ∉ reference, then reference[client_id] ≡ ⟨⟩
7:              let pubKey := reference[pub_key]    → See note for client ID above
8:              let s′.mtlsCache := s′.mtlsCache +^⟨⟩ ⟨request.host, clientId, pubKey, mtlsNonce⟩
9:          stop ⟨⟩, s′
10:     if reference[responseTo] ≡ CLIENT_MANAGEMENT then
            → Process a client information response [85, 86]. According requests are initiated by trigger messages, see Algorithm 9.
11:         let selectedAS := reference[selected_AS]
12:         let clientId := s′.asAccounts[selectedAS][client_id]    → client_id cannot be changed (see Sec. 2.2 of RFC 7592 [86])
13:         if m.status ≡ 204 ∧ request.method ≡ DELETE then    → Client was deleted at AS (see Sec. 2.3 of RFC 7592 [86])
14:             let s′.asAccounts := s′.asAccounts − selectedAS
15:             stop ⟨⟩, s′
16:         if m.body[client_type] ∉ {mTLS_mTLS, pkjwt_mTLS, mTLS_DPoP, pkjwt_DPoP} then
17:             stop    → Invalid client type
18:         let clientType := m.body[client_type]
19:         if m.body[jwks] ≢ reference[request][jwks] then
20:             stop    → AS changed client's jwks value: abort client metadata update
21:         let regClientUri := m.body[reg_client_uri]
22:         let regAt := m.body[reg_at]
23:         let s′.asAccounts[selectedAS] := [client_id: clientId, client_type: clientType, reg_at: regAt,
                ↪ reg_client_uri: regClientUri, sign_key: reference[sigKey],
                ↪ tls_key: reference[tlsKey], grant_types: m.body[grant_types]]
24:         if backchannel_token_delivery_mode ∈ m.body then
25:             let s′.asAccounts[selectedAS][backchannel_token_delivery_mode] :=
                    ↪ m.body[backchannel_token_delivery_mode]
26:             if m.body[backchannel_token_delivery_mode] ∈ {ping, push} then
27:                 if backchannel_client_notification_endpoint ∉ m.body then
28:                     stop
29:                 let clientNotificationEP := m.body[backchannel_client_notification_endpoint]
30:                 let s′.asAccounts[selectedAS][backchannel_client_notification_endpoint] := clientNotificationEP
31:         stop ⟨⟩, s′
32:     let sessionId := reference[session]
33:     let session := s′.sessions[sessionId]
34:     let selectedAS := session[selected_AS]
        → 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.
35:     if reference[responseTo] ≡ CONFIG then
36:         if m.body[issuer] ≢ selectedAS then    → Verify issuer identifier according to Sec. 3.3 of RFC 8414 [61]
37:             stop
38:         let s′.oauthConfigCache[selectedAS] := m.body
39:         call PREPARE_AND_SEND_INITIAL_REQUEST(sessionId, a, s′)
40:     else if reference[responseTo] ≡ REGISTRATION then
41:         if m.body[client_type] ∉ {mTLS_mTLS, pkjwt_mTLS, mTLS_DPoP, pkjwt_DPoP} then
42:             stop    → Invalid client type
43:         let clientType := m.body[client_type]
44:         let clientId := m.body[client_id]
45:         let regClientUri := m.body[reg_client_uri]    → DCM endpoint of AS Sec. 3 of RFC 7592 [86]
46:         let regAt := m.body[reg_at]    → DCM bearer token Sec. 3 of RFC 7592 [86]
47:         if m.body[jwks] ≢ reference[request][jwks] then
48:             stop    → AS changed client's jwks value: abort registration
            → Note: The jwks 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.
49:         let s′.asAccounts[selectedAS] := [client_id: clientId, client_type: clientType, reg_at: regAt,
                ↪ reg_client_uri: regClientUri, sign_key: reference[sigKey],
                ↪ tls_key: reference[tlsKey], grant_types: m.body[grant_types]]
```

→ 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
                → Client requested a signed authorization response
60:             let s′.sessions[sessionId][requested_signed_authz_response] := ⊤
61:         let requestUri := m.body[request_uri]
62:         let s′.sessions[sessionId][request_uri] := requestUri
63:         let clientId := session[client_id]
64:         let request := session[startRequest]
            → In the following, we construct the response to the initial request by some browser
65:         let authEndpoint := s′.oauthConfigCache[selectedAS][auth_ep]
            → The authorization endpoint URL may include query components, which must be retained while also ensuring that no
              parameter appears more than once (Sec. 3.1 of RFC 6749 [46]). However, following Sec. 4 of RFC 9126 [71] and Sec. 5
              of RFC 9101 [92] closely could introduce duplicates. We opted to overwrite client_id and request_uri parameters if
              present.
66:         let authEndpoint.parameters[client_id] := clientId
67:         let authEndpoint.parameters[request_uri] := requestUri
68:         let headers := [Location: authEndpoint]
69:         let headers[Set-Cookie] := [⟨__Host, sessionId⟩: ⟨sessionId, ⊤, ⊤, ⊤⟩]
70:         let response := enc_s(⟨HTTPResp, request[message].nonce, 303, headers, ⟨⟩⟩, request[key])
71:         let leakAuthZReq ← {⊤, ⊥}    → We assume that the authorization request, in particular request_uri and client_id, may
                                           leak to the attacker, see [27].
72:         if leakAuthZReq ≡ ⊤ then
73:             let leak := ⟨LEAK, authEndpoint⟩
74:             let leakAddress ← IPs
75:             stop ⟨⟨request[sender], request[receiver], response⟩, ⟨leakAddress, request[receiver], leak⟩⟩, s′
76:         else
77:             stop ⟨⟨request[sender], request[receiver], response⟩⟩, s′
78:     else if reference[responseTo] ≡ TOKEN then
79:         let useAccessTokenNow := ⊤
80:         if session[scope] ≡ openid then    → Non-deterministically decide whether to use the AT or check the ID token (if requested)
81:             let useAccessTokenNow ← {⊤, ⊥}
82:         if useAccessTokenNow ≡ ⊤ then
83:             call USE_ACCESS_TOKEN(reference[session], m.body[access_token], request.host, a, s′)
84:         let selectedAsTokenEp := s′.oauthConfigCache[selectedAS][token_ep]
85:         if request.host ≢ selectedAsTokenEp.host then
86:             stop    → Verify sender of HTTPS response is the expected AS (see [90, Sec. 3.1.3.7])
87:         call CHECK_ID_TOKEN(reference[session], m.body[id_token], s′)
88:     else if reference[responseTo] ≡ RESOURCE_USAGE then
            → Construct response to browser's request to the client's redirect endpoint (with the retrieved resource as payload)
89:         let expectSignedResponse ← {⊤, ⊥}    → Choose whether to expect a signed resource response
90:         let s′.sessions[sessionId][expect_signed_resource_res] := expectSignedResponse
91:         if expectSignedResponse ≡ ⊤ then    → Check whether client expects a signed response
92:             if hash(m.body) ≢ m.headers[Content-Digest] then
93:                 stop    → Content-digest is required by FAPI 2.0 Message Signing [39, Sec. 5.6.2.2]
94:             let coveredComponents := m.headers[Signature-Input][res]
95:             let rsDom := request.host    → RS to which the resource request was sent
96:             let pubKey := s′.rsSigKeys[rsDom]
97:             let signerSignatureBase := extractmsg(m.headers[Signature][res])
98:             if @status ∉ coveredComponents.1 ∨ content-digest ∉ coveredComponents.1 ∨
                    ↪ coveredComponents.2[tag] ≢ fapi-2-response then
99:                 stop    → See [39, Sec. 5.6.2.2], these components must be present
100:            if signerSignatureBase.2[tag] ≢ fapi-2-request ∨ keyid ∉ signerSignatureBase.2 then
101:                stop
102:            for component ∈ coveredComponents.1 do
103:                let isComponentEqual := IS_COMPONENT_EQUAL(m, request, signerSignatureBase, component)
104:                if isComponentEqual ≢ ⊤ then
105:                    stop
```

→ Algorithm continues on next page.

```
                → If we make it here, the response signature base matches the actual response data.
106:            if pubKey ≡ ⟨⟩ ∨ checksig(m.headers[Signature][res], pubKey) ≢ ⊤ then
107:                stop    → Invalid public key/message or signature does not verify
108:        let resource := m.body[resource]
109:        let s′.sessions[sessionId][resource] := resource    → Store received resource
110:        let s′.sessions[sessionId][resourceServer] := request.host    → Store the domain of the RS
111:        if session[cibaFlow] ≡ ⊥ then    → Send the resource as a response to the redirection endpoint request.
112:            let request := session[redirectEpRequest]    → Data on browser's request to client's redirect endpoint
113:            let m′ := enc_s(⟨HTTPResp, request[message].nonce, 200, ⟨⟩, resource⟩, request[key])
114:            stop ⟨⟨request[sender], request[receiver], m′⟩⟩, s′
115:        else    → Wait for the browser to send a request with the login session id, see Line 51 of Algorithm 2
116:            stop ⟨⟩, s′
117:    else if reference[responseTo] ≡ DPOP_NONCE then
118:        let dpopNonce := m.body[nonce]
119:        let rsDomain := request.host
120:        let s′.dpopNonces[rsDomain] := s′.dpopNonces[rsDomain] +^⟨⟩ dpopNonce
121:        stop ⟨⟩, s′
122:    else if reference[responseTo] ≡ CIBA_AUTH_REQ then
123:        let authnReqId := m.body[auth_req_id]
124:        let s′.sessions[sessionId][auth_req_id] := authnReqId    → Store received request identifier
            → If the client has registered the poll delivery mode, it can start polling at the token endpoint
125:        if s′.asAccounts[selectedAS][backchannel_token_delivery_mode] ≡ poll then
126:            let s′.sessions[sessionId][start_polling] := ⊤    → Client can start polling
127:        stop ⟨⟩, s′
128:    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 ∉ 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 [89], 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 ∈ {mTLS_mTLS, mTLS_DPoP} then    → mTLS client authentication
16:         let body[client_id] := clientId    → RFC 8705 [10] mandates client_id when using mTLS authentication
17:         let mtlsNonce such that ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
18:         let authData := [TLS_AuthN: mtlsNonce]
19:         let s'.mtlsCache := s'.mtlsCache −^⟨⟩ ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩
20:     else if clientType ∈ {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 ≡ 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 ≡ pkjwt_mTLS then    → mTLS sender constraining (fresh mTLS nonce)
30:         let mtlsNonce such that ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
31:         let s'.mtlsCache := s'.mtlsCache −^⟨⟩ ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩
32:         let body[TLS_binding] := mtlsNonce
33:     else    → Sender constraning using DPoP
34:         let htu := tokenEndpoint
35:         let htu.parameters := ⟨⟩    → [32, Sec. 4.2]: without query
36:         let htu.fragment := ⊥    → [32, 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 +^⟨⟩ authData
42:     let message := ⟨HTTPReq, v2, POST, tokenEndpoint.host, tokenEndpoint.path, tokenEndpoint.parameters, headers, body⟩
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 [10] 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 constraning using DPoP
31:         let htu := tokenEndpoint
32:         let htu.parameters := ⟨⟩    → [32, Sec. 4.2]: without query
33:         let htu.fragment := ⊥    → [32, 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, v2, 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
                                 access token (but no mTLS nonce, nor a DPoP proof for an honest server).
        → Note: All paths except the mTLS and DPoP preparation endpoints are resource paths at the RS.
5:      let resourceID ← 𝕊 such that resourceID ∉ {/MTLS-prepare, /DPoP-nonce}
6:      let url := ⟨URL, S, rsDomain, resourceID, ⟨⟩, ⊥⟩
7:      if s'.resourceASMapping[rsDomain][resourceID] ≠ tokenEPDomain then
8:          stop    → The AS from which the client received the AT is not managing the resource
        → The access token is sender-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 −^⟨⟩ ⟨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] −^⟨⟩ dpopNonce
21:         let htu := url
22:         let htu.parameters := ⟨⟩    → [32, Sec. 4.2]: without query
23:         let htu.fragment := ⊥    → [32, 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 [32, Sec. 7.1]
28:         let headers[DPoP] := dpopProof    → add DPoP header; the dpopJwt can be extracted with the extractmsg() function
29:     let signRequest ← {⊤, ⊥}    → Choose whether to sent a signed resource request
30:     if signRequest ≡ ⊤ then
31:         let clientSignKey := s'.asAccounts[selectedAS][sign_key]
32:         let headers[Content-Digest] := hash(body)    → See [39, Sec. 5.6.1.1 No. 8]
33:         let coveredComponents := ⟨⟨⟨@method, ⟨⟩⟩, ⟨@target-uri, ⟨⟩⟩, ⟨authorization, ⟨⟩⟩, ⟨content-digest, ⟨⟩⟩⟩,
                                      ↪ [tag: fapi-2-request, keyid: pub(clientSignKey)]⟩    → See [39, 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 +^⟨⟩ ⟨dpop, ⟨⟩⟩    → See [39, Sec. 5.6.1.1 No. 7]
37:             let signatureBase[⟨dpop, ⟨⟩⟩] := headers[DPoP]
38:         let signatureBase := signatureBase +^⟨⟩ coveredComponents.2    → Signature parameters, cf. [3, 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, ν3, 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 [90, 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] ≠ selectedAS then
8:          stop   → Check the issuer; note that previous checks ensure oauthConfig[issuer] ≡ selectedAS
9:      if data[aud] ≠ clientInfo[client_id] then
10:         stop   → Check the audience against own client id.
11:     if nonce ∈ session ∧ data[nonce] ≠ session[nonce] then
12:         stop   → If a nonce was used, check its value.
13:     let s′.sessions[sessionId][loggedInAs] := ⟨selectedAS, data[sub]⟩   → User is now logged in. Store user identity and issuer
                                                                              of ID token.
14:     let s′.sessions[sessionId][serviceSessionId] := ν4   → Choose a new service session id.
15:     if session[cibaFlow] ≡ ⊥ 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: ⟨ν4, ⊤, ⊤, ⊤⟩]   → Create a cookie containing the service session id,
                                                                             effectively logging the user identified by data[sub]
                                                                             in at this client.
18:         let m′ := encs(⟨HTTPResp, request[message].nonce, 200, headers, ok⟩, request[key])
19:         stop ⟨⟨request[sender], request[receiver], m′⟩⟩, s′
20:     else   → Wait for the browser to send a request with the login session id, see Line 51 of Algorithm 2
21:         stop ⟨⟩, 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 := {⟨URL, S, d, /redirect_ep, ⟨⟩, ⊥⟩ | d ∈ dom(c)}    → Set of redirect URIs for all domains of c.
3:      let redirectUri ← redirectUris    → Select a (potentially) different redirect URI for each authorization request
4:      let session := s'.sessions[sessionId]
5:      let selectedAS := session[selected_AS]    → AS selected by the user at the beginning of the flow.
        → Check whether the client needs to fetch AS metadata first and do so if required.
6:      if selectedAS ∉ s'.oauthConfigCache then
7:          let path ← {/.well_known/openid-configuration, /.well_known/oauth-authorization-server}
8:          let message := ⟨HTTPReq, ν5, GET, selectedAS, path, ⟨⟩, ⟨⟩, ⟨⟩⟩
9:          call HTTPS_SIMPLE_SEND([responseTo: CONFIG, session: sessionId], message, a, s')
10:     let oauthConfig := s'.oauthConfigCache[selectedAS]
11:     if selectedAS ∉ s'.asAccounts then    → c not yet registered with selectedAS – Dynamic Client Registration (see RFC 7591 [85])
12:         let url := oauthConfig[reg_ep]
13:         let signingKey := ν_cliSignK    → Generate signing key (pair) to use with selectedAS
14:         let tlsKey := ν_cliTlsK    → Generate mTLS key (pair) to use with selectedAS (see also Appendix C.7)
15:         let jwks := ⟨[use: sig, val: pub(signingKey)], [use: TLS, val: pub(tlsKey)]⟩
16:         let regData := [redirect_uris: ⟨redirectUris⟩, jwks: jwks]
17:         let cibaDeliveryMode ← {poll, ping, push}
18:         let regData[backchannel_token_delivery_mode] := cibaDeliveryMode
19:         if cibaDeliveryMode ≡ ping ∨ cibaDeliveryMode ≡ push then
20:             let regData[backchannel_client_notification_endpoint] ← {⟨URL, S, d, /ciba_notif_ep, ⟨⟩, ⊥⟩ | d ∈ dom(c)}
21:         if cibaDeliveryMode ≡ ping ∨ cibaDeliveryMode ≡ poll then
22:             let regData[grant_types] := ⟨authorization_code, urn:openid:params:grant-type:ciba⟩
23:         else
24:             let regData[grant_types] := ⟨authorization_code⟩
25:         let message := ⟨HTTPReq, ν5, POST, url.host, url.path, url.parameters, ⟨⟩, regData⟩
26:         call HTTPS_SIMPLE_SEND([responseTo: REGISTRATION, session: sessionId, sigKey: signingKey, tlsKey: tlsKey],
                ↪ message, a, s')
        → Construct pushed authorization request or CIBA authentication request
27:     if session[cibaFlow] ≡ ⊤ then
28:         let requestEndpoint := oauthConfig[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 ∈ {mTLS_mTLS, mTLS_DPoP} then    → mTLS client authentication
35:         let mtlsNonce such that ⟨requestEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
36:         let authData := [TLS_AuthN: mtlsNonce]
37:         let s'.mtlsCache := s'.mtlsCache −^⟨⟩ ⟨requestEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩
38:     else if clientType ∈ {pkjwt_mTLS, pkjwt_DPoP} then    → private_key_jwt client authentication
39:         let jwt := [iss: clientId, sub: clientId, aud: selectedAS]
40:         let jws := sig(jwt, clientSignKey)
41:         let authData := [client_assertion: jws]
42:     if session[cibaFlow] ≡ ⊤ then
43:         let requestData := [client_id: clientId, scope: openid, login_hint: session[selected_identity],
                ↪ binding_message: session[binding_message]]
44:         if cibaDeliveryMode ≡ ping then
45:             let requestData[client_notification_token] := ν_cibaNotifToken
46:     else
47:         let pkceVerifier := ν_pkce    → Fresh random value
48:         let pkceChallenge := hash(pkceVerifier)
49:         let requestData := [response_type: code, code_challenge_method: S256, client_id: clientId,
                ↪ redirect_uri: redirectUri, code_challenge: pkceChallenge]
50:         let useOidc ← {⊤, ⊥}    → Use of OIDC is optional
51:         if useOidc ≡ ⊤ then
52:             let requestData[scope] := openid
53:         let s'.sessions[sessionId][code_verifier] := pkceVerifier    → Store PKCE randomness in state
    → Algorithm continues on next page.
```

```
54:  let s'.sessions[sessionId] := s'.sessions[sessionId] +⟨⟩ requestData
55:  let requestData := requestData +⟨⟩ authData
56:  if session[cibaFlow] ≡ ⊥ then
57:      let requestSignedResponse ⟵ {⊤, ⊥}    → Choose whether to request a signed authorization response
58:      if requestSignedResponse ≡ ⊤ then
59:          let requestData[response_mode] := jwt    → Request signed authorization response (cf. JARM [68, Sec. 2.3] and [39, Sec.
                                                       5.4.2 No. 1])
       → Note: Following the recommendation in [90, Sec. 3.1.2.1], we do not set a response_mode for "regular" requests.
60:      let signPAR ⟵ {⊤, ⊥}    → Choose whether to use a signed authorization request
61:      if signPAR ≡ ⊤ then
62:          let requestData[aud] := selectedAS    → See [39, Sec. 5.3.2 No. 2]
63:          let body := sig(requestData, clientSignKey)    → Sign authorization request (FAPI 2.0 Message Signing)
64:      else
65:          let body := requestData
66:  else
67:      let body := requestData
68:  let req := ⟨HTTPReq, v_authReqNonce, POST, requestEndpoint.host, requestEndpoint.path, requestEndpoint.parameters, ⟨⟩, body⟩
69:  if session[cibaFlow] ≡ ⊤ then
70:      call HTTPS_SIMPLE_SEND([responseTo: CIBA_AUTH_REQ, session: sessionId], req, a, s')
71:  else
72:      call HTTPS_SIMPLE_SEND([responseTo: PAR, session: sessionId, response_mode: requestData[response_mode]]
                                ↪ , req, a, s')
```

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

```
1:  function PROCESS_TRIGGER(a, s')
2:      let action ← {MTLS_PREPARE_AS, MTLS_PREPARE_RS, MTLS_PREPARE_MISCONFIGURED_TOKEN_EP,
                ↪ 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 ← Doms such that server ∈ s'.asAccounts 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 := ⟨HTTPReq, ν_mtls, GET, server, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
10:             call HTTPS_SIMPLE_SEND([responseTo: MTLS, client_id: clientId], message, a, s')
11:         case MTLS_PREPARE_RS
                → Non-deterministically contact some RS to get an mTLS nonce for mTLS access token sender constraining (for an access
                  token issued by selectedAS, i.e., that token is bound to the mTLS key registered with selectedAS).
12:             let resourceServer ← Doms    → Note: This may or may not be a "real" RS.
13:             let selectedAS ← Doms such that selectedAS ∈ s'.asAccounts if possible; otherwise stop
14:             let mTlsPrivKey := s'.asAccounts[selectedAS][tls_key]
15:             let pubKey := pub(mTlsPrivKey)
16:             let body := [pub_key: pubKey]
17:             let message := ⟨HTTPReq, ν_mtls, GET, resourceServer, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
18:             call HTTPS_SIMPLE_SEND([responseTo: MTLS, pub_key: pubKey], message, a, s')
19:         case MTLS_PREPARE_MISCONFIGURED_TOKEN_EP
                → This case allows the client to retrieve mTLS nonces from attacker-controlled servers and subsequently make requests to
                  such servers. Without this case, the model would not capture attacks in which the client talks to attacker-controlled
                  endpoints protected by mTLS.
20:             let server ← Doms such that server ∈ s'.asAccounts if possible; otherwise stop
21:             let asAcc := s'.asAccounts[server]
22:             let clientId := asAcc[client_id]
23:             let host ← Doms    → Non-deterministically choose the domain instead of sending to the correct AS
24:             let body := [client_id: clientId]
25:             let message := ⟨HTTPReq, ν_mtls, GET, host, /MTLS-prepare, ⟨⟩, ⟨⟩, body⟩
26:             call HTTPS_SIMPLE_SEND([responseTo: MTLS, client_id: clientId], message, a, s')
27:         case GET_DPOP_NONCE
                → Our client uses DPoP server-provided nonces at the RS. The RS model offers a special endpoint to retrieve nonces.
28:             let resourceServer ← Doms    → Note: This may or may not be a "real" RS.
29:             let message := ⟨HTTPReq, ν_DPoPreq, GET, resourceServer, /DPoP-nonce, ⟨⟩, ⟨⟩, ⟨⟩⟩
30:             call HTTPS_SIMPLE_SEND([responseTo: DPOP_NONCE], message, a, s')
31:         case CHANGE_CLIENT_CONFIG    → Use dynamic client management at AS (see RFC 7592 [86])
                → Randomly select one of the ASs this client is registered with.
32:             let selectedAS ← Doms such that selectedAS ∈ s'.asAccounts if possible; otherwise stop
33:             let regClientUri := s'.asAccounts[selectedAS][reg_client_uri]    → Client management URI
34:             let regAt := s'.asAccounts[selectedAS][reg_at]    → Client management access token
35:             let authHeader := [Authorization: ⟨Bearer, regAt⟩]
36:             let DCMaction ← {UPDATE, DELETE}    → Randomly select a client management action
37:             switch DCMaction do
38:                 case DELETE    → Delete client at AS, see Sec. 2.3 of RFC 7592 [86]
39:                     let message := ⟨HTTPReq, ν_DELreq, DELETE, regClientUri.host, regClientUri.path, ⟨⟩, authHeader, ⟨⟩⟩
40:                     call HTTPS_SIMPLE_SEND([responseTo: CLIENT_MANAGEMENT], message, a, s')
41:                 case UPDATE    → Update client configuration at AT, see Sec. 2.2 of RFC 7592 [86]
                        → Generate fresh key pairs and update client keys at AS. Note that following our simplified model of mTLS, the
                          client metadata includes a public key instead of a distinguished name for mTLS (or similar, see Sec. 2.1.2 of
                          RFC 8705 [10]).
42:                     let redirectUris := {⟨URL, S, d, /redirect_ep, ⟨⟩, ⊥⟩ | d ∈ dom(c)}    → Set of redirect URIs for all domains of c
43:                     let newSigningKey := ν_cliSignK
44:                     let newTLSKey := ν_cliTlsK
45:                     let newJwks := ⟨[use: sig, val: pub(newSigningKey)], [use: TLS, val: pub(newTLSKey)]⟩
46:                     let body := [client_id: clientId, jwks: newJwks, redirect_uris: ⟨redirectUris⟩]
    → Algorithm continues on next page.
```

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

Algorithm 10 Relation of *script_client_index*

Input: ⟨*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*⟩ → **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).
```

C.11 Authorization Servers

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

Definition 14. *A* state $s \in Z^{\text{as}}$ of an authorization server as *is a term of the form* $\langle$*DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, jwk, pendingClientIds, clients, records, authorizationRequests, cibaAuthnRequests, mtlsRequests, cibaEndUserEndpoints, rsCredentials*$\rangle$ *with* $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $tlskeys \in [\mathsf{Doms} \times K_{TLS}]$ *(all former components as in Definition 83),* $jwk \in K_{sign}$, $pendingClientIds \in \mathcal{T}_{\mathcal{N}}$, $clients \in \left[\mathcal{T}_{\mathcal{N}} \times \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]\right]$, $records \in \mathcal{T}_{\mathcal{N}}$, $authorizationRequests \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $cibaAuthnRequests \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $mtlsRequests \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $cibaEndUserEndpoints \in \mathcal{T}_{\mathcal{N}}$, *and* $rsCredentials \in \mathcal{T}_{\mathcal{N}}$. *An* initial state s_0^{as} of as *is a state of* as *with*

- $s_0^{\text{as}}.\mathtt{DNSaddress} \in \mathsf{IPs}$,
- $s_0^{\text{as}}.\mathtt{pendingDNS} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{pendingRequests} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{corrupt} \equiv \bot$,
- $s_0^{\text{as}}.\mathtt{keyMapping}$ *being the same as the keymapping for browsers,*
- $s_0^{\text{as}}.\mathtt{tlskeys} \equiv tlskeys^{\text{as}}$ *(see Appendix C.3),*
- $s_0^{\text{as}}.\mathtt{jwk} \equiv \mathsf{signkey}(as)$ *(see Appendix C.3),*
- $s_0^{\text{as}}.\mathtt{pendingClientIds} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{clients} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{records} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{authorizationRequests} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{cibaAuthnRequests} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{mtlsRequests} \equiv \langle\rangle$,
- $s_0^{\text{as}}.\mathtt{cibaEndUserEndpoints} \equiv$ *userEp where userEp is a dictionary and* $\langle identity, ep\rangle \in^{\langle\rangle} userEp \Leftrightarrow (identity.\mathtt{domain} \in \mathsf{dom}(\mathsf{as}) \wedge \mathsf{dom}^{-1}(ep.\mathtt{host}) = \mathsf{ownerOfID}(identity))$*, 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*
- $s_0^{\text{as}}.\mathtt{rsCredentials} \equiv$ *rsCreds where rsCreds is a dictionary and* $\langle rsDom, c\rangle \in^{\langle\rangle} rsCreds \Leftrightarrow (\exists d \in \mathsf{dom}(\mathsf{as}), rsDom \in \mathsf{Doms}\colon c \equiv \mathsf{secretOfRS}(d, rsDom))$*, 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 G.12). 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 ≡ /.well-known/openid-configuration ∨
        ↪ m.path ≡ /.well-known/oauth-authorization-server then   → We model both OIDD, RFC 8414, and FAPI CIBA.
3:          let metaData := [issuer: m.host]
4:          let metaData[auth_ep] := ⟨URL, S, m.host, /auth, ⟨⟩, ⊥⟩
5:          let metaData[token_ep] := ⟨URL, S, m.host, /token, ⟨⟩, ⊥⟩
6:          let metaData[par_ep] := ⟨URL, S, m.host, /par, ⟨⟩, ⊥⟩
7:          let metaData[introspec_ep] := ⟨URL, S, m.host, /introspect, ⟨⟩, ⊥⟩
8:          let metaData[jwks_uri] := ⟨URL, S, m.host, /jwks, ⟨⟩, ⊥⟩
9:          let metaData[reg_ep] := ⟨URL, S, m.host, /reg, ⟨⟩, ⊥⟩
              → No support for push mode, see Section 5.2.2 of FAPI-CIBA [98]
10:         let metaData[backchannel_token_delivery_modes_supported] := ⟨poll, ping⟩
11:         let metaData[backchannel_authentication_endpoint] := ⟨URL, S, m.host, /backchannel-authn, ⟨⟩, ⊥⟩
12:         let metaData[grant_types_supported] := ⟨authorization_code, urn:openid:params:grant-type:ciba⟩
13:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, metaData⟩, k)
14:         stop ⟨⟨f, a, m'⟩⟩, s'
15:     else if m.path ≡ /jwks then
16:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, pub(s'.jwk)⟩, k)
17:         stop ⟨⟨f, a, m'⟩⟩, s'
18:     else if m.path ≡ /reg ∧ m.method ≡ POST then
19:         call REGISTER_CLIENT(m, k, a, f, s')   → See Algorithm 13
20:     else if m.path ≡ /manage ∧ m.method ≡ PUT then   → DCM: update client metadata (see Sec. 2.2 of RFC 7592 [86])
21:         let clientId := m.body[clientId]
22:         if clientId ∉ s'.clients then
23:             stop   → Unknown client
24:         let clientInfo := s'.clients[clientId]
25:         let regAT := m.headers[Authorization][Bearer]
26:         if regAT ≢ clientInfo[reg_at] then
27:             stop   → Wrong registration access token
28:         let redirectUris := m.body[redirect_uris]
29:         let jwks := m.body[jwks]   → Contains public keys of client
30:         let pubSigKey such that [use: sig, val: pubSigKey] ∈^⟨⟩ jwks if possible; otherwise stop
31:         let mtlsPubKey such that [use: TLS, val: mtlsPubKey] ∈^⟨⟩ jwks if possible; otherwise stop
32:         let regUri := ⟨URL, S, m.host, /manage, ⟨⟩, ⊥⟩
33:         let clientType ← {mTLS_mTLS, mTLS_DPoP, pkjwt_mTLS, pkjwt_DPoP}   → 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] := redirectUris
38:         let regResponse := [client_id: clientId, jwks: jwks, client_type: clientType, reg_at: regAT, reg_client_uri: regUri]
39:         let tokenDeliveryMode ← {poll, ping}   → Non-deterministic choice of CIBA token delivery mode
40:         let grantTypes := ⟨authorization_code, urn:openid:params:grant-type:ciba⟩   → 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 ≡ ping then
46:             if backchannel_client_notification_endpoint ∉ 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(⟨HTTPResp, m.nonce, 200, ⟨⟩, regResponse⟩, k)
53:         stop ⟨⟨f, a, m'⟩⟩, s'
54:     else if m.path ≡ /manage ∧ m.method ≡ DELETE then   → DCM: delete client (see Sec. 2.3 of RFC 7592 [86])
55:         let regAT := m.headers[Authorization][Bearer]
56:         let clientId such that s'.clients[clientId][reg_at] ≡ regAT if possible; otherwise stop
57:         let s'.clients[clientId][active] := ⊥   → Deactivate client account
58:         stop ⟨⟩, s'
```

→ Algorithm continues on next page.

```
59:     else if m.path ≡ /auth then    → Authorization endpoint: Reply with login page.
60:         if m.method ≡ GET then
61:             let data := m.parameters
62:         else if m.method ≡ POST then
63:             let data := m.body
64:         let requestUri := data[request_uri]
65:         if requestUri ≡ ⟨⟩ then
66:             stop    → 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] ≢ clientId then    → Check binding of request URI to client
70:             stop
71:         if clientId ∉ s'.clients ∨ s'.clients[clientId][active] ≢ ⊤ then
72:             stop    → Unknown client
73:         let s'.authorizationRequests[requestUri][auth2_reference] := ν5
74:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟨RefererPolicy, origin⟩⟩, ⟨script_as_form, [auth2_reference: ν5]⟩⟩, k)
75:         stop ⟨⟨f, a, m'⟩⟩, s'
76:     else if m.path ≡ /auth2 ∧ m.method ≡ POST ∧ m.headers[Origin] ≡ ⟨m.host, S⟩ then    → Second step of authorization
77:         let identity := m.body[identity]
78:         let password := m.body[password]
79:         if identity.domain ∉ dom(as) then
80:             stop    → This AS does not manage identity
81:         if password ≢ secretOfID(identity) then
82:             stop    → Invalid user credentials
83:         let auth2Reference := m.body[auth2_reference]
84:         let requestUri such that s'.authorizationRequests[requestUri][auth2_reference] ≡ auth2Reference
                ↪ if possible; otherwise stop
85:         let authzRecord := s'.authorizationRequests[requestUri]
86:         let authzRecord[subject] := identity
87:         let authzRecord[issuer] := m.host
88:         let authzRecord[code] := ν1    → Generate a fresh, random authorization code
89:         let s'.records := s'.records +^⟨⟩ authzRecord
90:         let redirectUri := authzRecord[redirect_uri]
91:         let responseData := [code: authzRecord[code]]
92:         if authzRecord[state] ≢ ⟨⟩ then
93:             let responseData[state] := authzRecord[state]
94:         if authzRecord[sign_authz_response] ≡ ⊤ then
95:             let responseData[iss] := authzRecord[issuer]    → iss claim is part of JWT instead of a parameter, see [39, Sec. 5.4.1]
96:             let responseData[aud] := clientId    → See JARM [68, Sec. 2.1]
97:             let responseData := [response: sig(responseData, s'.jwk)]    → Sign authorization response using JARM [68, Sec. 2.3.1]
98:         else
99:             let redirectUri.parameters[iss] := authzRecord[issuer]    → Overwrite iss parameter if present in redirectUri
100:        let redirectUri.parameters := redirectUri.parameters ∪ responseData
101:        let m' := enc_s(⟨HTTPResp, m.nonce, 303, ⟨⟨Location, redirectUri⟩⟩, ⟨⟩⟩, k)
102:        stop ⟨⟨f, a, m'⟩⟩, s'
103:    else if m.path ≡ /par ∧ m.method ≡ POST then    → Pushed Authorization Request
104:        let requireSignedPAR ← {⊤, ⊥}    → Choose whether to require a signed PAR
        → Note: If the client signed the PAR, but the AS chooses not to require a signature, client authentication below will fail.
105:        if requireSignedPAR ≡ ⊤ then
106:            let mBody := extractmsg(m.body)    → Note: If m.body ≁ sig(∗, ∗) (or mac(∗, ∗)), then there is no processing step
107:            if checksig(m.body, s'.clients[mBody[client_id]][jwt_key]) ≢ ⊤ then
108:                stop    → Invalid signature
109:            if mBody[aud] ≢ m.host then
110:                stop    → Wrong audience value in JWS, see [39, Sec. 5.3.1 No. 2]
111:        else
112:            let mBody := m.body
113:        let m.body := mBody    → In case of a signed PAR: Strip off the signature after verifying it
114:        if m.body[response_type] ≢ code ∨ m.body[code_challenge_method] ≢ S256 then
115:            stop
116:        let authnResult := AUTHENTICATE_CLIENT(m, s')    → Stops in case of errors/failed authentication
117:        let clientId := authnResult.1
118:        let s' := authnResult.2
119:        let mtlsInfo := authnResult.3
120:        if clientId ≢ mBody[client_id] then
121:            stop    → Key used in client authentication is not registered for m.body[client_id]
```

→ 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 := v4    → 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′ := enc_s(⟨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 mtlsInfo := 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
                    ↪ ∧code ≢ ⊥ ∧ ptr ∈ ℕ if possible; otherwise stop
158:             if record[code_challenge] ≢ hash(codeVerifier) ∨ record[redirect_uri] ≢ m.body[redirect_uri] then
159:                 stop    → PKCE verification failed or URI mismatch
160:         else if m.body[grant_type] ≡ urn:openid:params:grant-type:ciba then
161:             let authReqId := m.body[auth_req_id]
162:             if authReqId ≡ ⟨⟩ then
163:                 stop    → Missing auth_req_id
164:             let record, ptr such that record ≡ s′.records.ptr ∧ record[auth_req_id] ≡ auth_req_id
                    ↪ ∧auth_req_id ≢ ⊥ ∧ ptr ∈ ℕ if possible; otherwise stop
165:         if record[client_id] ≢ clientId then
166:             stop
167:         let clientType := s′.clients[clientId][client_type]
168:         if clientType ≡ pkjwt_DPoP ∨ clientType ≡ mTLS_DPoP then    → DPoP token binding
169:             let tokenType := DPoP
170:             let dpopProof := m.headers[DPoP]
171:             let dpopJwt := extractmsg(dpopProof)
172:             let verificationKey := dpopJwt[headers][jwk]
173:             if checksig(dpopProof, verificationKey) ≢ ⊤ ∨ verificationKey ≡ ⟨⟩ then
174:                 stop    → Invalid DPoP signature (or empty jwk header)
175:             let dpopClaims := dpopJwt[payload]
176:             let reqUri := ⟨URL, S, m.host, m.path, ⟨⟩, ⊥⟩
177:             if dpopClaims[htm] ≢ m.method ∨ dpopClaims[htu] ≢ reqUri then
178:                 stop    → DPoP claims do not match corresponding message
179:             let cnfContent := [jkt: hash(verificationKey)]
180:         else if clientType ≡ pkjwt_mTLS ∨ clientType ≡ mTLS_mTLS then    → mTLS token binding
181:             let tokenType := Bearer
182:             let mtlsNonce := m.body[TLS_binding]
```

→ Algorithm continues on next page.

```
183:            if clientType ≡ mTLS_mTLS then    → Client used mTLS authentication, reuse data from authentication
184:                if mtlsNonce ≢ mtlsInfo.1 then
185:                    stop    → Client tried to use different mTLS nonce for authentication and token binding
186:            else    → Client did not use mTLS authentication
187:                let mtlsInfo such that mtlsInfo ∈ s'.mtlsRequests[clientId] ∧ mtlsInfo.1 ≡ mtlsNonce if possible; otherwise stop
188:                let s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] −^⟨⟩ mtlsInfo
189:            let mTlsKey := mtlsInfo.2    → mTLS public key of client
190:            let cnfContent := [x5t#S256: hash(mTlsKey)]
191:        else
192:            stop    → Client used neither DPoP nor mTLS
193:        if m.body[grant_type] ≡ authorization_code then
194:            let s'.records.ptr[code] := ⊥    → Invalidate code
195:        else
196:            let s'.records.ptr[auth_req_id] := ⊥    → Invalidate request id
197:        let atType ← {JWT, opaque}    → The AS chooses randomly whether it issues a structured or an opaque access token
198:        if atType ≡ JWT then    → 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    → Opaque access token
202:            let accessToken := v_2    → Fresh random value
203:        let s'.records.ptr[access_token] := accessToken    → Store for token introspection
204:        let s'.records.ptr[cnf] := cnfContent    → Store for token introspection
205:        let body := [access_token: accessToken, token_type: tokenType]
206:        if record[scope] ≡ openid then    → 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 ∈ 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(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
215:        stop ⟨⟨f, a, m'⟩⟩, s'
216:    else if m.path ≡ /introspect ∧ m.method ≡ POST ∧ token ∈ m.body then
217:        let rsSecret such that ⟨Basic, rsSecret⟩ ≡ m.headers[Authorization] if possible; otherwise stop
218:        let rsDom such that s'.rsCredentials[rsDom] ≡ rsSecret if possible; otherwise stop    → RS authentication at AS
219:        let token := m.body[token]
220:        let record such that record ∈ s'.records ∧ record[access_token] ≡ token if possible; otherwise let record := ◇
221:        if record ≡ ◇ ∨ s'.clients[record[client_id]][active] ≢ ⊤ then    → Unknown token or deactivated client
222:            let body := [active: ⊥]
223:        else    → token was issued by this AS & client is active
224:            let clientId := record[client_id]
                → cnf claim contains hash of token binding key, the signing key is the key used by the client to sign HTTP messages
225:            let body := [active: ⊤, cnf: record[cnf], sub: record[subject], client_sig_key: s'.clients[clientId][jwt_key]]
226:        if m.headers[Accept] ≡ app/token-introspection+jwt then    → Check whether RS requested a signed response
227:            let body := sig([token_introspection: body, iss: m.host, aud: rsDom], s'.jwk)
228:        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
229:        stop ⟨⟨f, a, m'⟩⟩, s'
230:    else if m.path ≡ /MTLS-prepare then    → See Appendix C.7
231:        let clientId := m.body[client_id]
232:        if s'.clients[clientId][active] ≢ ⊤ then
233:            stop
234:        let mtlsNonce := v_3
235:        let clientKey := s'.clients[clientId][mtls_key]
236:        if clientKey ≡ ⟨⟩ ∨ clientKey ≡ pub(◇) then
237:            stop    → Client has no mTLS key
238:        let s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] +^⟨⟩ ⟨mtlsNonce, clientKey⟩
239:        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, enc_a(⟨mtlsNonce, s'.keyMapping[m.host]⟩, clientKey)⟩, k)
240:        stop ⟨⟨f, a, m'⟩⟩, s'
241:    else if m.path ≡ /backchannel-authn ∧ m.method ≡ POST then    → CIBA Authentication Request
242:        let authnResult := AUTHENTICATE_CLIENT(m, s')    → Stops in case of errors/failed authentication
243:        let clientId := authnResult.1
244:        let s' := authnResult.2
245:        let mtlsInfo := authnResult.3
```

→ Algorithm continues on next page.

246: **if** *clientId* ≢ *m*.body[client_id] **then**
247: **stop** → Key used in client authentication is not registered for *m*.body[client_id]
248: **if** openid ∉^⟨⟩ *m*.body[scope] **then**
249: **stop**
250: **if** urn:openid:params:grant-type:ciba ∉^⟨⟩ *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* ≡ ping **then**
258: **let** *authzRecord*[client_notification_token] := *m*.body[client_notification_token]
259: **if** *m*.body[selected_identity].domain ∉ dom(as) **then**
260: **stop** → This AS does not manage the requested identity
261: **let** *authzRecord*[authenticateUser] := ⊤ → Flag indicating whether the AS needs to obtain end-user consent/authorization
262: **let** *authnReqId* := $v_{\text{authn\_req\_id}}$
263: **let** *s′*.cibaAuthnRequests[*authnReqId*] := *authzRecord* → Store data linked to *authnReqId*
264: **let** *body* := [auth_req_id: *authnReqId*]
265: **let** *m′* := $\text{enc}_s(\langle \text{HTTPResp}, m.\text{nonce}, 200, \langle\rangle, body\rangle, k)$
266: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
267: **else if** *m*.path ≡ /ciba-auth **then** → Authorization endpoint for CIBA Flows: Reply with login page, include binding message.
268: **if** *m*.method ≢ POST **then**
269: **stop**
270: **if** ciba_user_nonce ∉ *m*.body **then**
271: **stop**
272: **let** *cibaUserNonce* := *m*.body[ciba_user_nonce]
273: **let** *authnReqId* **such that** *authnReqId* ∈ *s′*.cibaAuthnRequests ↪ ∧ *s′*.cibaAuthnRequests[*authnReqId*][cibaUserAuthNNonce] ≡ *cibaUserNonce* **if possible; otherwise stop**
274: **let** *bindingMessage* := *s′*.cibaAuthnRequests[*authnReqId*][binding_message]
275: **let** *clientId* := *s′*.cibaAuthnRequests[*authnReqId*][client_id]
276: **let** *clientDom* ← *s′*.clients[*clientId*][redirect_uris]
277: **let** *s′*.cibaAuthnRequests[*authnReqId*][ciba_auth2_reference] := $v_{\text{ciba\_auth2\_ref}}$
278: **let** *body* := ⟨script_as_ciba_form, [ciba_auth2_reference: $v_{\text{ciba\_auth2\_ref}}$, binding_message: *bindingMessage* ↪ client_domain: *clientDom*.host]⟩
279: **let** *m′* := $\text{enc}_s(\langle \text{HTTPResp}, m.\text{nonce}, 200, \langle\langle \text{ReferrerPolicy}, \text{origin}\rangle\rangle, body\rangle, k)$
280: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
281: **else if** *m*.path ≡ /ciba-auth2 ∧ *m*.method ≡ POST ∧ *m*.headers[Origin] ≡ ⟨*m*.host, S⟩ **then** → Finish authorization (CIBA)
282: **let** *identity* := *m*.body[identity]
283: **let** *password* := *m*.body[password]
284: **if** *identity*.domain ∉ dom(as) **then**
285: **stop** → This AS does not manage *identity*
286: **if** *password* ≢ 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] ≡ *auth2Reference* ↪ **if possible; otherwise stop**
290: **if** *identity* ≢ *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] := ⊥ → 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 +^⟨⟩ *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] ≡ ping **then**
300: **let** *clientURL* := *s′*.clients[*clientId*][backchannel_client_notification_endpoint]
301: **let** *body* := [auth_req_id: *authnReqId*]
302: **let** *headers* := [Authorization: ⟨Bearer, *authzRecord*[client_notification_token]⟩]
303: **let** *message* := ⟨HTTPReq, $v_{\text{ciba\_ping}}$, POST, *clientURL*.host, *clientURL*.path, ⟨⟩, *headers*, *body*⟩
304: **call** HTTPS_SIMPLE_SEND([responseTo: CIBAPingCallback], *message*, *a*, *s′*)
305: **else**
306: **stop** ⟨⟩, *s′*
307: **stop** → Request was malformed or sent to non-existing endpoint.

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

function AUTHENTICATE_CLIENT(m, s') → Check client authentication in message m. Stops the current processing step in case of errors or failed authentication.
  if `client_assertion` $\in m$.body **then** → `private_key_jwt` client authentication
    let *jwts* := m.body[`client_assertion`]
    let *clientId*, *verificationKey* **such that** *verificationKey* $\equiv s'$.clients[*clientId*][`jwt_key`] $\wedge$ checksig(*jwts*, *verificationKey*) $\equiv \top$ **if possible; otherwise stop**
    if *verificationKey* $\equiv \langle\rangle \vee$ *verificationKey* $\equiv$ pub($\diamond$) **then**
      stop → Client has no jwt key
    let *clientInfo* := s'.clients[*clientId*]
    let *clientType* := *clientInfo*[`client_type`]
    if *clientType* $\not\equiv$ `pkjwt_mTLS` $\wedge$ *clientType* $\not\equiv$ `pkjwt_DPoP` **then**
      stop → Client authentication type mismatch
    let *jwt* := extractmsg(*jwts*)
    if *jwt*[`iss`] $\not\equiv$ *clientId* $\vee$ *jwt*[`sub`] $\not\equiv$ *clientId* **then**
      stop
    if *jwt*[`aud`] $\not\equiv \langle$`URL, S`, m.`host`, /`token`, $\langle\rangle, \bot\rangle \wedge$ *jwt*[`aud`] $\not\equiv m$.`host` → `issuer` in AS metadata is just the host part $\wedge$ *jwt*[`aud`] $\not\equiv \langle$`URL, S`, m.`host`, /`par`, $\langle\rangle, \bot\rangle$ **then**
      stop → aud claim value is neither token, nor PAR endpoint nor AS issuer identifier
  else if `TLS_AuthN` $\in m$.body **then** → mTLS client authentication
    let *clientId* := m.body[`client_id`] → RFC 8705 [10] mandates `client_id` when using mTLS authentication
    let *mtlsNonce* := m.body[`TLS_AuthN`]
    let *mtlsInfo* **such that** *mtlsInfo* $\in s'$.mtlsRequests[*clientId*] $\wedge$ *mtlsInfo*.1 $\equiv$ *mtlsNonce* **if possible; otherwise stop**
    let *clientInfo* := s'.clients[*clientId*]
    let *clientType* := *clientInfo*[`client_type`]
    if *clientType* $\not\equiv$ `mTLS_mTLS` $\wedge$ *clientType* $\not\equiv$ `mTLS_DPoP` **then**
      stop → Client authentication type mismatch
    let s'.mtlsRequests[*clientId*] := s'.mtlsRequests[*clientId*] $-^{\langle\rangle}$ *mtlsInfo*
  else
    stop → Unsupported client (authentication) type
  if s'.clients[*clientId*][`active`] $\not\equiv \top$ **then**
    stop
  if *clientType* $\equiv$ `mTLS_mTLS` $\vee$ *clientType* $\equiv$ `mTLS_DPoP` **then**
    return $\langle$*clientId*, s', *mtlsInfo*$\rangle$
  else
    return $\langle$*clientId*, s', $\bot\rangle$ → `private_key_jwt` client authentication, i.e., no mTLS info

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

1: **function** REGISTER_CLIENT(m, k, a, f, s') → m is the decrypted HTTP request containing a client registration request
2: **let** *clientId* ← s'.pendingClientIds → Client ids are provided by the attacker (see also Algorithm 14).
3: **let** s'.pendingClientIds := s'.pendingClientIds $-^{\langle\rangle}$ *clientId*
→ Construct client information response (see Sec. 2 of RFC 7592 [86] and Sec. 3.2.1 of RFC 7591 [85])
4: **let** *redirectUris* := m.body[redirect_uris]
5: **let** *jwks* := m.body[jwks] → Contains public keys of client
6: **let** *pubSigKey* **such that** [use: sig, val: *pubSigKey*] $\in^{\langle\rangle}$ *jwks* **if possible; otherwise stop**
7: **let** *mtlsPubKey* **such that** [use: TLS, val: *mtlsPubKey*] $\in^{\langle\rangle}$ *jwks* **if possible; otherwise stop**
8: **let** *clientType* ← {mTLS_mTLS, mTLS_DPoP, pkjwt_mTLS, pkjwt_DPoP} → Non-deterministic choice of client type
9: **let** *regAT* := v_{regAT} → Registration access token (cf. Sec. 3 of RFC 7592 [86])
10: **let** *regUri* := ⟨URL, S, m.host, /manage, ⟨⟩, ⊥⟩ → Registration client uri (cf. Sec. 3 of RFC 7592 [86])
11: **let** *tokenDeliveryMode* ← {poll, ping} → Non-deterministic choice of CIBA token delivery mode
12: **let** *grantTypes* := ⟨authorization_code, urn:openid:params:grant-type:ciba⟩ → In the model, the AS always registers both types
13: **let** *regResponse* := [client_id: *clientId*, jwks: *jwks*, client_type: *clientType*, reg_at: *regAT*, reg_client_uri: *regUri*, ↪ grant_types: *grantTypes*]
14: **let** *clientInfo* := [client_type: *clientType*, redirect_uris: *redirectUris*, jwt_key: *pubSigKey*, ↪ mtls_key: *mtlsPubKey*, reg_at: *regAT*, grant_types: *grantTypes*]
15: **let** *clientInfo*[active] := ⊤ → 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* ≡ ping **then**
19: **if** backchannel_client_notification_endpoint ∉ 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(⟨HTTPResp, m.nonce, 201, ⟨⟩, *regResponse*⟩, k)
26: **stop** ⟨⟨f, a, m'⟩⟩, s'

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

1: **function** PROCESS_OTHER(m, a, f, s')
2: **let** *clientId* := m → m is a client id chosen by and sent by an attacker process (see also Line 2 of Algorithm 13)
3: **if** *clientId* ∈ s'.clients ∨ *clientId* ∈ s'.pendingClientIds **then**
4: **stop**
5: **let** s'.pendingClientIds := s'.pendingClientIds $+^{\langle\rangle}$ *clientId*
6: **stop** ⟨⟩, s'

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

1: **function** PROCESS_TRIGGER(a, s')
2: **let** *action* ← {CIBA_OBTAIN_CONSENT}
3: **switch** *action* **do**
4: **case** CIBA_OBTAIN_CONSENT
→ Choose one of the CIBA authentication requests for which the AS did not ask the end-user yet
5: **let** *authnReqId* **such that** *authnReqId* ∈ s'.cibaAuthnRequests ↪ ∧s'.cibaAuthnRequests[*authnReqId*][authenticateUser] ≡ ⊤ **if possible; otherwise stop**
6: **let** *selectedUser* := s'.cibaAuthnRequests[*authnReqId*][selected_identity]
→ Get the endpoint of the end-user
7: **let** *userEp* := s'.cibaEndUserEndpoints[*selectedUser*]
8: **let** *cibaUserAuthNNonce* := $v_{\text{cibaUserNonce}}$ → 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* := {⟨URL, S, d, /ciba_auth, ⟨⟩, ⊥⟩ | d ∈ dom(as)} → 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* := ⟨HTTPReq, v_{ciba}, GET, *userEp*.host, /start-ciba-authentication, ⟨⟩, ⟨⟩, *body*⟩
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: ⟨*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*⟩
1: **let** *url* := GETURL(*tree*, *docnonce*)
2: **let** *url*′ := ⟨URL, S, *url*.host, /auth2, ⟨⟩, ⊥⟩
3: **let** *formData* := *scriptstate*
4: **let** *identity* ← *ids*
5: **let** *secret* ← *secrets*
6: **let** *formData*[identity] := *identity*
7: **let** *formData*[password] := *secret*
8: **let** *command* := ⟨FORM, *url*′, POST, *formData*, ⊥⟩
9: **stop** ⟨*s*, *cookies*, *localStorage*, *sessionStorage*, *command*⟩

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

Input: ⟨*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*⟩
1: **let** *url* := GETURL(*tree*, *docnonce*)
2: **let** *url*′ := ⟨URL, S, *url*.host, /ciba-auth2, ⟨⟩, ⊥⟩
3: **let** *formData* := [ciba_auth2_reference: *scriptstate*[ciba_auth2_reference]]
4: **let** *bindingMessage* := *scriptstate*[binding_message]
5: **let** *clientDomain* := *scriptstate*[client_domain]
6: **let** *identity* ← *ids*
7: **let** *secret* ← *secrets*
8: **let** *formData*[identity] := *identity*
9: **let** *formData*[password] := *secret*
10: **let** *command* := ⟨CIBAFORM, *url*′, POST, *formData*, ⊥, *clientDomain*, *bindingMessage*⟩
11: **stop** ⟨*s*, *cookies*, *localStorage*, *sessionStorage*, *command*⟩

C.12 Resource Servers

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

Definition 15. *A* state $s \in Z^{rs}$ of a resource server rs *is a term of the form* $\langle$*DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, mtlsRequests, pendingResponses, resourceNonces, ids, asInfo, resourceASMapping, dpopNonces, jwk*$\rangle$ *with* $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{TLS}]$ *(all former components as in Definition 83),* $\mathit{mtlsRequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{pendingResponses} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{resourceNonces} \in \left[\mathsf{ID} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{ids} \subset^{\langle\rangle} \mathsf{ID}$, $\mathit{asInfo} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{resourceASMapping} \in \left[\mathsf{resourceURLPath}^{rs} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{dpopNonces} \in \mathcal{T}_{\mathcal{N}}$, *and* $\mathit{jwk} \in K_{sign}$.
An initial state s_0^{rs} of rs *is a state of* rs *with*

- $s_0^{rs}.\mathtt{DNSaddress} \in \mathsf{IPs}$,
- $s_0^{rs}.\mathtt{pendingDNS} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{pendingRequests} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{corrupt} \equiv \bot$,
- $s_0^{rs}.\mathtt{keyMapping}$ *being the same as the keymapping for browsers,*
- $s_0^{rs}.\mathtt{tlskeys} \equiv \mathit{tlskeys}^{rs}$ *(see Appendix C.3),*
- $s_0^{rs}.\mathtt{mtlsRequests} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{pendingResponses} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{resourceNonces}$ *being a dictionary where the RS stores the resource nonces for each identity and resource id pair, initialized as* $s_0^{rs}.\mathtt{resourceNonces}[id][\mathit{resourceID}] := \langle\rangle$, $\forall id \in^{\langle\rangle} s_0^{rs}.\mathtt{ids}, \forall \mathit{resourceID} \in \mathbb{S}$,
- $s_0^{rs}.\mathtt{ids} \subset^{\langle\rangle} \mathsf{ID}$ *such that* $\forall id \in s_0^{rs}.\mathtt{ids} : \mathsf{governor}(id) \in \mathsf{supportedAuthorizationServer}^{rs}$*, i.e., the RS manages only resources of identities that are governed by one of the AS supported by the RS,*
- $s_0^{rs}.\mathtt{asInfo}$*: for each domain of a supported AS* $dom_{as} \in \mathsf{supportedAuthorizationSeverDoms}^{rs}$*, let* $s_0^{rs}.\mathtt{asInfo}$ *contain a dictionary entry with the following values:*
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{as\_introspect\_ep}] \equiv \langle \mathsf{URL}, \mathsf{S}, dom_{as}, /\mathtt{introspect}, \langle\rangle, \bot\rangle$ *(the URL of the introspection endpoint of the AS)*
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{as\_key}] \equiv \mathsf{pub}(\mathsf{signkey}(\mathsf{dom}^{-1}(dom_{as})))$ *being the verification key for the AS*
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{rs\_credentials}]$ *being a sequence s.t.* $\forall c : c \in^{\langle\rangle} s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{rs\_credentials}] \Leftrightarrow (\exists \mathit{rsDom} \in \mathsf{dom}(rs) : c \equiv \mathsf{secretOfRS}(dom_{as}, \mathit{rsDom}))$*, i.e., the secrets used by the RS for authenticating at the AS*

  Hence, setting up the ASs supported by this RS,
- $s_0^{rs}.\mathtt{resourceASMapping} \in \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(\mathit{resourceID}))$ $\forall \mathit{resourceID} \in \mathsf{resourceURLPath}^{rs}$ *(a domain of the AS managing the resource identified by resourceID),*
- $s_0^{rs}.\mathtt{dpopNonces} \equiv \langle\rangle$*, and*
- $s_0^{rs}.\mathtt{jwk} \equiv \mathsf{signkey}(rs)$ *(used for HTTP message signing, see Appendix C.3).*

The relation R^{rs} is again based on the generic HTTPS server model (see Appendix G.12), 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 ≡ /MTLS-prepare then
3:          let mtlsNonce := v_1
4:          let clientKey := m.body[pub_key]    → Certificate is not required to be checked [10, Section 4.2]
5:          let s'.mtlsRequests := s'.mtlsRequests +^⟨⟩ ⟨mtlsNonce, clientKey⟩
6:          let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, enc_a(⟨mtlsNonce, s'.keyMapping[m.host]⟩, clientKey)⟩, k)
7:          stop ⟨⟨f, a, m'⟩⟩, s'
8:      else if m.path ≡ /DPoP-nonce then
9:          let freshDpopNonce := v_dpop
10:         let s'.dpopNonces := s'.dpopNonces +^⟨⟩ freshDpopNonce
11:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, [nonce: freshDpopNonce]⟩, k)
12:         stop ⟨⟨f, a, m'⟩⟩, s'
13:     else
14:         let expectSignedRequest ← {⊤, ⊥}    → Decide whether to expect a signed resource request.
15:         let resourceID := m.path
16:         let responsibleAS := s'.resourceASMapping[resourceID]
17:         if responsibleAS ≡ ⟨⟩ then
18:             stop    → Resource is not managed by any of the supported ASs
19:         let asInfo := s'.asInfo[responsibleAS]
20:         if Authorization ∉ 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 ≡ Bearer then    → mTLS sender constraining
25:             let mtlsNonce := m.body[TLS_binding]
26:             let mtlsInfo such that mtlsInfo ∈^⟨⟩ s'.mtlsRequests ∧ mtlsInfo.1 ≡ mtlsNonce if possible; otherwise stop
27:             let s'.mtlsRequests := s'.mtlsRequests −^⟨⟩ mtlsInfo
28:             let mtlsKey := mtlsInfo.2
29:             let cnfValue := [x5t#S256: hash(mTlsKey)]
30:         else if authnScheme ≡ 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) ≢ ⊤ ∨ verificationKey ≡ ⟨⟩ then
35:                 stop    → Invalid DPoP signature (or empty jwk header)
36:             let dpopClaims := dpopJwt[payload]
37:             let reqUri := ⟨URL, S, m.host, m.path, ⟨⟩, ⊥⟩
38:             if dpopClaims[htm] ≢ m.method ∨ dpopClaims[htu] ≢ reqUri then
39:                 stop    → DPoP claims do not match corresponding message
40:             if dpopClaims[nonce] ∉ s'.dpopNonces then
41:                 stop    → Invalid DPoP nonce
42:             if dpopClaims[ath] ≢ hash(accessToken) then
43:                 stop    → Invalid access token hash
44:             let s'.dpopNonces := s'.dpopNonces −^⟨⟩ dpopClaims[nonce]
45:             let cnfValue := [jkt: hash(verificationKey)]
46:         else
47:             stop    → Wrong Authorization header value
```

→ Algorithm continues on next page.

```
48:     let resource := ν4    → Generate a fresh resource nonce
49:     let accessTokenContent such that accessTokenContent ≡ extractmsg(accessToken)
            ↪ if possible; otherwise let accessTokenContent := ⋄
50:     if accessTokenContent ≡ ⋄ then    → Not a structured AT, do Token Introspection
51:         let requestSignedIntrospecResponse ← {⊤, ⊥}    → Whether to request a signed introspection response
            → Store values for the pending request (needed when the RS gets the introspection response)
52:         let requestId := ν2
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 ← asInfo[rs_credentials]    → Secret for authenticating at the AS (see also Sec. 2.1 of RFC 7662 [83])
56:         let headers := [Authorization: ⟨Basic, rsCred⟩]
57:         if requestSignedIntrospecResponse ≡ ⊤ then
58:             let headers[Accept] := app/token-introspection+jwt    → Request signed introspection response [72, Sec. 4]
59:         let body := [token: accessToken]
60:         let message := ⟨HTTPReq, ν3, POST, url.domain, url.path, url.parameters, headers, body⟩
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 ≢ accessTokenContent[cnf].1 ∨ cnfValue.2 ≢ accessTokenContent[cnf].2 then
63:         stop    → AT is bound to a different key
64:     if checksig(accessToken, asInfo[as_key]) ≢ ⊤ then
65:         stop    → Verification of AT signature failed
66:     if expectSignedRequest ≡ ⊤ then
67:         let verificationKey := accessTokenContent[client_sig_key]    → AS includes the client's HTTP Message Signing key
                                                                            in structured AT.
68:         let hasValidSignature := VERIFY_REQUEST_SIGNATURE(m, verificationKey)
69:         if hasValidSignature ≢ ⊤ then
70:             stop
71:     let id := accessTokenContent[sub]
72:     if id ∉^⟨⟩ s′.ids 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] +^⟨⟩ resource
75:     let body := [resource: resource]    → This will be the resource response message body
76:     let signResResponse ← {⊤, ⊥}    → Whether to sign the resource response
77:     if signResResponse ≡ ⊤ then
78:         let headers := SIGN_RESOURCE_RESPONSE(body, s′)
79:     else
80:         let headers := ⟨⟩
81:     let m′ := enc_s(⟨HTTPResp, m.nonce, 200, headers, body⟩, k)
        → Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
82:     let leakingMessage := ⟨HTTPReq, ν_RRleak, POST, m.domain, m.path, m.parameters, m.headers, []⟩
83:     let leakAddress ← IPs
84:     stop ⟨⟨f, a, m′⟩, ⟨leakAddress, a, ⟨LEAK, leakingMessage⟩⟩⟩, s′
```

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

function PROCESS_HTTPS_RESPONSE(m, *reference*, *request*, *key*, a, f, s')
  if *reference*[responseTo] ≡ TOKENINTROSPECTION **then**
    let *pendingRequestInfo* := s'.pendingResponses[*reference*[requestId]]
    let s'.pendingResponses := s'.pendingResponses − *reference*[requestId]
    let *clientAddress* := *pendingRequestInfo*[requestingClient]
    let *expectedCNF* := *pendingRequestInfo*[expectedCNF]
    let *origReq* := *pendingRequestInfo*[originalRequest]
    let *originalRequestKey* := *pendingRequestInfo*[originalRequestKey]
    let *resourceID* := *pendingRequestInfo*[resourceID]
    let *resource* := *pendingRequestInfo*[resource]
    let *responsibleAS* := s'.resourceASMapping[*resourceID*]
    if *responsibleAS* ≡ ⟨⟩ **then**
      stop → Resource is not managed by any of the supported ASs
    let *asInfo* := s'.asInfo[*responsibleAS*]
    if *pendingRequestInfo*[requestSignedIntrospecResponse] ≡ ⊤ **then**
      if checksig(m.body, *asInfo*[as_key]) ≢ ⊤ **then**
        stop
      let *response* := extractmsg(m.body)
      if *response*[iss] ≢ *responsibleAS* ∨ *response*[aud] ≢ m.host ∨ token_introspection ∉ *response* **then**
        stop
      let m.body := *response*[token_introspection] → Remove signature for uniform handling of m below
    if m.body[active] ≢ ⊤ **then**
      stop → Access token was invalid
    let *responseCNF* := m.body[cnf]
    if *responseCNF*.1 ≢ *expectedCNF*.1 ∨ *responseCNF*.2 ≢ *expectedCNF*.2 **then**
      stop → Access token was bound to a different key
    let *id* := m.body[sub]
    if *id* $\notin^{\langle\rangle}$ s'.ids **then**
      stop → RS does not manage resources of this RO
    → Handle signed resource requests (i.e., HTTP Message Signatures)
    let *expectSignedRequest* := *reference*[expectSignedRequest]
    if *expectSignedRequest* ≡ ⊤ **then**
      let *verificationKey* := m.body[client_sig_key] → AS includes the client's HTTP Message Signing key in introspection response.
      → Now that *rs* knows the client's HTTP Message Signing key, it can verify the signature on the resource request.
      let *hasValidSignature* := VERIFY_REQUEST_SIGNATURE(*origReq*, *verificationKey*)
      if *hasValidSignature* ≢ ⊤ **then**
        stop
    → Token binding etc. successfully checked, the RS now gives access to a resource of the identity
    let s'.resourceNonces[*id*][*resourceID*] := s'.resourceNonces[*id*][*resourceID*] $+^{\langle\rangle}$ *resource*
    let *body* := [resource: *resource*] → This will be the resource response message body
    if *reference*[signResResponse] ≡ ⊤ **then**
      let *headers* := SIGN_RESOURCE_RESPONSE(*body*, s')
    else
      let *headers* := ⟨⟩
    let m' := enc_s(⟨HTTPResp, *origReq*.nonce, 200, *headers*, *body*⟩, *originalRequestKey*)
    → Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
    let *leakingMessage* := ⟨HTTPReq, ν_{RRleak}, POST, *origReq*.domain, *origReq*.path, *origReq*.parameters, *origReq*.headers, []⟩
    let *leakAddress* ← IPs
    stop ⟨⟨f, a, m'⟩, ⟨*leakAddress*, a, ⟨LEAK, *leakingMessage*⟩⟩⟩, s'
  stop → 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 [39, Sec. 5.6.2.1 No. 6]
       → See [39, 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 [3, 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 [39, 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 [39, 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
```

D 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 1, 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`
script_client_index	`script_client_index`
script_as_form	`script_as_form`
script_as_ciba_form	`script_as_ciba_form`

Table 1. 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*.

E 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 Web systems). However, in order to do so, we first need some definitions.

Notion of an AT being bound to key, AS, Client Id, and identity. We recall and explain in more detail Definition 1, capturing 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 16 (Access Token bound to Key, Authorization Server, Client Id, and Identity). *Let $k \in \mathcal{T}_{\mathcal{N}}$ 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 $rec \in^{\langle\rangle} S(as).\texttt{records}$ such that*

$$rec[\texttt{access\_token}] \equiv t \wedge \tag{1}$$
$$rec[\texttt{subject}] \equiv id \wedge \tag{2}$$
$$rec[\texttt{client\_id}] \equiv clientId \wedge \tag{3}$$
$$\big((rec[\texttt{cnf}] \equiv [\texttt{jkt}\colon \texttt{hash}(k)]) \vee \tag{4}$$
$$(rec[\texttt{cnf}] \equiv [\texttt{x5t\#S256}\colon \texttt{hash}(k)])\big) \tag{5}$$

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 17 (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 Web system $\mathcal{FAPI}$), if all of the following hold true:*

(i) $P = (S^p, E^p, N^p) \xrightarrow[\mathsf{AS} \to E^P_{out}]{e^P_{in} \to \mathsf{AS}} (S^{p+1}, E^{p+1}, N^{p+1})$

(ii) $e^P_{in} = \langle x^P, y^P, m^P \rangle$*, with* $m^P = \mathsf{enc_a}(\langle regReq, k \rangle, pk_{\mathsf{AS}})$*, where regReq* matches $\langle \mathsf{HTTPReq}, n, \mathsf{POST}, d_{\mathsf{AS}}, *, *, *, regData \rangle$ *(Definition 43).*

(iii) There is a processing step $Q = (S^q, E^q, N^q) \xrightarrow[\mathsf{C} \to E^Q_{out}]{e^Q_{in} \to \mathsf{C}} (S^{q+1}, E^{q+1}, N^{q+1})$ *prior to* P *in* ρ *such that there is an event* $\langle x, y, m^P \rangle \in E^Q_{out}$*, i.e.,* C emits m^P *in* Q *(Definition 84).*

(iv) $E^P_{out} = \langle \langle y^P, x^P, resp \rangle \rangle$*, with* $resp = \mathsf{enc_s}(\langle \mathsf{HTTPResp}, n, 201, \langle\rangle, regResp \rangle, k)$ *(i.e., a response to the request in* m^P*), where* $regResp[\texttt{client\_id}] = clientId$*.*

Lemma 1. *If a client identifier clientId has been issued to* C *by an honest* AS $\in$ AS *in processing step* $P = (S, E, N) \to (S', E', N')$ *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 $clientId \in S'(\mathsf{AS}).\texttt{clients}$.

(III) Condition (ii) in Definition 17 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 17) 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 17). 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) $clientId \in S'(\mathsf{AS}).\texttt{clients}$.

(III). An honest AS only outputs an event as described in condition (iv) of Definition 17 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 41, where the input event must match $\mathsf{enc_a}(\langle\langle \mathsf{HTTPReq}, *, *, *, *, *, *, * \rangle, * \rangle, *)$ (see Line 8 of Algorithm 41). Therefore, the format of the input event e^P_{in} 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 41, 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 18 (HTTPS Response to HTTPS Request Sent by p to p'). *Let $p, p' \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{B} \cup \mathsf{RS}$, and ρ some run of a FAPI Web system $\mathcal{FAPI}$ 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) $resp \in \mathsf{HTTPSResponses}$, *i.e.,* $resp \sim \mathsf{enc_s}(\langle \mathsf{HTTPResp}, n, *, *, * \rangle, k)$
(ii) $req \in \mathsf{HTTPSRequests}$, *i.e.,* $req \sim \mathsf{enc_a}(\langle\langle \mathsf{HTTPReq}, n', *, *, *, *, *, * \rangle, k' \rangle, pubKey)$
(iii) $\exists d_{p'} \in \mathsf{dom}(p')$ *such that* $\mathsf{tlskey}(d_{p'}) \equiv pubKey$
(iv) $k \equiv k'$
(v) $n \equiv n'$
(vi) There is a processing step $Q = (S^q, E^q, N^q) \xrightarrow[p' \to E^Q_{out}]{} (S^{q'}, E^{q'}, N^{q'})$ *in ρ, such that there is an event* $\langle x, y, resp \rangle \in E^Q_{out}$.
(vii) Prior to Q, there is a processing step $R = (S^r, E^r, N^r) \xrightarrow[p \to E^R_{out}]{} (S^{r'}, E^{r'}, N^{r'})$ *in ρ, such that there is an event* $\langle x', y', req \rangle \in E^R_{out}$.

E.1 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.

E.1.1 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 [54] on how our model manages resources). (ii) The token *at* is bound to some key k, AS, the user identity *id*, and some client identifier *clientId* (see Definition 1. (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 19 (Authorization Property). *We say that a FAPI Web system $\mathcal{FAPI}$* is secure w.r.t. authorization *iff for every run* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of $\mathcal{FAPI}$, every RS* $rs \in \mathsf{RS}$ *that is honest in S^n, every identity* $id \in^{\langle\rangle} s_0^{rs}.\mathtt{ids}$ *with* $b = \mathsf{ownerOfID}(id)$ *being an honest browser in S^n, every processing step* $Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \to E^Q_{out}]{e^Q_{in} \to rs} (S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$ *in ρ, every resourceID* $\in \mathbb{S}$ *with* $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ *being honest in S^Q, it holds true that:*[11]

[11] $\mathsf{authorizationServerOfResource}^{rs}$ is a mapping from resource ids to the authorization server that manages the respective resource, see Definition 12.

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

(i) $\exists$ *a processing step* $P = s_i \xrightarrow[rs \to E^{P}_{out}]{e^{P}_{in} \to rs} s_{i+1}$ *such that*
(i.a) either $P = Q$*, or* P *is prior to* Q *in* ρ*, and*
(i.b) e^{P}_{in} *is an event* $\langle x, y, \mathsf{enc_a}(\langle m_{req}, k_1\rangle, k_2)\rangle$ *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.,* $t \equiv m_{req}.\mathtt{headers}[\mathtt{Authorization}].2$*, 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* $k \in \mathcal{T}_{\mathcal{N}}$*, as, a client identifier clientId, and id in* S^{Q} *(see Definition 1).*
(iii) If there exists a client $c \in \mathsf{C}$ *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.,* $r \notin d_{\emptyset}(S^n(\mathsf{attacker}))$*.*

E.1.2 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 by recalling the following definition (Definition 4), capturing that the client logged in a user with a service session id, before presenting the authentication property itself.

Definition 20 (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 Web system with network attacker* $\mathcal{FAPI}$ *iff there exists some session id* x *and a domain* $d_{\mathsf{AS}} \in \mathsf{dom}(\mathsf{governor}(id))$ *such that* $S(\mathsf{C}).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d_{\mathsf{AS}}, id\rangle$ *and* $S(\mathsf{C}).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$*.*

Definition 21 (Authentication Property). *We say that a FAPI 2.0 Web system with network attacker* $\mathcal{FAPI}$ is secure w.r.t. authentication *iff for every run* ρ *of* $\mathcal{FAPI}$*, every configuration* (S, E, N) *in* ρ*, every* $\mathsf{C} \in \mathsf{C}$ *that is honest in* S*, every identity id* $\in \mathsf{ID}$ *with* $\mathsf{AS} = \mathsf{governor}(id)$ *being an honest AS (in* S*) and with* $b = \mathsf{ownerOfID}(id)$ *being an honest browser in* S*, every service session identified by some nonce* n *for id at* C*,* n *is not derivable from the attackers knowledge in* S *(i.e.,* $n \notin d_{\emptyset}(S(\mathsf{attacker}))$*).*

E.1.3 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 22 (User is logged in). *For a run* ρ *of a FAPI Web system with a network attacker* $\mathcal{FAPI}$ *we say that a browser* b *was authenticated to a client* c *using an authorization server as and an identity id in a login session identified by a nonce lsid in processing step* Q *in* ρ *with*

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

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

Definition 23 (User started authorization code login flow). *For a run* ρ *of a FAPI Web system with a network attacker FAPI we say that the user of the browser* b *started a login session identified by a nonce* $lsid$ *at the client* c *in a processing step* Q *in* ρ *if (1) in that processing step, the browser* b *was triggered, selected a document loaded from an origin of* c*, executed the script script_client_index in that document, and in that script, executed 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* $\langle$`Set-Cookie`, [$\langle$`__Host`, `sessionId`$\rangle$: $\langle lsid, \top, \top, \top\rangle$]$\rangle$. *We then write* $\mathsf{started}^{Q}_{\rho}(b, c, lsid)$.

Definition 24 (User started CIBA login flow). *For a run* ρ *of a FAPI Web system with a network attacker FAPI 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* $\langle$`HTTPReq`, $*$, $*$, *clientDom*, /`start_ciba`, $*$, $\langle\rangle$, *body*$\rangle$, *with* $clientDom \in \mathsf{dom}(c)$*, 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* $\langle$`Set-Cookie`, [$\langle$`__Host`, `sessionId`$\rangle$: $\langle lsid, \top, \top, \top\rangle$]$\rangle$. *We then write* $\mathsf{startedCIBA}^{Q}_{\rho}(b, c, lsid)$.

Definition 25 (User authenticated at an AS for authorization code flow). *For a run* ρ *of a FAPI Web system with a network attacker FAPI we say that the user of the browser* b *authenticated to an authorization server as using an identity* id *for a login session identified by a nonce* $lsid$ *at the client* c *if there is a processing step* $Q = (S, E, N) \rightarrow (S', E', N')$ *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* $scriptstate$[`auth2_reference`] $\equiv auth2Reference$*, and*
- *there is a nonce* $requestUri$ *such that* $S(as)$.`authorizationRequests`[$requestUri$][`auth2_reference`] $\equiv auth2Reference$*, and*
- $S(c)$.`sessions`[$lsid$][`request_uri`] $\equiv requestUri$.

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

Definition 26 (User authenticated at an AS for CIBA flow). *For a run* ρ *of a FAPI Web system with a network attacker FAPI we say that the user of the browser* b *authenticated to an authorization server as using an identity* id *for a login session identified by a nonce* $lsid$ *at the client* c *if there is a processing step* $Q = (S, E, N) \rightarrow (S', E', N')$ *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* $scriptstate$[`ciba_auth2_reference`] $\equiv auth2Reference$*, and*
- *there is a nonce* $authnReqId$ *such that* $S(as)$.`cibaAuthnRequests`[$authnReqId$][`ciba_auth2_reference`] $\equiv auth2Reference$*, and*
- $S(c)$.`sessions`[$lsid$][`auth_req_id`] $\equiv authnReqId$.

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

Definition 27 (Resource Access). *For a run ρ of a FAPI Web system with a network attacker $\mathcal{FAPI}$ we say that a browser $b \in \mathsf{B}$ gets access to a resource 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 $Q = (S, E, N) \xrightarrow[c \to E_{out}]{} (S', E', N')$ in ρ if*

- $S(c).\mathtt{sessions}[lsid][\mathtt{cibaFlow}] \equiv \bot$ *and c executes Line 114 of Algorithm 3 in Q, or*
- $\mathtt{resource} \in S(c).\mathtt{sessions}[lsid]$ *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., $\exists \langle x', y', m\rangle \in^{\langle\rangle} E_{out}$ such that $m \sim \mathsf{enc}_s(\langle \mathtt{HTTPResp}, *, *, *, r\rangle, *)$), and it holds true that

(i) $r \in^{\langle\rangle} S'(rs).\mathtt{resourceNonces}[u][resourceId]$ *and $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ (for some value $resourceId \in \mathcal{T}_{\mathcal{N}}$), and*

(ii) $\langle\langle \_\_\mathtt{Host}, \mathtt{sessionid}\rangle, \langle lsid, y, z, z'\rangle\rangle \in^{\langle\rangle} S'(b).\mathtt{cookies}[d]$ *for $d \in \mathsf{dom}(c)$, $y, z, z' \in \mathcal{T}_{\mathcal{N}}$, and*

(iii) $S'(c).\mathtt{sessions}[lsid][\mathtt{resourceServer}] \in \mathsf{dom}(rs)$*, and*

(iv) the request to which the client is responding in Q contains a `Cookie` *header with the cookie* $\langle \_\_\mathtt{Host}, \mathtt{sessionId}\rangle$ *with the value lsid.*

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

Definition 28 (Client Leaked Authorization Request). *Let $\mathcal{FAPI}$ be an FAPI Web system with a network attacker. For a run ρ of $\mathcal{FAPI}$ with a processing step Q, a client $c \in \mathsf{C}$, a browser b, an authorization server $as \in \mathsf{AS}$, an identity id, a login session id lsid, and* $\mathsf{loggedIn}^{Q}_{\rho}(b, c, id, as, lsid)$*, we say that c* leaked the authorization request for *lsid, if there is a processing step $Q' = (S, E, N) \xrightarrow[c \to E_{out}]{} (S', E', N')$ 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 $\langle x, y, m\rangle \in E_{out}$ with $m.1 \equiv \mathtt{LEAK}$ and $m.2.\mathtt{parameters}[\mathtt{request\_uri}] \equiv$ requestUri such that $S'(c).\mathtt{sessions}[lsid][\mathtt{request\_uri}] \equiv$ requestUri.*

E.1.4 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 29 (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 $Q = (S, E, N) \to (S', E', N')$ in ρ, every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity id, every client $c \in \mathsf{C}$ that is honest in S, every nonce lsid with $S(c).\mathtt{sessions}[lsid][\mathtt{cibaFlow}] \equiv \bot$, and* $\mathsf{loggedIn}^{Q}_{\rho}(b, c, id, as, lsid)$ *and c did not leak the authorization request for lsid (see Definition 28), we have that (1) there exists a processing step Q' in ρ (before Q) such that* $\mathsf{started}^{Q'}_{\rho}(b, c, lsid)$*, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that* $\mathsf{authenticated}^{Q''}_{\rho}(b, c, id, as, lsid)$.

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 $Q = (S, E, N) \xrightarrow[b \to E_{out}]{} (S', E', N')$ in ρ, every browser b that is honest in S, every event $\langle x, y, m\rangle \in E_{out}$, we have that if $m \sim \mathsf{enc_a}(\langle\langle \mathtt{HTTPReq}, *, *, *, /\mathtt{start\text{-}ciba}, *, *, body\rangle, *\rangle, *)$, then $body \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathtt{identity} \in body$, and $\mathsf{ownerOfID}(body[\mathtt{identity}]) \equiv b$.*

Definition 30 (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 $Q = (S, E, N) \to (S', E', N')$ in ρ, every browser b that is honest in S and behaves according to Assumption 1, every $as \in \mathsf{AS}$, every identity id, every client $c \in \mathsf{C}$ that is honest in S, every nonce $lsid$ with $S(c).\mathtt{sessions}[lsid][\mathtt{cibaFlow}] \equiv \top$, and $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$ we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{startedCIBA}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticatedCIBA}_\rho^{Q''}(b, c, id, as, lsid)$.*

By *session integrity for authentication* we denote the conjunction of both session integrity for authentication for authorization code flows and FAPI-CIBA (Definition 29 and Definition 30).

E.1.5 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 31 (Session Integrity for Authorizationfor Authorization Code Flows). *Let $\mathcal{FAPI}$ be a FAPI Web system with a network attacker. We say that $\mathcal{FAPI}$* is secure w.r.t. session integrity for authorization for authorization code flows *iff for every run ρ of $\mathcal{FAPI}$, every processing step $Q = (S, E, N) \to (S', E', N')$ in ρ, every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client $c \in \mathsf{C}$ that is honest in S, every $rs \in \mathsf{RS}$ that is honest in S, every nonce r, every nonce $lsid$ with $S(c).\mathtt{sessions}[lsid][\mathtt{cibaFlow}] \equiv \bot$, we have that if $\mathsf{accessesResource}_\rho^Q(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 28), , then (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, c, u, as, lsid)$.*

Definition 32 (Session Integrity for Authorization for FAPI-CIBA). *Let $\mathcal{FAPI}$ be a FAPI Web system with a network attacker. We say that $\mathcal{FAPI}$* is secure w.r.t. session integrity for FAPI-CIBA *iff for every run ρ of $\mathcal{FAPI}$, every processing step $Q = (S, E, N) \to (S', E', N')$ in ρ, every browser b that is honest in S and behaves according to Assumption 1, every $as \in \mathsf{AS}$, every identity u, every client $c \in \mathsf{C}$ that is honest in S, every $rs \in \mathsf{RS}$ that is honest in S, every nonce r, every nonce $lsid$ with $S(c).\mathtt{sessions}[lsid][\mathtt{cibaFlow}] \equiv \top$, we have that if $\mathsf{accessesResource}_\rho^Q(b, r, u, c, rs, as, lsid)$, , then (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{startedCIBA}_\rho^{Q'}(b, c, lsid)$, and (2)*

if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticatedCIBA}_{\rho}^{Q''}(b, c, u, as, lsid)$.

By *session integrity for authorization* we denote the conjunction of both session integrity for authorization for authorization code flows and FAPI-CIBA (Definition 31 and Definition 32).

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

E.2 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.

E.2.1 Signed Authorization Requests.

Definition 33 (Non-Repudiation for Signed Authorization 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 authorization requests *iff for every run ρ of $\mathcal{FAPI}$, every configuration (S^n, E^n, N^n) in ρ, every process $as \in \mathsf{AS}$ that is honest in S^n, every request uri requestUri, we have that if $S^n(as).\mathtt{authorizationRequests}[requestUri][\mathtt{signed\_par}] \equiv \top$, then all of the following hold true:*

(I) There exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \to as} (S', E', N')$ with (S, E, N) prior to (S^n, E^n, N^n) in ρ, such that $requestUri \notin S(as).\mathtt{authorizationRequests}$ and $requestUri \in S'(as).\mathtt{authorizationRequests}$.

(II) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $\mathsf{enc_a}(\langle\langle \mathsf{HTTPReq}, \cdot, \mathtt{POST}, selectedAS, /\mathtt{par}, \cdot, \langle\rangle, body \rangle, \cdot \rangle, \cdot)$, where body is of the form $\mathsf{sig}(par, signKey)$ and $selectedAS \in \mathsf{dom}(as)$.

(III) If there is a process $c \in \mathsf{C}$ which is honest in S^n, and a configuration (S^i, E^i, N^i) in ρ with $S^i(c).\mathtt{asAccounts}[selectedAS][\mathtt{sign\_key}] \equiv signKey$, then there is a processing step $P = (S^j, E^j, N^j) \xrightarrow[c \to E_{out}]{} (S^{j+1}, E^{j+1}, N^{j+1})$ 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.

E.2.2 Signed Authorization Responses.

Definition 34 (Non-Repudiation for Signed Authorization 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 authorization responses *iff for every run ρ of $\mathcal{FAPI}$, every configuration (S^n, E^n, N^n) in ρ, every session id sessionId, every process $c \in \mathsf{C}$ that is honest in S^n, we have that if*

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

(2) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $\mathsf{enc_a}(\langle\langle \mathsf{HTTPReq}, \cdot, \cdot, \cdot, /\mathtt{redirect\_ep}, parameters, headers, \cdot \rangle, \cdot \rangle, \cdot)$, and

(3) $S^n(c).\mathtt{sessions}[sessionId][\mathtt{requested\_signed\_authz\_response}] \equiv \top$,

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* $\mathsf{sig}(authzResponse, signKey)$*, with authzResponse being a dictionary with at least the keys* `iss` *and* `code`*.*

(II) If there is an as $\in \mathsf{AS}$ *with* $S^n(as).\mathtt{jwk} \equiv signKey$*, 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.

E.2.3 Signed Introspection Responses.

Definition 35 (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* $rs \in \mathsf{RS}$ *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* $S(rs).\mathtt{pendingResponses}[requestId][\mathtt{requestSignedIntrospecResponse}] \equiv \top$*, and* $requestId \notin S'(rs).\mathtt{pendingResponses}$*, 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* $\mathsf{enc_s}(\langle \mathsf{HTTPResp}, \cdot, \cdot, \cdot, body\rangle, \cdot)$*, where body is of the form* $\mathsf{sig}(introspecResponse, signKey)$*.*

(II) If there is an as $\in \mathsf{AS}$ *with* $S^n(as).\mathtt{jwk} \equiv signKey$*, 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.*

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.

E.2.4 Signed Resource Requests.

Definition 36 (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* $rs \in \mathsf{RS}$ *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 $\mathsf{enc_s}(\langle \mathsf{HTTPResp}, \cdot, \cdot, \cdot, body\rangle, \cdot)$ *with* $body \equiv [\mathtt{resource}\colon resource]$*.*

(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* $e_{in} = \langle y, x, resReq\rangle$*, and rs generated resource during* R *in Line 48 of Algorithm 18.*

(III) resReq is of the form $\mathsf{enc_a}(\langle\langle \texttt{HTTPReq}, \cdot, \textit{method}, \textit{host}, \textit{path}, \textit{parameters}, \textit{headers}, \textit{body}\rangle, \cdot\rangle, \cdot)$ *with* `Signature` $\in$ *headers,* `Signature-Input` $\in$ *headers, and headers*[`Signature`] *being a dictionary with at least a key* `req` *with value* $\mathsf{sig}(\textit{signatureBase}, \textit{clientSignKey})$.

(IV) headers[`Signature-Input`][`req`] *is a sequence* $\langle \textit{coveredComponents}, \textit{metadata}\rangle$ *(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*, $\bot\rangle$, $\langle$`authorization`, $\langle\rangle\rangle$`:` *headers*[`Authorization`], $\langle$`content-digest`, $\langle\rangle\rangle$`:` $\mathsf{hash}(\textit{body})$] $+^{\langle\rangle}$ [`tag:` `fapi-2-request`, `keyid:` *keyId*] *for some keyId; however, the dictionaries may contain additional elements.*

(VI) If there is a client $c \in \mathsf{C}$ *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* $P = (S^i, E^i, N^i) \xrightarrow[c \to E'_{out}]{} (S^{i+1}, E^{i+1}, N^{i+1})$ *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.

E.2.5 Signed Resource Responses.

Definition 37 (Non-Repudiation for Signed Resource 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 resource responses *iff for every run* $\rho = ((S^0, E^0, N^0), \ldots)$ *of $\mathcal{FAPI}$, every configuration* (S^n, E^n, N^n) *in* ρ*, every client* $c \in \mathsf{C}$ *which is honest in* S^n*, every nonce resource, and every session id* $\textit{sessionId} \in S^n(c)$.`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 $P = (S, E, N) \xrightarrow{e_{in} \to c} (S', E', N')$ *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* $\mathsf{enc_s}(\langle\texttt{HTTPResp}, \cdot, \textit{status}, \textit{headers}, \textit{body}\rangle, \cdot)$*, 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`, $\langle\rangle\rangle$,

$\langle$`content-digest`, $\langle\rangle\rangle \in^{\langle\rangle}$ *components, and metadata is a dictionary with at least the key* `tag` *such that* *metadata*[`tag`] ≡ `fapi-2-response`.

(III) *headers*[`Signature`] *is a dictionary with at least a key* `res` *such that* *headers*[`Signature`][`res`] ≡ sig(*signatureBase*, *rsSigKey*). *In addition, signatureBase is of the form* [$\langle$`@status`, $\langle\rangle\rangle$: *status*, $\langle$`content-digest`, $\langle\rangle\rangle$: hash(*body*)] $+^{\langle\rangle}$ [`tag`: `fapi-2-response`, `keyid`: *keyId′*] *for some keyId′; however, the dictionaries may contain additional elements.*

(IV) *There exists a domain* $rsDom \in S^n(c).\texttt{rsSigKeys}$ *such that* $S^n(c).\texttt{rsSigKeys}[rsDom] \equiv \texttt{pub}(rsSigKey)$.

(V) *If process* $rs := \texttt{dom}^{-1}(rsDom)$ *is honest in* S^n, *then there is a processing step* $Q = s \xrightarrow[rs \to E_{out}]{} s'$, *and rs signed the resource response contained in m during Q in Line 6 of Algorithm 20.*

E.3 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).

F PROOFS

F.1 Helper Lemmas

Lemma 2 (Host of HTTP Request). *For any run ρ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ and every process $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ that is honest in S it holds true that if the generic HTTPS server calls* PROCESS_HTTPS_REQUEST(m_{dec}, k, a, f, s) *in Algorithm 41, then* $m_{dec}.\texttt{host} \in \texttt{dom}(p)$, *for all values of k, a, f and s.*

Proof. PROCESS_HTTPS_REQUEST is called only in Line 9 of Algorithm 41. The input message m is an asymetrically encrypted ciphertext. Intuitively, such a message is only decrypted if the process knows the private TLS key, where the private key used to decrypt is chosen (non-deterministically) according to the host of the decrypted message.
More formally, when PROCESS_HTTPS_REQUEST is called, the **stop** in Line 8 is not called. Therefore, it holds true that

$$\begin{aligned} & \exists\, inDomain, k' : \langle inDomain, k' \rangle \in S(p).\texttt{tlskeys} \wedge m_{dec}.\texttt{host} \equiv inDomain \\ \Rightarrow\ & \exists\, inDomain, k' : \langle inDomain, k' \rangle \in \texttt{tlskeys}^p \wedge m_{dec}.\texttt{host} \equiv inDomain \\ \overset{\text{Def. (Appendix C.3)}}{\Rightarrow}\ & \exists\, inDomain, k' : \langle inDomain, k' \rangle \in \{\langle d, tlskey(d)\rangle | d \in \texttt{dom}(p)\} \\ & \qquad \wedge\, m_{dec}.\texttt{host} \equiv inDomain \end{aligned}$$

From this, it follows directly that $m_{dec}.\texttt{host} \in \texttt{dom}(p)$.
The first implication holds true due to $S(p).\texttt{tlskeys} \equiv s_0^p.\texttt{tlskeys} \equiv \texttt{tlskeys}^p$, as this sequence is never changed by any honest process $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ and due to the definitions of the initial states of clients, authorization servers, and resource servers (Definition 13, Definition 14, Definition 15).

□

Lemma 3 (Generic Server – Correctness of Reference and Request). *For any run ρ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every processing step $P = (S^P, E^P, N^P) \to (S^{P\prime}, E^{P\prime}, N^{P\prime})$ in ρ, every $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ being honest in S^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 $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ be honest in S^P. p calls the PROCESS_HTTPS_RESPONSE function only in the generic HTTPS server algorithm in Line 26 of Algorithm 41. The values *reference* and *request* are taken from $S^P(p)$.pendingRequests in Line 19 of Algorithm 41. 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 41, 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 13, 14, and 15). In O, the process p takes both values from $S^O(p)$.pendingDNS in Line 13 and Line 14 of Algorithm 41. 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 36, where the reference and request values are the input arguments of HTTPS_SIMPLE_SEND. Thus, in some processing step prior to O, p called HTTPS_SIMPLE_SEND with *reference* being the first and *request* being the second input argument.

□

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

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*[sigKey] 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*[sigKey] is a fresh nonce v_{cliSignK} (see Line 13 of Algorithm 8), which is not stored anywhere else and is only sent out as $\mathsf{pub}(v_{\text{cliSignK}})$ during Q. In addition, the *reference* with v_{cliSignK} in it is not used anywhere until P: A client only accesses its pendingRequests state subterm in Line 41 of Algorithm 41, and if an entry of pendingRequests is used at all, it is immediately removed from the pendingRequests subterm (see Line 25 of Algorithm 41), 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*[sigKey] 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 $\mathsf{pub}(\cdot)$. 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(·), i.e., as a public key, from which the original value cannot be recovered (see the equational theory in Figure 10), 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(·, *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 10).
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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of a FAPI Web system* $\mathcal{FAPI}$ *with a network attacker, every configuration* (S, E, N) *in* ρ*, every client* $c \in \mathsf{C}$ *that is honest in* S*, every domain* $d_c \in \mathsf{dom}(c)$*, and every process* p *with* $p \neq c$*, all of the following hold true:*

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

Proof. With Definition 13, $\langle d_c, \mathsf{tlskey}(c)\rangle \in^{\langle\rangle} S^0(c).\mathtt{tlskeys}$ is equivalent to $\langle d_c, \mathsf{tlskey}(d_c)\rangle \in^{\langle\rangle} \mathit{tlskeys}^c$. This, in turn, follows immediately from the definition of $\mathit{tlskeys}^c$ in Appendix C.3. Building on this, it is easy to check that the client never changes the contents of its tlskeys state subterm, i.e., we have $\langle d_c, \mathsf{tlskey}(d_c)\rangle \in^{\langle\rangle} S(c).\mathtt{tlskeys}$.

The only place in which an honest client accesses any value in its tlskeys state subterm is Line 7 of Algorithm 41, 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, $\mathit{tlskeys}^p$ in Appendix C.3 and the initial states of authorization servers (Definition 14), clients (Definition 13), browsers (Definition 9), and resource servers (Definition 15), we have that no other process initially knows $\mathsf{tlskey}(d_c)$.

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

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

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 v_{cliTlsK} (see Line 14 of Algorithm 8). I.e., v_{cliTlsK} is not derivable by any process prior to Q. Furthermore, v_{cliTlsK} is only used in two places (during processing step Q):

- **Line 15 of Algorithm 8** Here, v_{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, v_{cliTlsK} is passed to HTTPS_SIMPLE_SEND as part of the first argument, i.e., the aforementioned *reference* (see Algorithm 36). This *reference* is stored in the client's `pendingDNS` state subterm (Line 2 of Algorithm 36) and not used anywhere else. Values stored in the client's `pendingDNS` state subterm are only accessed in Lines 10ff. of Algorithm 41, 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 41, 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 v_{cliTlsK}, is no longer in the client's state once *reference* is passed to PROCESS_HTTPS_RESPONSE. Hence, we established that the use of v_{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
$\forall j \leq i.\ issuer \in S^j(c).\texttt{asAccounts} \Rightarrow S^j(c).\texttt{asAccounts}[issuer][\texttt{tls\_key}] \notin d_\emptyset(S^i(p))$. □

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

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

*in ρ with $c \in \mathsf{C}$ 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] ≡ TOKEN *and* code $\in^{\langle\rangle}$ *request*.body*, then there is a previous configuration $(S^{L\prime}, E^{L\prime}, N^{L\prime})$ such that request*.body[code] ≡ $S^{L\prime}(c)$.sessions[*reference*[session]][redirectEpRequest][message].parameters[code].

Proof. As shown in Lemma 3, there exists a processing step $L = (S^L, E^L, N^L) \to (S^{L\prime}, E^{L\prime}, N^{L\prime})$ 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 *reference*[responseTo] ≡ 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 $S^{L\prime}(c)$.sessions[*sessionId*][redirectionEpRequest][message] in Line 33 of Algorithm 2, with *sessionId* ≡ *reference*[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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration $Q = (S, E, N)$ in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every term t with* checksig$(t, \mathrm{pub}(\mathrm{signkey}(as))) \equiv \top$*, and every process p with $p \neq as$, all of the following hold true:*

- signkey$(as) \notin d_\emptyset(S(p))$
- signkey$(as) \equiv s_0^{as}$.jwk
- signkey$(as) \equiv S(as)$.jwk
- *if t is* known *(Definition 89) to p in Q, then t was* created *(Definition 87) by as in a processing step s_e prior to Q in ρ*

Proof. signkey$(as) \equiv s_0^{as}$.jwk immediately follows from Definition 14. signkey$(as) \equiv S(as)$.jwk follows from Definition 14 and by induction over the processing steps: state subterm jwk of an honest authorization server is never changed.

By Definitions 13, 14, 15, 72, and Appendix C.3, we have that no process (except *as*) initially knows signkey(as), i.e., signkey$(as) \notin d_\emptyset(S^0(p))$.

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

Lines 15f. of Algorithm 11 Here, the value of the `jwk` state subterm is only used in a $\mathsf{pub}(\cdot)$ term constructor as private key from which a public key is derived, i.e., cannot be extracted from the resulting public key (see Figure 10). 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 `jwk` state subterm is only used in a $\mathsf{sig}(\cdot,\cdot)$ term constructor as signature key, i.e., cannot be extracted from the resulting term (see Figure 10). Thus, it does not matter where that term are stored or sent to.

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

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

By Definitions 13, 14, 15, 9, and Appendix C.3, 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 63 and Definition 87, 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 82), e must have been *emitted* by some process in a processing step s_e prior to Q' in ρ. Definition 63 and Definition 84 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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration $s = (S, E, N)$ in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S^n, every client $c \in \mathsf{C}$ that is honest in S^n and has been issued client id $clientId$ by as in some processing step $R = s^r \to s^{r+1}$ with s^r prior to s in ρ, every $i \in \mathbb{N}$ with $0 < i \leq |S(as).\mathtt{mtlsRequests}[clientId]|$, and every process p with $as \neq p \neq c$ it holds true that $mtlsNonce := S(as).\mathtt{mtlsRequests}[clientId].i.1$ is not derivable by p, i.e., $mtlsNonce \notin d_\emptyset(S^n(p))$.*

Proof.

(A) **Initial state.** Initially, the `mtlsRequests` subterm of the authorization server's state is empty: $S^0(as).\mathtt{mtlsRequests} \equiv \langle\rangle$ (Definition 14).

(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 $P = (S^j, E^j, N^j) \to (S^{j+1}, E^{j+1}, N^{j+1})$ 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 $clientId \in S^j(as).\mathtt{clients}$ (otherwise, the check in Line 232 of Algorithm 11 would fail). Since an honest authorization server never removes entries from its `clients` state subterm, $clientId \in S^j(as).\mathtt{clients}$ implies $clientId \in S^l(as).\mathtt{clients}$ 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 $S^l(as).\mathtt{clients}[clientId][\mathtt{mtls\_key}] \equiv \mathsf{pub}(k_{\text{mtls}})$, 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 $clientKey \equiv S^j(as).\mathtt{clients}[clientId][\mathtt{mtls\_key}]$ (Line 235 of Algorithm 11). We will refer to this ciphertext $\mathsf{enc_a}(\langle mtlsNonce, x\rangle, clientKey)$ as *mtlsResp*.
From (B), we have $\exists k_{\text{mtls}} \in \mathcal{N}.\ clientKey \equiv \mathsf{pub}(k_{\text{mtls}})$, 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 10).

(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 once: 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 41), 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 `mtlsCache` state subterm. As in the previous cases, the client removes the corresponding sequence from the `mtlsCache` 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 `mtlsCache` subterm). Let $req_{c \to as}$ be the HTTP request that the client sends by calling HTTPS_SIMPLE_SEND.

HTTPS_SIMPLE_SEND stores the request $req_{c \to as}$ (which contains the mTLS nonce) in the `pendingDNS` state subterm of c, see Line 2 of Algorithm 36 and then stops with the DNS request (which does not contain the nonce) in Line 3 of Algorithm 36. 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 41, 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 41, or creates a new `pendingRequests` entry containing the request $req_{c \to as}$ (and thus, also the mTLS nonce) in Line 15 of Algorithm 41. In this case, the client removes the request from the `pendingDNS` state subterm in Line 17 of Algorithm 41, 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 $req_{c \to as}$ with the key of the domain that was stored along with the mTLS nonce, i.e., a key of *as*, see Lines 16 and 18 of Algorithm 41, and (E).

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

As the client encrypts $req_{c \to as}$ asymetrically with a key of *as*, it follows that only *as* can decrypt the HTTPS request (Lemma 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 41, as this is the only place where an authorization server decrypts a message asymetrically, and then used as a function argument for PROCESS_HTTPS_REQUEST which is modeled in Algorithm 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 `mtlsRequests` state subterm and returns this entry; the `/par` endpoint code does not use this value. The token endpoint uses this value for token binding (Lines 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 $req_{c\to as}$, the generic HTTPS server removes the message from the *pendingRequests* state subterm and calls PROCESS_HTTPS_RESPONSE with the request as the third function argument (see Lines 19ff. of Algorithm 41). 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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of a FAPI Web system* $\mathcal{FAPI}$ *with a network attacker, every configuration* $Q = (S, E, N)$ *in* ρ*, every resource server* $rs \in \mathsf{RS}$ *that is honest in* S*, every term* t *with* $\mathsf{checksig}(t, \mathsf{pub}(\mathsf{signkey}(rs))) \equiv \top$*, and every process* p *with* $p \neq rs$*, all of the following hold true:*

- $\mathsf{signkey}(rs) \notin d_\emptyset(S(p))$
- $\mathsf{signkey}(rs) \equiv s_0^{rs}.\mathtt{jwk}$
- $\mathsf{signkey}(rs) \equiv S(rs).\mathtt{jwk}$
- *if* t *is* known *(Definition 89) to* p *in* Q*, then* t *was* created *(Definition 87) by* rs *in a processing step* s_e *prior to* Q *in* ρ

Proof. $\mathsf{signkey}(rs) \equiv s_0^{rs}.\mathtt{jwk}$ immediately follows from Definition 15. $\mathsf{signkey}(rs) \equiv S(rs).\mathtt{jwk}$ follows from Definition 15 and by induction over the processing steps: state subterm `jwk` of an honest resource server is never changed.

By Definitions 13, 14, 15, 72, and Appendix C.3, we have that no process (except rs) initially knows $\mathsf{signkey}(rs)$, i.e., $\mathsf{signkey}(rs) \notin d_\emptyset(S^0(p))$.

The only place in which an honest resource server accesses the `jwk` state subterm is Line 6 of Algorithm 20. There, the value of the `jwk` state subterm is only used in a $\mathsf{sig}(\cdot,\cdot)$ term constructor as signature key, i.e., cannot be extracted from the resulting term (see Figure 10). Thus, it does not matter where that term are stored or sent to. We conclude that this usage of the `jwk` state subterm does not leak $\mathsf{signkey}(rs)$ to any other process, in particular p, and hence, $\mathsf{signkey}(rs) \notin d_\emptyset(S(p))$.

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

By Definitions 13, 14, 15, 9, and Appendix C.3, we have that no process (including rs) initially knows such a term t, i.e., $t \notin d_\emptyset(S^0(p))$. Together with Definition 63 and Definition 87, 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 82), e must have been *emitted* by some process in a processing step s_e prior to Q' in ρ. Definition 63 and Definition 84 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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of a FAPI Web system* $\mathcal{FAPI}$ *with a network attacker, every configuration* (S, E, N) *in* ρ*, every resource server* $rs \in \mathsf{RS}$ *that is honest in* S^n*, every client* $c \in \mathsf{C}$ *that is honest in* S^n*, every nonce* $k_{mtls} \in \mathcal{N}$*, every* $i \in \mathbb{N}$ *with* $0 \leq i \leq |S(rs).\mathtt{mtlsRequests}|$ *and with* $S(rs).\mathtt{mtlsRequests}.i.2 \equiv \mathsf{pub}(k_{mtls})$*, 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*

$k_{mtls} \notin d_\emptyset(S^n(p_1))$, *then* $mtlsNonce := S(rs).\texttt{mtlsRequests}.i.1$ *does not leak to* p_2*, i.e.,* $mtlsNonce \notin d_\emptyset(S^n(p_2))$.

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., $S^0(rs).\texttt{mtlsRequests} \equiv \langle\rangle$ (Definition 15). 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 $\mathrm{pub}(k_{\mathrm{mtls}})$ (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 41), 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 $\langle request.\texttt{host}, clientId, pubKey, mtlsNonce\rangle$ into the `mtlsCache` subterm of its state (Line 8 of Algorithm 3), where $clientId, pubKey \in \mathcal{T}_{\mathcal{N}}$ and, in particular,

- $request.\texttt{host}$ 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 $req_{c\to rs}$ be the request that the client sends by calling HTTPS_SIMPLE_SEND.

HTTPS_SIMPLE_SEND stores the request $req_{c\to rs}$ (which contains the mTLS nonce) in the `pendingDNS` state subterm of c, see Line 2 of Algorithm 36, and then stops with the DNS request (which does not contain the nonce) in Line 3 of Algorithm 36. 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 41, 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 41, or creates a new `pendingRequests` entry containing the request $req_{c\to rs}$ (and thus, also the mTLS nonce) in Line 15 of Algorithm 41. In this case, the client removes the request from the `pendingDNS` state subterm in Line 17 of Algorithm 41, i.e., regarding the client state, the mTLS nonce is only contained in the newly created `pendingRequests` entry. The client finishes the processing step by encrypting $req_{c\to rs}$ with the key of the domain that was stored along with the mTLS nonce, i.e., a key of rs, see Lines 16 and 18 of Algorithm 41.

***rs* does not leak mTLS nonce contained in request.** As the HTTP request $req_{c\to rs}$ is is encrypted asymetrically with a key of rs, it follows that only the resource server can decrypt the request. The resource server only decrypts terms in the generic HTTPS server algorithms. More specifically,

this request is decrypted (only) in Line 7 of Algorithm 41, as this is the only place where an resource server decrypts a message asymetrically, and then used as a function argument for PROCESS_HTTPS_REQUEST which is modeled in Algorithm 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 $req_{c\to rs}$:

- $req_{c\to rs}$.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.
- $req_{c\to rs}$.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 $req_{c\to rs}$ (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).
- $req_{c\to rs}$.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.
- $req_{c\to rs}$.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 $req_{c\to rs}$, the generic HTTPS server removes the message from the *pendingRequests* state subterm and calls PROCESS_HTTPS_RESPONSE with the request as the third function argument. Algorithm 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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration (S^i, E^i, N^i) in ρ, every 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*

$s \longrightarrow s'$ with s prior to (S^i, E^i, N^i) in ρ), every domain issuer $\in \mathsf{dom}(as)$*, every index* $j \leq i$*, every term* $clientSignKey := S^j(c).\mathtt{asAccounts}[issuer][\mathtt{sign\_key}]$*, every term* t *with*

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

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 $\mathsf{checksig}(t, \mathsf{pub}(clientSignKey)) \equiv \top$ (see Figure 10). 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 $\mathsf{extractmsg}(t)[\mathtt{aud}]$ (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 $\mathsf{extractmsg}(t)[\mathtt{aud}]$ 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 41). 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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of a FAPI Web system* $\mathcal{FAPI}$ *with a network attacker, every authorization server* $as \in \mathsf{AS}$*, every client* $c \in \mathsf{C}$*, every processing step* Q *in* ρ

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

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

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

then c created m *(Definition 87).*

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

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

(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 41. Since we have E_{out} not empty, Q cannot finish at one of the parameterless **stop**s in Algorithm 41. We also have $m \not\equiv$ CORRUPT and *as* is honest, i.e., $S(as).\mathtt{corrupt} \equiv \bot$, and therefore, Q does not stop in Line 6 of Algorithm 41.
The **stop** in Line 18 of Algorithm 41 cannot be reached, since $m \notin$ DNSResponses (see Appendix G.2.5).
All other **stop**s within Algorithm 41 are parameterless, hence, execution during Q must reach one of the function calls in Algorithm 41:
Line 9 of Algorithm 41 (PROCESS_HTTPS_REQUEST**)** As the third element within e_{in} has the correct structure, this function call can be reached.

Line 24 of Algorithm 41 (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 41 (PROCESS_HTTPS_RESPONSE**)** Since $m \notin$ HTTPResponses (Definition 54), this function call cannot be reached (due to the check in Line 23 of Algorithm 41).

Line 28 of Algorithm 41 (PROCESS_TRIGGER**)** Since $m \not\equiv$ `TRIGGER`, this function call cannot be reached.

Line 30 of Algorithm 41 (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 41 (see (B)).
If m.`path` $\equiv$ `/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 $\exists k_{\text{jwt}} \in \mathcal{N}$ such that $S(as)$.`clients`[*clientId*][`jwt_key`] $\equiv \mathsf{pub}(k_{\text{jwt}})$, and $k_{\text{jwt}} \notin d_{\emptyset}(S^n(p))$ for any process $p \neq c$. Hence, we have m.body[`client_assertion`] $\sim \mathsf{sig}(*, \mathsf{pub}(k_{\text{jwt}}))$.
A term $t \sim \mathsf{sig}(*, \mathsf{pub}(k_{\text{jwt}}))$ is not part of any processes' initial state (Definition 14, Definition 13, Definition 9, Definition 15). This, together with Figure 10, 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] ≡ *clientId*, and *cli_assertion*[sub] ≡ *clientId*. Furthermore, *cli_assertion*[aud].host ∈ dom(*as*) or *cli_assertion*[aud] ∈ 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 ∈ m.body**.** From Lines 17–19 of Algorithm 12 it follows that

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

Note that client_id ∈ 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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every resource server $rs \in \mathsf{RS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S, every nonce signKey $\in \mathcal{N}$, 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*)) ≡ ⊤
- extractmsg(t)[payload][htu].host ∈ dom(rs),
- ath $\in^{\langle\rangle}$ extractmsg(t)[payload],
- extractmsg(t)[payload][nonce] ∈ S(rs).dpopNonces

it holds true that if signKey $\notin d_\emptyset(S^n(p_1))$, 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 **stop**s before the removal of the nonce from the state of the resource server do not modify the state of the resource server and do not lead to new events.

We note that the dpopNonces state subterm of the resource server does not contain any value twice, as the resource server only adds fresh nonces to the state subterm, see the endpoint in Line 8 of Algorithm 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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of FAPI with a network attacker,*
- *every authorization server as* $\in \mathsf{AS}$ *that is honest in* S^n*,*
- *every client* $c \in \mathsf{C}$ *that is honest in* S^n*,*
- *every client identifier clientId* $\in \mathcal{T}_{\mathcal{N}}$ *that has been issued to* c *by as in some processing step* $R = (S^r, E^r, N^r) \to (S^{r\prime}, E^{r\prime}, N^{r\prime})$ *in* ρ *(according to Definition 17),*

it holds true that $S^n(as).\texttt{clients}[clientId][\texttt{reg\_at}] \equiv S^{r\prime}(as).\texttt{clients}[clientId][\texttt{reg\_at}]$.

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., $S(as).\texttt{clients}[clientId][\texttt{reg\_at}] \equiv S'(as).\texttt{clients}[clientId][\texttt{reg\_at}]$.

Case 1: Line 51 of Algorithm 11. In this case, the AS stores the value *clientInfo*, which is equal to $S(as).\texttt{clients}[clientId]$ (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 $S(as).\texttt{pendingClientIds}$ in Line 2 of Algorithm 13. However, as $clientId \in^{\langle\rangle} S(as).\texttt{clients}$, it follows that $clientId \notin^{\langle\rangle} S(as).\texttt{pendingClientIds}$.

□

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

- *every run* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of FAPI with a network attacker,*
- *every authorization server* $as \in \mathsf{AS}$ *that is honest in* S^n*,*
- *every client* $c \in \mathsf{C}$ *that is honest in* S^n*,*
- *every client identifier* $clientId \in \mathcal{T}_{\mathcal{N}}$ *that has been issued to* c *by* as *in some processing step* $R = (S^r, E^r, N^r) \rightarrow (S^{r\prime}, E^{r\prime}, N^{r\prime})$ *in* ρ *(according to Definition 17),*

it holds true that $S^{r\prime}(as).\texttt{clients}[clientId][\texttt{reg\_at}] \notin d_\emptyset(S^n(attacker))$.

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 14). As the AS issues *clientId* in R, it follows that for all configurations (S', E', N') prior to $(S^{r\prime}, E^{r\prime}, N^{r\prime})$, $clientId \notin^{\langle\rangle} S'(as).\texttt{clients}$, as the AS takes *clientId* from $S^r(as).\texttt{pendingClientIds}$ (Line 2 of Algorithm 13). Initially, `pendingClientIds` is empty (Definition 14), 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 $\langle x, y, \mathsf{enc_a}(\langle regReq, k\rangle, pk_{\mathsf{AS}})\rangle$ be the input event that *as* processes in R. The request $\mathsf{enc_a}(\langle regReq, k\rangle, pk_{\mathsf{AS}})$ was created by c (see Definition 17).
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 $(S^{r\prime}, E^{r\prime}, N^{r\prime})$, the registration access token is only stored in $S^{r\prime}(as).\texttt{clients}[clientId]$ 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 jwks in the body. Let $T = (S^t, E^t, N^t) \rightarrow (S^{t\prime}, E^{t\prime}, N^{t\prime})$ 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* ≡ 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* ∈ dom(as) (Lemma 2). The client stores the registration access token in $S^{t\prime}$.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 $S^{t\prime}$.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.,
$S^{t\prime}$.asAccounts[*selectedAS*][reg_client_uri].host ≡ *selectedAS* and
$S^{t\prime}$.asAccounts[*selectedAS*][reg_client_uri].path ≡ /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 $(S^{t\prime}, E^{t\prime}, N^{t\prime})$, 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 $(S^{t\prime}, E^{t\prime}, N^{t\prime})$.
- 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 $U = (S^u, E^u, N^u) \rightarrow (S^{u\prime}, E^{u\prime}, N^{u\prime})$ be the processing step in which the client accesses the token stored in asAccounts[*selectedAS*][reg_at] (with *selectedAS* ∈ 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] ≡ *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 $P = (S^p, E^p, N^p) \rightarrow (S^{p\prime}, E^{p\prime}, N^{p\prime})$ 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 $S^p(as).\texttt{clients}[clientId\prime][\texttt{active}]$, for some $clientId\prime$, and does not send any messages.

Processing the Management Response: Let $Q = (S^q, E^q, N^q) \rightarrow (S^{q\prime}, E^{q\prime}, N^{q\prime})$ 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 $S^{q\prime}.\texttt{asAccounts}[selectedAS][\texttt{reg\_at}]$ (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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of FAPI with a network attacker,*
- *every authorization server* $as \in \mathsf{AS}$ *that is honest in* S^n*,*
- *every client* $c \in \mathsf{C}$ *that is honest in* S^n*,*
- *every client identifier* $clientId \in \mathcal{T}_{\mathcal{N}}$ *that has been issued to* c *by* as *in some processing step* $R = (S^r, E^r, N^r) \rightarrow (S^{r\prime}, E^{r\prime}, N^{r\prime})$ *in* ρ *(according to Definition 17),*
- *every configuration* (S^i, E^i, N^i) *(at position* i *in* ρ*),*

it holds true that if $clientId \in^{\langle\rangle} S^i(as).\texttt{clients}$*, then:*
$\exists k_{mtls}, k_{jwt} \in \mathcal{N}$ *such that*

(1) $S^i(as).\texttt{clients}[clientId][\texttt{mtls\_key}] \equiv \texttt{pub}(k_{mtls})$*, and*
(2) $S^i(as).\texttt{clients}[clientId][\texttt{jwt\_key}] \equiv \texttt{pub}(k_{jwt})$*, 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 14). As the AS issues *clientId* in R, it follows that for all configurations (S', E', N') prior to $(S^{r'}, E^{r'}, N^{r'})$, $clientId \notin^{\langle\rangle} S'(as).\texttt{clients}$, as the AS takes *clientId* from $S^r(as).\texttt{pendingClientIds}$ (Line 2 of Algorithm 13). Initially, `pendingClientIds` is empty (Definition 14), 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 $clientId \in^{\langle\rangle} S^r(as).\texttt{clients}$ (see Line 22 and Line 56 of Algorithm 11). Let $\langle x, y, \mathrm{enc_a}(\langle regReq, k\rangle, pk_{\mathsf{AS}})\rangle$ be the input event that as processes in R. The request $\mathrm{enc_a}(\langle regReq, k\rangle, pk_{\mathsf{AS}})$ was created by c (see Definition 17). The AS takes the key values from *regReq*, i.e., $\exists i, j \in \mathbb{N}$ s.t.
- $S^{r'}(as).\texttt{clients}[clientId][\texttt{mtls\_key}] \equiv regReq.\texttt{body}[\texttt{jwks}].i.[\texttt{val}]$, and
- $S^{r'}(as).\texttt{clients}[clientId][\texttt{jwt\_key}] \equiv regReq.\texttt{body}[\texttt{jwks}].j.[\texttt{val}]$, and
- $regReq.\texttt{body}[\texttt{jwks}].i.[\texttt{use}] \equiv \texttt{TLS}$, and
- $regReq.\texttt{body}[\texttt{jwks}].j.[\texttt{use}] \equiv \texttt{sig}$

(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 $regReq.\texttt{body}[\texttt{jwks}].i.[\texttt{val}] \equiv \texttt{pub}(t_1)$, and $regReq.\texttt{body}[\texttt{jwks}].j.[\texttt{val}] \equiv \texttt{pub}(t_2)$, 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 36). 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 41 (and removes the value from `pendingDNS` in Line 17 of Algorithm 41).
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 $reference[\texttt{responseTo} \equiv \texttt{REGISTRATION}]$.
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 $reference[\texttt{responseTo}] \equiv \texttt{REGISTRATION}$). 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 $S^i(as).\texttt{clients}[clientId]$, then $S^{i'}(as).\texttt{clients}[clientId][\texttt{mtls\_key}] \equiv S^i(as).\texttt{clients}[clientId][\texttt{mtls\_key}]$ and $S^{i'}(as).\texttt{clients}[clientId][\texttt{jwt\_key}] \equiv S^i(as).\texttt{clients}[clientId][\texttt{jwt\_key}]$, and and the property still holds true. Thus, we consider all cases in which the AS changes $S^i(as).\texttt{clients}[clientId]$. 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 $\langle x, y, \mathsf{enc_a}(\langle \mathit{updateReq}, k\rangle, \mathit{pk}_{\mathsf{AS}})\rangle$ 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 41, i.e., the message is an encrypted HTTP request, see Line 8 of Algorithm 41). The AS updates $S^i(\mathit{as})$.clients[$\mathit{clientId}$] with $\mathit{clientId} \equiv \mathit{updateReq}$.body[clientId] (Line 21 of Algorithm 11). The AS also checks that $\mathit{updateReq}$.headers[Authorization][Bearer] $\equiv S^i(\mathit{as})$.clients[$\mathit{clientId}$][reg_at] (Line 25 and Line 26 of Algorithm 11).
As shown in Lemma 15, the registration access token never changes, i.e., $S^i(\mathit{as})$.clients[$\mathit{clientId}$][reg_at] $\equiv S^{r\prime}(\mathit{as})$.clients[$\mathit{clientId}$][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.
- $S^{i'}(\mathit{as})$.clients[$\mathit{clientId}$][mtls_key] $\equiv \mathit{updateReq}$.body[jwks].$i$.[val], and
- $S^{i'}(\mathit{as})$.clients[$\mathit{clientId}$][jwt_key] $\equiv \mathit{updateReq}$.body[jwks].$j$.[val], and
- $\mathit{updateReq}$.body[jwks].i.[use] $\equiv$ TLS, and
- $\mathit{updateReq}$.body[jwks].j.[use] $\equiv$ 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 $\mathit{updateReq}$.body[jwks].i.[val] $:=$ pub(t_3) and $\mathit{updateReq}$.body[jwks].j.[val] $:=$ 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 $\mathit{reference}'$, which stores the values (only) in the pendingDNS state subterm of the client (Line 2 of Algorithm 36). When processing the corresponding DNS response, the client stores $\mathit{reference}'$ into pendingRequests in Line 15 of Algorithm 41 (and removes the value from pendingDNS in Line 17 of Algorithm 41). 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 $\mathit{reference}'$[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 $\mathit{reference}'$ 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 $S^i(\mathit{as})$.clients[$\mathit{clientId}$][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 $S^i(\mathit{as})$.clients[$\mathit{clientId}$]: The AS chooses a client identifier $\mathit{clientId}'$ from $S^i(\mathit{as})$.pendingClientIds (Line 2 of Algorithm 13) and stores $S^i(\mathit{as})$.clients[$\mathit{clientId}'$]. However, $\mathit{clientId} \neq \mathit{clientId}'$, 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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of FAPI with a network attacker,*
- *every resource server* $rs \in \mathsf{RS}$ *that is honest in* S^n*,*
- *every identity* $id \in^{\langle\rangle} s_0^{rs}.\mathtt{ids}$*,*
- *every processing step in* ρ

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

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

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

(I) There exists a processing step

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

such that

(1) either $P = Q$ *or* P *prior to* Q *in* ρ*, and*
(2) e^P_{in} *is an event* $\langle x, y, \mathsf{enc_a}(\langle m_{req}, k_1\rangle, k_2)\rangle$ *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.,* $t \equiv m_{req}.\mathtt{headers}[\mathtt{Authorization}].2$*, 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 1).*

(III) If there exists a client $c \in \mathsf{C}$ *such that clientId has been issued to* c *by as in a previous processing step (see Definition 17), and if* c *is honest in* S^n*, then the message in* e^P_{in} *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^Q_{in} is an event containing an HTTPS request, as Algorithm 18 is only called by the generic HTTPS server in Line 9 of Algorithm 41. As the check done in Line 7 of Algorithm 41 was true and the `stop` in Line 8 was not executed, it follows that the input event of Algorithm 41 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., $\mathtt{extractmsg}(m_{\text{req}}.\mathtt{headers}[\mathtt{Authorization}].2)$ is a structured access token (see Lines 23 and 49).
The access token is signed by $\mathsf{authorizationServerOfResource}^{rs}(resourceID)$: The value of *responsibleAS* (in Line 16) is equal to

$$
\begin{aligned}
& S^{Q}(rs).\texttt{resourceASMapping}[resourceID] && \text{(Line 16 of Algorithm 18)}\\
\equiv\ & s_0^{rs}.\texttt{resourceASMapping}[resourceID] && \text{(value is never changed)}\\
\in\ & \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(resourceID)) && \text{(Definition 15)}
\end{aligned}
$$

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

$$
\begin{aligned}
& asInfo[\texttt{as\_key}]\\
\equiv\ & S^{Q}(rs).\texttt{asInfo}[responsibleAS][\texttt{as\_key}] && (responsibleAS \in \mathsf{dom}(as), \text{Line 19})\\
\equiv\ & s_0^{rs}.\texttt{asInfo}[responsibleAS][\texttt{as\_key}] && \text{(value is never changed)}\\
\equiv\ & \mathsf{signkey}(\mathsf{dom}^{-1}(responsibleAS)) && \text{(Definition 15)}\\
\equiv\ & \mathsf{signkey}(as)
\end{aligned}
$$

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 $\mathsf{extractmsg}(m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2)$ 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 $O = (S^O, E^O, N^O) \xrightarrow[as \to E^O_{\text{out}}]{e^O_{\text{in}} \to as} (S^{O\prime}, E^{O\prime}, N^{O\prime})$ be the processing step in which the authorization server created and signed the access token. After finishing the processing step, as stores the access token in $S^{O\prime}(as).\texttt{records}.i[\texttt{access\_token}]$, for some natural number i (as Line 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 $\mathsf{extractmsg}(m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2)[\texttt{sub}] \in^{\langle\rangle} S^{Q}(rs).\texttt{ids}$ (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., $S^{Q}(rs).\texttt{ids} \equiv s_0^{rs}.\texttt{ids}$. When creating the access token, the authorization server takes this value from $S^{O}(as).\texttt{records}.i[\texttt{sub}]$ with the same i as above (Line 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 $S^{O}(as).\texttt{records}.i[\texttt{sub}] \equiv S^{O\prime}(as).\texttt{records}.i[\texttt{sub}]$.

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

- $accessTokenContent$[cnf].1 ≡ x5t#S256 or
- $accessTokenContent$[cnf].1 ≡ 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., $S^{O'}(as)$.records.i[cnf] is either [jkt: hash(k)] or [x5t#S256: 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., $S^{O'}(as)$.records.i[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 do 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., $e^{O}_{\text{in}} = \langle x', y', req_{\text{token}} \rangle$, for some values x',y'. In Line 179 of Algorithm 11, the AS sets the cnf value to hash(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 ∈ req_{token}.body or auth_req_id ∈ req_{token}.body (see Line 156 and Line 163 of Algorithm 11). An honest client creates requests containing an authorization code or an authorization request identifier only in Line 43 of Algorithm 4 and Line 40 of Algorithm 5. In both cases, it holds true that extractmsg(req_{token}.headers[DPoP])[headers][jwk] ≡ pub(*clientSignKey*), with *clientSignKey* ≡ $S^l(c)$.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* $\notin d_{\emptyset}(S^n(p))$for all processes $p \neq c$.

Request was created by client. As the structured access token contains the value $accessTokenContent$[cnf].1 ≡ 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 ≡ jkt). The corresponding key is taken from the resource request, i.e., the key is extractmsg(m_{req}.headers[DPoP])[headers][jwk] (see Lines 31-33 of Algorithm 18)
All preconditions of Lemma 14 are true, with *dpopProof* ≡ m_{req}.headers[DPoP]:

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

As *clientSignKey* ∉ $d_\emptyset(S^n(p))$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^Q_{out} 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^Q_{out} 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^P_{in}.

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 ≡ 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 14), i.e., the AS added *mtlsInfo* in some processing step $M = (S^m, E^m, N^m) \rightarrow (S^{m\prime}, E^{m\prime}, N^{m\prime})$.
  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 an conclude that there exists a nonce k_{mtls} such that $S^m(as)$.clients[*clientId*][mtls_key] $\equiv \mathsf{pub}(k_{\text{mtls}})$ and for every process $p \neq c$ it holds true that $k_{\text{mtls}} \notin d_{\emptyset}(S^n(p))$.
The structured access token contains the values *accessTokenContent*[cnf].1 $\equiv$ x5t#S256 and *accessTokenContent*[cnf].2 $\equiv \mathsf{hash}(\mathsf{pub}(k_{\text{mtls}}))$. Thus, the resource server executes Line 29 of Algorithm 18 (in the processing step P). This means that e^P_{in} contains a value *mtlsNonce* in the body of the request such that
$\langle \mathit{mtlsNonce}, \mathsf{pub}(k_{\text{mtls}})\rangle \in^{\langle\rangle} S^P(rs)$.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[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 $resp_{\text{introsp}}$ (with the reference 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 $resp_{\text{introsp}}$. The processing step in which the resource server emitted req_{introsp} is P (as in the postcondition of the lemma): The input event of P contains an HTTP request m_{req} (again as in the first postcondition) with an access token $t \equiv m_{\text{req}}$.headers[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 $\langle \mathsf{URL}, \mathsf{S}, \mathit{dom}_{as}, /\mathtt{introspect}, \langle\rangle, \bot\rangle$, with $\mathit{dom}_{as} \in \mathsf{dom}(as)$ (see Definition 15).
Furthermore, req_{introsp} contains the value m_{req}.headers[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 $S^Q(rs).\mathit{ids}$ (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 [jkt: $\mathsf{hash}(k)$] (see Line 179 of Algorithm 11), or [x5t#S256: $\mathsf{hash}(k)$] (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 $resp_{\text{introsp}}$ 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^P_{in}.

□

Lemma 19 (Redirect URI Properties). *For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S, every client identifier clientId that has been issued to c by as in a previous processing step (see Definition 17), 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`] $\equiv$ cliendId implies both rec[`redirect_uri`].`protocol` $\equiv$ `S`, and rec[`redirect_uri`].`host` $\in \mathsf{dom}(c)$*

Proof. Initially, the `authorizationRequests` state subterm of as is empty (see Definition 14). 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`] $\equiv$ *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 **stop**s 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` $\in \mathsf{dom}(c)$ (or extractmsg(*req*.body)[`redirect_uri`].`host` $\in \mathsf{dom}(c)$ in the case of a signed request) and hence rec[`redirect_uri`].`host` $\in \mathsf{dom}(c)$. □

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

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

Proof. Since we have $S^0(c).\mathtt{sessions} \equiv \langle\rangle$ (Definition 13), we know that if $lsid \in S(c).\mathtt{sessions}$, 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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S, and every domain $d \in \mathsf{dom}(as)$, it holds true that if $d \in S(c).\mathtt{oauthConfigCache}$, we have all of the following:*

1) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{issuer}] \equiv d$
2) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{auth\_ep}] \equiv \langle \mathsf{URL}, \mathsf{S}, d, /\mathtt{auth}, \langle\rangle, \bot\rangle$
3) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{token\_ep}] \equiv \langle \mathsf{URL}, \mathsf{S}, d, /\mathtt{token}, \langle\rangle, \bot\rangle$
4) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{par\_ep}] \equiv \langle \mathsf{URL}, \mathsf{S}, d, /\mathtt{par}, \langle\rangle, \bot\rangle$
5) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{introspec\_ep}] \equiv \langle \mathsf{URL}, \mathsf{S}, d, /\mathtt{introspect}, \langle\rangle, \bot\rangle$
6) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{jwks\_uri}] \equiv \langle \mathsf{URL}, \mathsf{S}, d, /\mathtt{jwks}, \langle\rangle, \bot\rangle$
7) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{reg\_ep}] \equiv \langle \mathsf{URL}, \mathsf{S}, d, /\mathtt{reg}, \langle\rangle, \bot\rangle$
8) $S(c).\mathtt{oauthConfigCache}[d][\mathtt{backchannel\_authentication\_endpoint}] \equiv \langle \mathsf{URL}, \mathsf{S}, d, /\mathtt{backchannel\text{-}authn}, \langle\rangle, \bot\rangle$

We note that this implies that all these entries in $S(c).\mathtt{oauthConfigCache}[d]$ 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 $S^0(c).\mathtt{oauthConfigCache} \equiv \langle\rangle$ (Definition 13), 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, $d \in S(c).\mathtt{oauthConfigCache}$ implies that there must have been a processing step $Q = (S^Q, E^Q, N^Q) \to (S^{Q'}, E^{Q'}, N^{Q'})$ in ρ such that $d \notin S^Q(c).\mathtt{oauthConfigCache}$ and $d \in S^{Q'}(c).\mathtt{oauthConfigCache}$. In Q, PROCESS_HTTPS_RESPONSE must have been called with

a *reference* as second argument, such that *reference*[responseTo] ≡ CONFIG. In addition, *reference*[session] must contain a value *sessionId* such that $S^Q(c)$.sessions[*sessionId*][selected_AS] ≡ 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] ≡ 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 36, and Lines 10ff. of Algorithm 41 (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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server as $\in$ AS that is honest in S, every client $c \in$ 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 17), every identity $id \in \mathsf{ID}^{as}$ with b = ownerOfID(id) being an honest browser in S, every authorization code code $\not\equiv \bot$ for which there is a record rec $\in^{\langle\rangle} S(as)$.records with rec[code] ≡ code, rec[client_id] ≡ clientId, and rec[subject] ≡ id and every process $p \notin \{as, c, b\}$, it holds true that code $\notin d_\emptyset(S(p))$.*

Proof.

(1) For *code* to end up in $\big(S(as)$.records.$x\big)$[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 14 – records state subterm are:
- Line 194 of Algorithm 11: This line overwrites the stored authorization code with $\bot$, 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 $\bot$.
- 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 14). 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 authenticateUser), 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 $s_i \to s_{i+1}$ is not known to any process up to and including s_i. Let e_{in} be the event processed by *as* in $s_i \to s_{i+1}$. In order to reach Line 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* ≡ secretOfID(*identity*). Since *as* does not send requests to itself and secretOfID(*identity*) is only known to *as* and ownerOfID(*identity*), *req* must have been created by ownerOfID(*identity*) if the sender of *req* is honest. W.l.o.g., let *identity* ≡ *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`] ≡ *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 ⟨`__Host`, `sessionId`⟩ 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* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of a FAPI Web system* $\mathcal{FAPI}$ *with a network attacker, every configuration* (S^i, E^i, N^i) *in* ρ*, every client* $c \in \mathsf{C}$ *that is honest in* S^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* $S^i(c).\texttt{sessions}[lsid][\texttt{code\_verifier}] \equiv$ *codeVerifier implies:*

(I) $S^j(c).\texttt{sessions}[lsid][\texttt{code\_verifier}] \equiv$ *codeVerifier for all* $j \geq i$*, and*
(II) $S^i(c).\texttt{sessions}[lsid'][\texttt{code\_verifier}] \not\equiv$ *codeVerifier for all* $lsid' \not\equiv lsid$*.*

Proof. We start by noting that an honest client only ever stores something in an entry in `sessions` under key `code_verifier` in Line 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 41, where the value for *reference* is taken from the client's `pendingRequests` state subterm. The `pendingRequests` state subterm is initially empty (Definition 13) and the only place where elements are added to this state subterm is Line 15 of Algorithm 41. 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 36, 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 $reference[\texttt{responseTo}] \equiv \texttt{CONFIG}$.
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 `pendingCIBARequests` 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 `pendingCIBARequests` state subterm is empty (Definition 13) 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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI Web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every client identifier clientId, every authorization server as $\in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ 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 $b \in \mathsf{B}$ that is honest in S, every domain $d_c \in \mathsf{dom}(c)$, every login session id lsid, every nonce codeVerifier with*

(1) $\langle\langle \_\_\mathtt{Host}, \mathtt{sessionId}\rangle, \langle lsid, \top, \top, \top\rangle\rangle \in S(b).\mathtt{cookies}[d_c]$, and
(2) $S(c).\mathtt{sessions}[lsid][\mathtt{code\_verifier}] \equiv codeVerifier$, and
(3) $S(c).\mathtt{sessions}[lsid][\mathtt{selected\_AS}] \in \mathsf{dom}(as)$, and
(4) c does not leak the authorization request for lsid (see Definition 28),

then all of the following hold true:

(I) There is exactly one nonce requestUri, such that

$$S(as).\mathtt{authorizationRequests}[requestUri][\mathtt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$$

(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) $\mathsf{hash}(codeVerifier)$ **does not leak.** We start off by showing that $\mathsf{hash}(codeVerifier)$ does not leak to any process other than c and as. For this, we look at how *codeVerifier* (from (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, $\mathsf{hash}(codeVerifier)$ is only derivable by c.
PREPARE_AND_SEND_INITIAL_REQUEST ends with the client sending a PAR request which contains $\mathsf{hash}(codeVerifier)$ under the key `code_challenge`. So we have to check who receives/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., $\mathsf{hash}(codeVerifier)$ – 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 $\mathtt{hash}(\mathit{codeVerifier})$ 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** $\mathtt{hash}(\mathit{codeVerifier})$**.** 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 80), 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 76). 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 75).
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 $S(c).\mathtt{sessions}[\mathit{lsid}]$ 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 $\mathtt{code\_verifier} \notin S(c).\mathtt{sessions}[\mathit{lsid}]$, 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 $\mathtt{hash}(\mathit{codeVerifier})$ to *as* (see (B)) and got a response (because the $\mathit{reference}[\mathtt{responseTo}]$ 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., $\mathtt{hash}(\mathit{codeVerifier})$ here – in the authorization request storage. Thus, we can conclude that there must be some $\mathit{requestUri}'$ such that

$$S(\mathit{as}).\mathtt{authorizationRequests}[\mathit{requestUri}'][\mathtt{code\_challenge}] \equiv \mathtt{hash}(\mathit{codeVerifier})$$

(D) **Proof for (I).** From (B), we have that only c and *as* know the value $\mathtt{hash}(\mathit{codeVerifier})$ 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 $\mathtt{hash}(\mathit{codeVerifier})$ 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 $S(\mathit{as}).\mathtt{authorizationRequests}[\mathit{requestUri}][\mathtt{code\_challenge}] \equiv \mathtt{hash}(\mathit{codeVerifier})$.

(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 $\langle\langle$`__Host`, `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 $requestUri \notin d_\emptyset(S(p))$.

□

Lemma 25 (CIBA Login Session IDs do not Leak). *For any run* $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ *of a FAPI Web system* $\mathcal{FAPI}$ *with a network attacker, every configuration* (S, E, N) *in* ρ*, every browser* $b \in \mathsf{B}$ *that is honest in* S^n*, every client* $c \in \mathsf{C}$ *that is honest in* S^n*, every domain* $d_c \in \mathsf{dom}(c)$*, every 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* $lsid \notin d_\emptyset(S^n(p))$.

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 80).

Thus, there exists a processing step

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

with $(S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$ 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. $e^Q_{\text{in}} = \langle \_, \_, m \rangle$ and there exists a key $k \in \mathcal{T}_{\mathcal{N}}$ and a term $plaintext \in \mathcal{T}_{\mathcal{N}}$ such that $plaintext \equiv \text{dec}_s(m, key)$, $\pi_1(plaintext) \equiv$ `HTTPResp` (Line 60 of Algorithm 30) and `binding_message` $\in^{\langle\rangle}$ $plaintext$.`body` (Line 65 of Algorithm 30).
The browser stores the values $\langle request.\texttt{host}, plaintext.\texttt{body}[\texttt{binding\_message}] \rangle$ into $S^{Q\prime}(b)$.`cibaBindingMessages`, where *request* is a term stored along with the key in `pendingRequests`, i.e., $\langle \_, request, \_, key, \_ \rangle \in^{\langle\rangle} S^Q(b).\texttt{pendingRequests}$ (see Line 60 of Algorithm 30).
Initially, `pendingRequests` is empty (Definition 80). 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 asymetrically with the key $s^b_0.\texttt{keyMapping}[d_c]$ (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

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

with $(S^{P\prime}, E^{P\prime}, N^{P\prime})$ prior to $(S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$. 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^Q_{in}) in Line 43 of Algorithm 2.
Before emitting the response, the client stores the binding message into $S^{P\prime}(c).\texttt{sessions}[lsid][\texttt{binding\_message}]$ (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 $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every client $c \in \mathsf{C}$ 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. By contradiction: assume $S(c)$.sessions[*lsid*][cibaFlow] = $\top$ and code_verifier $\in$ $S(c)$.sessions[*lsid*].

Initially, the sessions state subterm of c is empty (Definition 13). 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 $Q = (S^Q, E^Q, N^Q) \rightarrow (S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$ 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* $\mathcal{FAPI}$ *with a network attacker, every configuration* (S, E, N) *in* ρ*, every client* $c \in \mathsf{C}$ *that is honest in S, every term lsid* $\in \mathcal{T}_{\mathcal{N}}$*, it holds true that if for any cibaKey* $\in$ {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 13). 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 Uniquenuess for Access Token). *For any run* ρ *of a FAPI Web system* $\mathcal{FAPI}$ *with a network attacker, every configuration* (S, E, N) *in* ρ*, every authorization server*

as ∈ AS *that is honest in S, every values i, j* ∈ ℕ*, it holds true that if*

$$S(as).\texttt{records}.i.2[\texttt{access\_token}] \equiv S(as).\texttt{records}.j.2[\texttt{access\_token}]$$

then $S(as).\texttt{records}.i.2[\texttt{subject}] \equiv S(as).\texttt{records}.j.2[\texttt{subject}]$.

Proof. Initially, the records state subterm of *as* is empty (Definition 14). 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
$S(as).\texttt{records}.i.2[\texttt{access\_token}] \equiv S(as).\texttt{records}.j.2[\texttt{access\_token}]$, 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* $\mathcal{FAPI}$ *with a network attacker, every configuration* (S, E, N) *in* ρ*, every client* $c \in$ C *that is honest in S, every authorization server as* ∈ 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) $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}] \in \texttt{dom}(as)$*, and*
(b) $S(c).\texttt{sessions}[lsid][\texttt{auth\_req\_id}] \equiv authnReqId$*, and*
(c) $S(as).\texttt{records}[authnReqId][\texttt{binding\_message}] \equiv bindingMsg$

then it holds true that $S(c).\texttt{sessions}[lsid][\texttt{binding\_message}] \equiv bindingMsg$.

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 $P = (S^P, E^P, N^P) \rightarrow (S^{P\prime}, E^{P\prime}, N^{P\prime})$ 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., $S^P(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$ (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] ≡ ⟨URL, S, *selectedAS*, /backchannel-authn, ⟨⟩, ⊥⟩ (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* ∈ dom(*as*) (Precondition (a)), i.e., the only *as* can decrypt the HTTPS request.

Let $Q = (S^Q, E^Q, N^Q) \to (S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$ 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 $S^{Q\prime}(as)$.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 FAPI with a network attacker, every configuration* (S, E, N) *in ρ, every client* $c \in \mathsf{C}$ *that is honest in S, every AS* $as \in \mathsf{AS}$ *that is honest in S, every domain* $d \in \mathrm{dom}(as)$*, if* $S(c)$.asAccounts[d][client_id] ≢ ⟨⟩*, 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 17).*

Proof. Initially, asAccounts is empty (see Definition 13), 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 $R = (S^r, E^r, N^r) \to (S^{r\prime}, E^{r\prime}, N^{r\prime})$ in which the client executes Line 49 of Algorithm 3 and sets $S^{r\prime}(c)$.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 $P = (S^p, E^p, N^p) \to (S^{p\prime}, E^{p\prime}, N^{p\prime})$, 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 $S^{r\prime}(c)$.asAccounts[d], it follows that $S^r(c)$.sessions[*sessionId*][selected_AS] ≡ *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] ≡ *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 17 are fulfilled, and in particular, *Q* is the processing step in which the AS issued the client identifier.

□

F.2 Authorization Property

In this section, we show that the authorization property from Definition 19 holds.

Lemma 31 (Authorization). *For*

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

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

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

it holds true that:

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

(I) There exists a processing step

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

such that

(1) either $P = Q$ *or* P *prior to* Q *in* ρ*, and*

(2) e^P_{in} *is an event* $\langle x, y, \mathsf{enc_a}(\langle m_{req}, k_1\rangle, k_2)\rangle$ *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.,* $t \equiv m_{req}.\mathtt{headers}[\mathtt{Authorization}].2$*, 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 1).*

(III) If there exists a client $c \in \mathsf{C}$ *such that* $clientId$ *has been issued to* c *by* as *in a previous processing step (see Definition 17), and if* c *is honest in* S^n*, then* r *is not derivable from the attackers knowledge in* S^n *(i.e.,* $r \notin d_\emptyset(S^n(\mathsf{attacker}))$*).*

Proof. **Resource server sends resource to correct client.** The first and the second postcondition are shown in Lemma 18, where we also showed that the message contained in the event e^P_{in} was created by c (as intuitively, the access token is bound to c via mTLS or DPoP, and no other process can prove possession of the secret key to which the token is bound). The resource r is sent back as a response to e^P_{in}: If the resource server sends out the resource in Line 84 of Algorithm 18, then it encrypts the HTTPS response (symmetrically) with the key contained in e^P_{in}. Otherwise, the resource server sends out the response in Line 45 of Algorithm 19, encrypted (symmetrically) with the key contained in e^P_{in} (the resource server stored the key in its state).

Thus, the resource server sends out the resource r back to c, encrypted with a symmetric key that only c and rs can derive. This response can only be decrypted by c: A resource server can decrypt symmetrically only in Line 19 of Algorithm 41 (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 41).

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 $R = (S^R, E^R, N^R) \xrightarrow[c \to E^R_{\text{out}}]{e^R_{\text{in}} \to c} (S^{R\prime}, E^{R\prime}, N^{R\prime})$ (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., $S^R(c).\texttt{sessions}[\textit{sessionId}][\texttt{cibaFlow}]$ is either $\top$ or $\bot$. As shown in Lemma 20, this value is not changed by the client after initially choosing it.

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

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 $\bot$ 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 req_{redir} stored in $S^R.\texttt{sessions}[\textit{sessionId}][\texttt{redirectEpRequest}]$, for some value *sessionId* (and in particular, encrypts the response with the key contained in req_{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., req_{redir} is a request that was previously received by the client. This request contains a value (an authorization code) in $req_{\text{redir}}.\texttt{parameters}[\texttt{code}]$ (or $\text{extractmsg}(req_{\text{redir}}.\texttt{parameters}[\texttt{response}])[\texttt{code}]$ if the authorization response is signed), which the client puts into the token request in Algorithm 4..

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 $\text{authorizationServerOfResource}^{rs}(\textit{resourceID})$ (see Definition 13). 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 $T = (S^T, E^T, N^T) \xrightarrow[as \to E^T_{\text{out}}]{e^T_{\text{in}} \to as} (S^{T\prime}, E^{T\prime}, N^{T\prime})$. 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 $rec \in^{\langle\rangle} S^T(as).\texttt{records}$ with $rec[\texttt{code}] \equiv code$ and $code \not\equiv \bot$ (Line 157 of Algorithm 11). Furthermore, the record has the following values:

- $rec[\texttt{clientId}] \equiv clientId$ (as the access token is bound to this client id),
- $rec[\texttt{subject}] \equiv id$ (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 $S^R(c).\texttt{sessions}[sessionId][\texttt{cibaFlow}] \equiv \top$.

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 $S^{R\prime}(c).\texttt{sessions}[sessionId][\texttt{resource}]$ (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 $S^{R\prime}$.

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 $reference[\texttt{responseTo}] \equiv \texttt{RESOURCE\_USAGE}$ (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 13). 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 a parameter of Algorithm 3 (PROCESS_HTTPS_RESPONSE).

As the client calls Line 83 of Algorithm 3, it follows that *reference'*[responseTo] ≡ 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 $T = (S^T, E^T, N^T) \xrightarrow[c \to E^T_{\text{out}}]{e^T_{\text{in}} \to c} (S^{T\prime}, E^{T\prime}, N^{T\prime})$ 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, $S^T(c)$.sessions[*sessionId*][code_verifier] is empty, as a client never changes the cibaFlow value of a session and we here have $S^R(c)$.sessions[*sessionId*][cibaFlow] ≡ ⊤.

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 an authentication request identifier *authnReqId* value and the grant_type urn:openid:params:grant-type:ciba in the body of the request (Lines 6 and 39 of Algorithm 5). Note that the client retrieves the request id from the session identified by *sessionId*, i.e., $S^T(c)$.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 of *as*. It holds true that $d_c \equiv S^T(c)$.session[*lsid*][selected_AS], and therefore,

$$S^T(c).\texttt{session}[lsid][\texttt{selected\_AS}] \in \mathsf{dom}(as) \tag{6}$$

because the client takes the domain used for the token request from $S^T(c)$.oauthConfigCache[*selectedAS*][token_ep], with *selectedAS* ≡ $S^T(c)$.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] ≡ *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'* $\in^{\langle\rangle}$ $S^Q(as)$.records such that *rec'*[access_token] ≡ *t* and *rec'*[subject] ≡ *id*. From Lemma 28, it follows that *rec* and *rec'* have the same subject entry, i.e., *rec*[subject] ≡ *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-auth2-req*.body[identity] ≡ *id* (see Line 282 and Line 294 of Algorithm 11).

This request also contains the password of *id*, i.e., *ciba-auth2-req*.body[password] ≡ secretOfID(*identity*) (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* ≡ *ciba-auth2-req*.body[ciba_auth2_reference] (Line 288 of Algorithm 11) such that *rec*[ciba_auth2_reference] ≡ *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*[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 $S(c)$.sessions[*sessionId*][selected_AS] ∈ dom(*as*). From Lemma 29, it follow that $S(c)$.sessions[*sessionId*][binding_message] ≡ *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] ≡ *bindingMsg*. However, this contradicts Lemma 25, i.e., the attacker cannot derive *sessionId*.

□

F.3 Authentication Property

In this section, we show that the authentication property from Definition 21 holds. This will be a proof by contradiction, i.e., we assume that there is a FAPI Web system $\mathcal{FAPI}$ in which the authentication property is violated and deduce a contradiction.

Assumption 2. *There exists a FAPI Web system with a network attacker $\mathcal{FAPI}$ such that there exists a run ρ of $\mathcal{FAPI}$ with a configuration (S, E, N) in ρ, some $c \in \mathsf{C}$ that is honest in S, some identity $id \in \mathsf{ID}$ with $as = \mathsf{governor}(id)$ being an honest AS and $b = \mathsf{secretOfID}(id)$ being browser honest in S, some service session identified by some nonce n for id at c, and n is derivable from the attacker's knowledge in S (i.e., $n \in d_\emptyset(S(\mathsf{attacker}))$).*

Lemma 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 4, we have that there is a session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id)) = \mathsf{dom}(as)$

with $S(c).\texttt{sessions}[x][\texttt{loggedInAs}] \equiv \langle d, id \rangle$ and $S(c).\texttt{sessions}[x][\texttt{serviceSessionId}] \equiv n$. Assumption 2 says that n is derivable from the attacker's knowledge. Since we have $S(c).\texttt{sessions}[x][\texttt{serviceSessionId}] \equiv n$, we can check where such an entry in c's state can be created.

The only place in which an honest client stores a service session id is in the function CHECK_ID_TOKEN, specifically in Line 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 $S(c).\texttt{sessions}[x][\texttt{loggedInAs}]$, 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 $S(c).\texttt{sessions}[x][\texttt{selected\_AS}] \equiv d$. In Line 85 of Algorithm 3, the client checks whether the host of the corresponding token request is equal to $S(c).\texttt{oauthConfigCache}[selectedAS][\texttt{token\_ep}].\texttt{host}$, with $selectedAS \equiv S(c).\texttt{sessions}[x][\texttt{selected\_AS}]$ (see Lines 32-34 and Line 84 of Algorithm 3). As shown in Lemma 21, $S(c).\texttt{oauthConfigCache}[selectedAS][\texttt{token\_ep}].\texttt{host} \equiv selectedAS$, i.e., the client sent the token request to as. Let req_{token} be the token request, and $R = (S^r, E^r, N^r) \rightarrow (S^{r\prime}, E^{r\prime}, N^{r\prime})$ 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 17): 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., $req_{\text{token}}.\texttt{body}[\texttt{client\_id}]$ (if the client uses mTLS client authentication) or $\text{extractmsg}(req_{\text{token}}.\texttt{body}[\texttt{client\_assertion}])[\texttt{iss}]$ (if the client uses `private_key_jw` client authentication) to the value $S^r(c).\texttt{asAccounts}[selectedAS][\texttt{client\_id}]$ 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 $clientId \not\equiv \langle\rangle$, 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 $req_{\text{token}}.\texttt{body}[\texttt{grant\_type}] \equiv \texttt{authorization\_code}$, then the token request contains a $code$ such that there is a record $rec \in^{\langle\rangle} S(as).\texttt{records}$ with $rec[\texttt{code}] \equiv code$, $rec[\texttt{client\_id}] \equiv clientId$, and $rec[\texttt{subject}] \equiv id$.

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 $\bot$, 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 $\bot$.

As the `cibaFlow` value of this session is $\bot$, 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 $\not\equiv \bot$, 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] $\equiv$ 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 $req_{\text{ciba-auth2}}$ at its /ciba-auth2 endpoint (Line 281 of Algorithm 11) with $req_{\text{ciba-auth2}}$.body[identity] $\equiv$ id (see Line 282

and Line 294 of Algorithm 11). This request also contains the password of *id*, i.e., $req_{\text{ciba-auth2}}$.body[password] ≡ 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* ≡ $req_{\text{ciba-auth2}}$.body[ciba_auth2_reference] (Line 288 of Algorithm 11) such that *rec*[ciba_auth2_reference] ≡ *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] ∈ dom(*as*). It also holds true that $S(c)$.sessions[x][auth_req_id] ≡ *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] ≡ *bindingMsg*. From Lemma 29, it follow that $S(c)$.sessions[x][binding_message] ≡ *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 $\langle d_c, \textit{bindingMsg} \rangle \in^{\langle\rangle} S(b)$.cibaBindingMessages (note that the browser does not remove or modify existing cibaBindingMessages values). As noted

previously, it holds true that $S(c).\texttt{sessions}[x][\texttt{binding\_message}] \equiv \mathit{bindingMsg}$. 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 $req_{\text{get-ssid}}$ with a session id cookie, and responds with the corresponding service session id. Thus, it follows that $req_{\text{get-ssid}}$ 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.

□

F.4 Session Integrity for Authentication Property

In this section, we show that the session integrity for authentication properties from Definition 29 and Definition 30 hold.

We start by proving that the property for authorization code flows (Definition 29) 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 $\mathcal{FAPI}$ such that there exists a run ρ of $\mathcal{FAPI}$ with a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ, a browser b honest in S, an authorization server $as \in \mathsf{AS}$, an identity id, a client $c \in \mathsf{C}$ honest in S, and a nonce $lsid$ s.t. $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \bot$, with $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$ 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 $\mathsf{started}_\rho^{Q'}(b, c, lsid)$, or
(II) as is honest in S, and there is no processing step Q'' prior to Q in ρ such that $\mathsf{authenticated}_\rho^{Q''}(b, c, id, as, lsid)$.

Lemma 33 (Sess. Integ. for Authentication for Authorization Code Flows Prop. Holds). *Assumption 3 is a contradiction.*

Proof.

(I) We have that $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$. With Definition 22, we know that c sent out a service session id associated with $lsid$ to b (i.e., set a cookie $\langle \texttt{serviceSessionId}, \langle ssid, \top, \top, \top \rangle \rangle$, and stored $ssid$ in its sessions storage). For a session with $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \bot$, 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 $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$). 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, $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$, 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 $[\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle : lsid]$ (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 22), 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 $S(c).\texttt{sessions}[lsid]$ under the key `redirectEpRequest`. Hence, the sender of req_{redir}, i.e., b by Definition 22, 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 $\langle \texttt{Set-Cookie}, [\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle : \langle lsid, \top, \top, \top\rangle]\rangle$.
Hence, we have that $\mathsf{started}_{\rho}^{Q'}(b, c, lsid)$.

(II) Again, we have $\mathsf{loggedIn}_{\rho}^{Q}(b, c, id, as, lsid)$ and we know that c sent out a service session id associated with *lsid* to b. This can only happen in the client's function CHECK_ID_TOKEN, which only produces an output if c received an id token t (via a token response). From $S(c).\texttt{sessions}[lsid][\texttt{loggedInAs}] \equiv \langle d, id\rangle$, we know – since CHECK_ID_TOKEN produced an output – that for $t_c := \mathsf{extractmsg}(t)$, we have $t_c[\texttt{iss}] \equiv d$, $t_c[\texttt{sub}] \equiv id$, and $t_c[\texttt{aud}] \equiv clientId$ (for some *clientId*). Due to the check in Line 85 of Algorithm 3, this id token must have been sent by *as* (because $d \in \mathsf{dom}(as)$). *as* will only output such a term t if there is a record *rec* in *as*'s `records` state subterm with $rec[\texttt{subject}] \equiv id$, $rec[\texttt{client\_id}] \equiv clientId$, and $rec[\texttt{code\_challenge}] \equiv codeChallenge$ (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 17).
By construction of c and tracking of $\texttt{sessions}[lsid]$ in c's state, it is easy to see that once c reaches CHECK_ID_TOKEN, the session storage $S(c).\texttt{sessions}[lsid]$ must contain a key `code_verifier` under which a nonce *codeVerifier* is stored. We note that $S(b).\texttt{cookies}[d_c]$ must contain a cookie $\langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle lsid, \top, \top, \top\rangle\rangle$ for $d_c \in \mathsf{dom}(c)$, because b sends a cookie $[\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle : lsid]$ as explained above, b is honest (and will thus

not accept __Host headers for d_c from parties other than c), and if c sets a cookie $\langle$__Host, sessionId$\rangle$, it will do so with the attributes set as shown here.
Hence, we can apply Lemma 24 (note that $S(c).\texttt{sessions}[lsid][\texttt{loggedInAs}] \equiv \langle d, id\rangle$ with $d \in \mathsf{dom}(as)$ implies $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}] \equiv d \in \mathsf{dom}(as)$). I.e., we now have that there is exactly one nonce *requestUri* such that $S(as).\texttt{authorizationRequests}[requestUri][\texttt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$, and only b, c, and as know *requestUri*.
We know from Line 158 of Algorithm 11 that the token request which leads to as issuing t must contain a code verifier such that $\mathsf{hash}(codeVerifier) \equiv rec[\texttt{code\_challenge}]$ (with *rec* from above). Since we know that c must have sent the token request (otherwise, c would not have received t), we can track where and how c creates such a request. This is only the case in function SEND_TOKEN_REQUEST (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 $authZrec[\texttt{auth2\_reference}] \equiv auth2Reference$ and that *auth2Reference* is contained in the request to as's /auth2 endpoint. Such an *auth2Reference*, in turn, is only created at as's /auth endpoint. For a request to this endpoint to lead to storing *auth2Reference*, the request must contain *reqUri* under the key request_uri.
Note that by Lemma 24, we established that there is exactly one *requestUri* in as's state such that $S(as).\texttt{authorizationRequests}[requestUri][\texttt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$. Therefore, $reqUri \equiv requestUri$. In addition, it is easy to see that c and as do not send any requests to as's /auth endpoint. Hence, b must have sent a request with *reqUri* to /auth. Since *auth2Reference* from above is only sent to whoever sent the first request to /auth (and – if b receives it – b does not leak that value) we know that b must have sent the POST request to /auth2 as well. As b is honest, this can only happen trough 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 $rec[\texttt{subject}] \equiv id$ and such a *rec* is only stored at /auth2 endpoint if the identity in the request is equivalent to *id*) and sent a request to as's /auth2 endpoint containing *auth2Reference* – hence, the *scriptstate* contained a key auth2_reference with value *auth2Reference*.
Therefore, we have $\mathsf{authenticated}_\rho^{Q''}(b, c, id, as, lsid)$ which – together with (I) from above – contradicts Assumption 3, therefore proving the lemma.

□

This leaves us with the property for FAPI-CIBA flows (Definition 30). 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 $\mathcal{FAPI}$ such that there exists a run ρ of $\mathcal{FAPI}$ 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 $\in \mathsf{AS}$*, an identity id, a client* $c \in \mathsf{C}$ *honest in S, and a nonce lsid s.t.* $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$*, and* $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$ *such that*

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

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 $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$, we know that during Q, c emits an event which contains an HTTPS response with a header $\langle$`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 $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \bot$ (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 $S(c).\texttt{sessions}[lsid][\texttt{serviceSessionId}]$ – 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 $S(c).\texttt{sessions}[lsid][\texttt{serviceSessionId}] \equiv S(c).\texttt{sessions}[lsid'][\texttt{serviceSessionId}]$.
(A.ii) $\texttt{serviceSessionId} \in S(c).\texttt{sessions}[lsid]$: 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 $S^0(b).\texttt{cookies} \equiv \langle\rangle$ (Definition 9), 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 $reference[\texttt{responseTo}] \equiv \texttt{PAR}$. 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 $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \bot$, 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 $b = \mathsf{ownerOfID}(S(c).\texttt{sessions}[lsid][\texttt{selected\_identity}])$ (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 41, Definition 13, and Appendix C.3: 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 41). 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 $\mathsf{startedCIBA}_\rho^{Q'}(b, c, lsid)$.

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

(E.i) From (A.ii) we have $\texttt{serviceSessionId} \in S(c).\texttt{sessions}[lsid]$. 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 $S(c).\texttt{sessions}[lsid][\texttt{serviceSessionId}]$ was set to the value *ssid* used in Q (i.e., in loggedIn, see Definition 22). I.e., c must have reached Line 14 of Algorithm 7 during P, and – since we have $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$ 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 $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$ (see Definition 22), 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 $idToken[\texttt{sub}] \equiv id$.

(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., $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$) 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 $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$ (see Definition 22), 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 $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$ 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., $\texttt{code\_verifier} \in S^{\text{tokReq}}(c).\texttt{sessions}[lsid]$. However, we have

$S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$, which together with Lemma 20 gives us $S^{\text{tokReq}}(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$ – 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 $authnReqId := S^{\text{tokReq}}(c).\texttt{sessions}[lsid][\texttt{auth\_req\_id}]$. Note: The session id used in P_c^{tokReq}, i.e., $lsid$, is the same as in $\text{loggedIn}_\rho^Q(b, c, id, as, lsid)$: 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 $P_{as}^{\text{tokRes}} = (S^{\text{tokRes}}, E^{\text{tokRes}}, N^{\text{tokRes}}) \to (S^{\text{tokRes}'}, E^{\text{tokRes}'}, N^{\text{tokRes}'})$. 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 $S^{\text{tokRes}}(as).\texttt{records}$ such that $rec[\texttt{subject}] \equiv id$ (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 $rec[\texttt{auth\_req\_id}] \notin \{\langle\rangle, \bot\}$ (Lines 163 and 164 of Algorithm 11. Furthermore, $rec[\texttt{auth\_req\_id}]$ must be equivalent to the $authnReqId$ in the token request from (E.vi).

An AS only adds a value $\neq \bot$ 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 $P_{as}^{\text{auth2}} = (S^{\text{auth2}}, E^{\text{auth2}}, N^{\text{auth2}}) \to (S^{\text{auth2}'}, E^{\text{auth2}'}, N^{\text{auth2}'})$ 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., $S^{\text{auth2}}(as).\texttt{cibaAuthnRequests}[authnReqId'][\texttt{selected\_identity}] \equiv id'$ (Line 290 of Algorithm 11).

(E.ix.3) The contents of $S^{\text{auth2}}(as).\texttt{cibaAuthnRequests}[authnReqId']$, 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: $\langle\texttt{subject}, id'\rangle$, and $\langle\texttt{issuer}, d_{as}\rangle$ (for a $d_{as} \in \text{dom}(as)$), and $\langle\texttt{auth\_req\_id}, authnReqId'\rangle$. Since we are looking at the record rec from (E.vii) and (E.viii), we must have $authnReqId' \equiv authnReqId$ 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 $authnReqId = S^{\text{tokReq}}(c).\texttt{sessions}[lsid][\texttt{auth\_req\_id}]$, 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., $S^{\text{tokReq}}(c).\texttt{sessions}[lsid][\texttt{selected\_identity}]$ (note that the client never overwrites this value after initially setting it).
Overall, this gives us

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

However, from (C), we have that *b* is the owner of identity $S(c).\texttt{sessions}[lsid][\texttt{selected\_identity}]$, 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 *authnReqId*′ with $S^{\text{auth2}}(as).\texttt{cibaAuthnRequests}[authnReqId'][\texttt{ciba\_auth2\_reference}] \equiv auth2Reference$ (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 $script_a s_c iba_f orm$.
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 $script_a s_c iba_f orm$. 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 $script_a s_c iba_f orm$ 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 $S^{\text{auth2}}(as)$.cibaAuthnRequests[$authnReqId'$][ciba_auth2_reference] ≡ *auth2Reference* (see above). Since as never changes that value, and with (E.ix.3) we get $S(as)$.cibaAuthnRequests[$authnReqId$][ciba_auth2_reference] ≡ *auth2Reference*.
(E.ix.6) additionally gives us $authnReqId = S^{\text{tokReq}}(c)$.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 $\mathsf{authenticatedCIBA}_\rho^{Q''}(b, c, id, as, lsid)$.

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

□

F.5 Session Integrity for Authorization Property

In this section, we show that the session integrity properties for authorization from Definition 31 and Definition 32 hold.

We start by proving that the property for authorization code flows (Definition 31) holds.

Lemma 35 (Session Integrity for Authorization Property Holds). *For every run ρ of FAPI, every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ, every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client $c \in \mathsf{C}$ that is honest in S, every $rs \in \mathsf{RS}$ that is honest in S, every nonce r, every nonce lsid such that $S(c)$.sessions[$lsid$][cibaFlow] ≡ ⊥, we have that if $\mathsf{accessesResource}_\rho^{\mathrm{Q}}(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for lsid (see Definition 28), then (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, c, u, as, lsid)$.*

Proof. **(1).** Due to $\mathsf{accessesResource}_\rho^{\mathrm{Q}}(b, r, u, c, rs, as, lsid)$ (Definition 27), 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] ≡ ⊥. 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). $\mathsf{accessesResource}_\rho^{Q}(b, r, u, c, rs, as, lsid)$ 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] ≡ ⊥.

Client received resource from *rs.* As the client executes Line 114 of Algorithm 3 during Q, and as $S'(c)$.sessions[$lsid$][resourceServer] ∈ dom(rs) (see accessesResource) is only set in Line 110 of Algorithm 3, it follows that c received the resource r in a response $resp_{\text{resource}}$ 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 $resp_{\text{resource}}$ with $\texttt{resource} \in resp_{\text{resource}}.\texttt{body}$ either in Line 84 of Algorithm 18 or Line 45 of Algorithm 19. Let $P_{rs}^{\text{resResp}} = (S^{\text{resResp}}, E^{\text{resResp}}, N^{\text{resResp}}) \rightarrow (S^{\text{resResp}'}, E^{\text{resResp}'}, N^{\text{resResp}'})$ be the processing step in which rs sends $resp_{\text{resource}}$.
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 $P_{rs}^{\text{resReq}} = (S^{\text{resReq}}, E^{\text{resReq}}, N^{\text{resReq}}) \rightarrow (S^{\text{resReq}'}, E^{\text{resReq}'}, N^{\text{resReq}'})$ be the processing step in which rs receives req_{resource}).
Furthermore, as the RS stored the resource in $S(rs).\texttt{resourceNonces}[u][resourceId]$ (see accessesResource, for some $resourceId \in \mathcal{T}_{\mathcal{N}}$), it follows that $req_{\text{resource}}.\texttt{path} \equiv resourceId$ (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 RS in Line 16 of Algorithm 18 during $P_{rs}^{\text{resReq}}$is a domain of as (the resource server never changes the `resourceASMapping` subterm of its state, see also Definition 15, and from accessesResource, we have $as = \text{authorizationServerOfResource}^{rs}(resourceID)$).
We now look at the two places in which rs could have produced $resp_{\text{resource}}$ 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 $rec[\texttt{access\_token}] \equiv t$ and $rec[\texttt{subject}] \equiv u$.

Token request was sent to as. An honest client sends resource requests only in Algorithm 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 $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}]$ 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 $\texttt{auth\_req\_id} \in S(c).\texttt{sessions}[lsid]$ – with Lemma 27, this gives us $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$, 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 AS 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 AS *as* exchanged the authorization code *code* at the token endpoint and the issued access token is associated with the identity u, it follows that identity u logged in at the /auth2 endpoint of *as*, and the request to /auth2 contained a value *auth2reference* in its body equal to $S''(as).\texttt{authorizationRequests}[requestUri][\texttt{auth2\_reference}]$ (with S'' being the state of a configuration prior to Q; see also Line 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 /auth request, and received *auth2reference* in the response.
As $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}]$ 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 /auth endpoint, it follows that the request to the /auth endpoint of *as* was sent by b. The remaining argumentation is the same as for the proof of Lemma 33.

□

This leaves us with the property for FAPI-CIBA flows (Definition 32). 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 $\mathcal{FAPI}$ such that there exists a run ρ of $\mathcal{FAPI}$ with a processing step $Q = (S, E, N) \to (S', E', N')$ in ρ, a browser b that is honest in S and behaves according to Assumption 1, an authorization server $as \in \mathsf{AS}$, an identity id, a client $c \in \mathsf{C}$ honest in S, a resource server $rs \in \mathsf{RS}$ that is honest in S, a nonce r, and a nonce lsid s.t. $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$, and $\mathsf{accessesResource}^{Q}_{\rho}(b, r, id, c, rs, as, lsid)$ such that*

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

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 $\mathsf{accessesResource}^{Q}_{\rho}(b, r, id, c, rs, as, lsid)$, we know that during Q, c emits an event which contains an HTTPS response whose body is r. Furthermore, due to precondition $S(c).\texttt{sessions}[lsid][\texttt{cibaFlow}] \equiv \top$, c must have executed Line 68 of Algorithm 2 during Q, which implies $S(c).\texttt{sessions}[lsid][\texttt{resource}] \equiv r$ (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 $\langle \texttt{\_\_Host}, \texttt{sessionid} \rangle$ with value *lsid* (Line 52 of Algorithm 2).

(C) We have that $\mathsf{accessesResource}^{Q}_{\rho}(b, r, id, c, rs, as, lsid)$. Therefore, we also have the following: $\langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle lsid, y, z, z'\rangle\rangle \in^{\langle\rangle} S'(b).\texttt{cookies}[d]$ for some $d \in \mathsf{dom}(c)$. 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 $res_{\text{start-ciba}}$) 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 $S'(b).\texttt{cookies}[d]$ 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 $b = \mathsf{ownerOfID}(S(c).\texttt{sessions}[lsid][\texttt{selected\_identity}])$.

(E) With (C) and (D), we can apply the same argumentation as in (D) in the proof of Lemma 34, and conclude $\mathsf{startedCIBA}^{Q'}_{\rho}(b, c, lsid)$.

(F) From (A), we have $S(c).\texttt{sessions}[lsid][\texttt{resource}] \equiv r$. 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 $P_c^{\text{storeR}} = (S^{\text{storeR}}, E^{\text{storeR}}, N^{\text{storeR}}) \rightarrow (S^{\text{storeR}'}, E^{\text{storeR}'}, N^{\text{storeR}'})$, hence $res_{\text{resource}}.\texttt{body}[\texttt{resource}] \equiv r$. 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 41).
In that same processing step P_c^{storeR}, c also executes Line 110 of Algorithm 3, i.e., sets a value for $S(c).\texttt{sessions}[lsid][\texttt{resourceServer}]$. 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 $\mathsf{accessesResource}^{Q}_{\rho}(b, r, id, c, rs, as, lsid)$, we have $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ and $r \in^{\langle\rangle} S'(rs).\texttt{resourceNonces}[id][resourceId]$ (for some value $resourceId \in \mathcal{T}_{\mathcal{N}}$).

(H) For $r \in^{\langle\rangle} S'(rs).\texttt{resourceNonces}[id][resourceId]$ 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 15). 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 $id \not\equiv id' \vee resourceId \not\equiv resourceId'$ such that $r \in^{\langle\rangle} S'(rs).\texttt{resourceNonces}[id'][resourceId']$, 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 $verifKey := S^{i_P}(rs).\texttt{asInfo}[d_{as}][\texttt{as\_key}]$ (see Line 19 of Algorithm 18), where $d_{as} = S^{i_P}(rs).\texttt{resourceASMapping}[resourceId]$. Since `resourceASMapping` is never changed, initialized using authorizationServerOfResource (see Definition 15), and from (G) we have $as = \text{authorizationServerOfResource}^{rs}(resourceID)$, this implies $d_{as} \in \text{dom}(as)$. Hence, by Definition 15, 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 $rec[\texttt{subject}] \equiv id$. 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 $as = \text{authorizationServerOfResource}^{rs}(resourceID)$), 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 $res_{\text{introspect}}.\texttt{body}[\texttt{sub}] \equiv id$ (see (I) and Line 27 of Algorithm 19), and $\texttt{active} \in res_{\text{introspect}}.\texttt{body}$. 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 $rec[\texttt{subject}] \equiv id$ (Line 225 of Algorithm 11). Furthermore, we must have $\texttt{access\_token} \in rec$. 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) Thus, we have $\text{loggedIn}_{\rho}^{Q}(b, c, id, as, lsid)$ (see (E)) and $\text{authenticatedCIBA}_{\rho}^{Q''}(b, c, id, as, lsid)$, which gives us a contradiction to Assumption 5.

□

F.6 Non-Repudiation Properties

In this section, we show that the non-repudiation properties hold.

Lemma 37 (Non-repudiation for Signed Authorization Requests (Definition 33) Holds). *For every run $\rho = ((S^0, E^0, N^0), \ldots)$ of FAPI, every configuration (S^n, E^n, N^n) in ρ, every process $as \in \mathsf{AS}$ that is honest in S^n, every request uri requestUri, we have that if $S^n(as).\texttt{authorizationRequests}[requestUri][\texttt{signed\_par}] \equiv \top$, then all of the following hold true:*

(I) There exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \to as} (S', E', N')$ with (S, E, N) prior to (S^n, E^n, N^n) in ρ, such that $requestUri \notin S(as).\texttt{authorizationRequests}$ and $requestUri \in S'(as).\texttt{authorizationRequests}$.

(II) $e_{in} = \langle x, y, m \rangle$ contains a message m of the form $\mathsf{enc_a}(\langle\langle \mathsf{HTTPReq}, \cdot, \mathsf{POST}, selectedAS, /\mathsf{par}, \cdot, \langle\rangle, body\rangle, \cdot\rangle, \cdot)$, where $body$ is of the form $\mathsf{sig}(par, signKey)$ and $selectedAS \in \mathsf{dom}(as)$.

(III) If there is a process $c \in \mathsf{C}$ which is honest in S^n, and a configuration (S^i, E^i, N^i) in ρ with $S^i(c).\texttt{asAccounts}[selectedAS][\texttt{sign\_key}] \equiv signKey$, then there is a processing step $P = (S^j, E^j, N^j) \to (S^{j+1}, E^{j+1}, N^{j+1})$ 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 `authorizationRequests` state subterm is empty (Definition 14). Hence, we have $requestUri \notin S^0(as).\texttt{authorizationRequests}$. By induction, we get that there must be some processing step Q in ρ, during which *requestUri* is added to the `authorizationRequests` state subterm.

(II) The only place in which an honest AS adds a new record to its `authorizationRequests` state subterm is Line 142 of Algorithm 11.[12] This line is only executed when processing an HTTPS request, i.e., a message of the form $\mathsf{enc_a}(\langle \mathsf{HTTPReq}, \cdot, method, host, path, \cdot, \langle\rangle, body\rangle, \cdot)$ (see Lines 7ff. of Algorithm 41). Due to Line 103 of Algorithm 11, we know $path \equiv /\mathsf{par}$ and $method \equiv \mathsf{POST}$. Furthermore, Lemma 2 gives us $host \in \mathsf{dom}(as)$. We can also infer that *body* must have the form $\mathsf{sig}(par, signKey)$ from Lines 106 and 107 of Algorithm 11, since we know that $requireSignedPAR \equiv \top$ 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 $c \in \mathsf{C}$, honest in S^n, and a configuration (S^i, E^i, N^i) such that $S^i(c).\texttt{asAccounts}[host][\texttt{sign\_key}] \equiv signKey$. 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 $\mathsf{sig}(par, signKey)$ (see the equational theory in Figure 10).

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`.

Line 26 of Algorithm 6 The signed value is a dictionary, but does not contain a key `client_id`.

[12] Note that Line 73 of Algorithm 11 does not add a new record, but extends an existing one, see Line 67 of Algorithm 11.

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 $\mathrm{sig}(par, signKey)$ cannot be derivable by any process prior to $P^{\mathrm{createPAR}}$.
Hence, we conclude that c must have signed *par* during some processing step $P^{\mathrm{createPAR}}$ in Line 63 of Algorithm 8.

□

Lemma 38 (Non-repudiation for Signed Authorization Responses (Definition 34) Holds). *For every run $\rho = ((S^0, E^0, N^0), \ldots)$ of FAPI, every configuration (S^n, E^n, N^n) in ρ, every process $c \in \mathsf{C}$ that is honest in S^n, every session id sessionId, we have that if*

(1) there exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \to c} (S', E', N')$ 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 $\mathrm{enc_a}(\langle\langle \mathsf{HTTPReq}, \cdot, \cdot, \cdot, /\texttt{redirect\_ep}, parameters, headers, \cdot\rangle, \cdot\rangle, \cdot)$*, and*
*(3) $S^n(c)$.`sessions`[*sessionId*][`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* $\mathrm{sig}(authzResponse, signKey)$*, with authzResponse being a dictionary with at least the keys* `iss` *and* `code`.
(II) If there is an $as \in \mathsf{AS}$ with $S^n(as)$.`jwk` $\equiv signKey$, and as honest in S^n, then there is a processing step $P = (S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ 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 $\mathrm{enc_a}(\langle\langle \mathsf{HTTPReq}, \cdot, \cdot, \cdot, path, parameters, headers, \cdot\rangle, \cdot\rangle, \cdot)$, see Lines 7ff. of Algorithm 41, where $path \equiv$ `/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:
- `response` $\in$ *parameters*: otherwise *parameters*[`response`] $\equiv \langle\rangle$ (Definition 47), and by Figure 10, $\mathrm{checksig}(\langle\rangle, k) \not\equiv \top$ for any k, and
- *parameters*[`response`] must be of the form $\mathrm{sig}(authzResponse, signKey)$ (see equational theory in Figure 10), 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 $as \in \mathsf{AS}$ 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 $\mathrm{sig}(authzResponse, signKey)$ (see Figure 10). From the same lemma – in conjunction with Figure 10, and the fact that c

knows sig(*authzResponse*, *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 35) Holds). *For every run $\rho = ((S^0, E^0, N^0), ...)$ of FAPI, every configuration (S^n, E^n, N^n) in ρ, every process $rs \in \mathsf{RS}$ that is honest in S^n, every request id requestId, we have that if there exists a processing step $Q = (S, E, N) \xrightarrow{e_{in} \to rs} (S', E', N')$ in ρ such that*

$S(rs).\mathtt{pendingResponses}[requestId][\mathtt{requestSignedIntrospecResponse}] \equiv \top$, *and* $requestId \notin S'(rs).\mathtt{pendingResponses}$, *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 $\mathsf{enc_s}(\langle \mathtt{HTTPResp}, \cdot, \cdot, \cdot, body \rangle, \cdot)$, where body is of the form $\mathsf{sig}(introspecResponse, signKey)$.

(II) If there is an $as \in \mathsf{AS}$ with $S^n(as).\mathtt{jwk} \equiv signKey$, and as honest in S^n, then there is a processing step $P = (S^i, E^i, N^i) \xrightarrow[as \to E_{out}]{} (S^{i+1}, E^{i+1}, N^{i+1})$ 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 $requestId \in S(rs).\mathtt{pendingResponses}$ (otherwise, $S(rs).\mathtt{pendingResponses}[requestId]$ would have the value $\langle\rangle \not\equiv \top$). 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 $\mathsf{enc_s}(resp, \cdot)$, where $resp \in \mathtt{HTTPResponses}$, and hence *resp* is of the form $\langle \mathtt{HTTPResp}, \cdot, \cdot, \cdot, body \rangle$ (see Lines 19ff. of Algorithm 41).
Since $S(rs).\mathtt{pendingResponses}[requestId][\mathtt{requestSignedIntrospecResponse}] \equiv \top$, 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 $\mathsf{sig}(introspecResponse, signKey)$ (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 $as \in \mathsf{AS}$ with $S^n(as).\mathtt{jwk} \equiv signKey$, 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 10). From the same lemma – in conjunction with Figure 10, 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`, $\bot\rangle$
(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 47). extractmsg($\langle\rangle$) in Line 6 of Algorithm 21 is undefined (Figure 10), 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 47). Hence, *coveredComponents* = $\langle\rangle$ in Line 5 of Algorithm 21. Recall Definition 48 and Figure 10: $\langle\rangle.1 := \pi_1(\langle\rangle) := \diamond$. 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 47), 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$(*,*)$ (see Figure 10).

(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 36) Holds). *For every run* $\rho = ((S^0, E^0, N^0), \ldots)$ *of FAPI, 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 $Q = (S, E, N) \xrightarrow[rs \to E_{out}]{} (S', E', N')$ *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 $\mathsf{enc}_s(\langle$`HTTPResp`$, \cdot, \cdot, \cdot, body\rangle, \cdot)$ *with body* $\equiv$ [`resource`: *resource*].
(II) There exists a processing step $R = s^r \xrightarrow{e_{in} \to rs} s^{r'}$ *prior or equal to Q in* ρ *such that* $e_{in} = \langle y, x, resReq\rangle$*, and rs generated resource during R in Line 48 of Algorithm 18.*
(III) resReq is of the form $\mathsf{enc}_a(\langle\langle$`HTTPReq`$, \cdot,$ *method, host, path, parameters, headers, body*$\rangle, \cdot\rangle, \cdot)$ *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*, $\bot\rangle$, $\langle$`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 $c \in \mathsf{C}$ *which is honest in* S^n*, a domain selectedAS, and an index* $j \leq n$ *such that* $S^j(c)$.asAccounts[*selectedAS*][sign_key] ≡ *clientSignKey, then there is a processing step* $P = (S^i, E^i, N^i) \xrightarrow[c \to E'_{out}]{} (S^{i+1}, E^{i+1}, N^{i+1})$ *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 $\mathsf{enc}_s(\langle \mathtt{HTTPResp}, \cdot, 200, \cdot, \mathit{body} \rangle, \cdot)$. 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 $\mathsf{enc}_s(\langle \mathtt{HTTPResp}, \cdot, 200, \cdot, \mathit{body} \rangle, \cdot)$. 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 60), 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 60), hence concluding this sub-proof.

(III) From (II), we know that during R, *rs* processed an input event $\langle y, x, \mathit{resReq} \rangle$, and executed Line 48 of Algorithm 18. Hence, *rs* must have executed Algorithm 18, which is only called in Line 9 of Algorithm 41. This in turn only happens if the input message, i.e., *resReq*, is of the form $\mathsf{enc}_a(\langle\langle \mathtt{HTTPReq}, \cdot, \mathit{method}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body} \rangle, \cdot\rangle, \cdot)$ (see Lines 7ff. of Algorithm 41).

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 36 – 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 41 with the decrypted first sequence element of the input event as the first element, i.e., $m \equiv \mathrm{dec_a}(\mathit{resReq}, k).1$ with the "correct" k, and hence $m \equiv \langle \mathtt{HTTPReq}, \cdot, \mathit{method}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body} \rangle$. 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., $\mathit{origReq} \equiv r[\mathtt{originalRequest}]$, 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 $\mathit{origReq} \equiv \mathrm{dec_a}(\mathit{resReq}, k).1$ (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 $c \in \mathsf{C}$, honest in S^n, an issuer identifier *selectedAS*, and an index $j \leq n$ such that $S^j(c).\mathtt{asAccounts}[\mathit{selectedAS}][\mathtt{sign\_key}] \equiv \mathit{clientSignKey}$ (i.e., the key used for the signature from (III)).

Since c is honest, and we have $selectedAS \in S^j(c).\texttt{asAccounts}$, 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 $\mathsf{sig}(signatureBase, clientSignKey)$ (see Figure 10).
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 $\mathsf{sig}(signatureBase, clientSignKey)$, 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 37) Holds). *For every run $\rho = ((S^0, E^0, N^0), \ldots)$ of FAPI, every configuration (S^n, E^n, N^n) in ρ, every client $c \in \mathsf{C}$ which is honest in S^n, every session id $sessionId \in S^n(c).\texttt{sessions}$ such that*

(1) $S^n(c).\texttt{sessions}[sessionId][\texttt{expect\_signed\_resource\_res}] \equiv \top$, and
(2) $S^n(c).\texttt{sessions}[sessionId][\texttt{resource}] \equiv resource$,

then all of the following hold true:

(I) There exists a processing step $P = (S, E, N) \xrightarrow{e_{in} \to c} (S', E', N')$ 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 $\mathsf{enc}_\mathsf{s}(\langle \mathsf{HTTPResp}, \cdot, status, headers, body \rangle, \cdot)$, where $body \equiv [\texttt{resource}\colon resource]$, and $S(c) \neq S'(c)$.
(II) $headers[\texttt{Signature-Input}]$ is a dictionary with at least a key $\texttt{res}$ such that $headers[\texttt{Signature-Input}][\texttt{res}]$ is a sequence with at least two elements. For those first two elements, components, and metadata, we have $\langle \texttt{@status}, \langle\rangle \rangle$, $\langle \texttt{content-digest}, \langle\rangle \rangle \in^{\langle\rangle}$ components, and metadata is a dictionary with at least the key $\texttt{tag}$ such that $metadata[\texttt{tag}] \equiv \texttt{fapi-2-response}$.
(III) $headers[\texttt{Signature}]$ is a dictionary with at least a key $\texttt{res}$ such that $headers[\texttt{Signature}][\texttt{res}] \equiv \mathsf{sig}(signatureBase, rsSigKey)$. In addition, $signatureBase$ is of the form $[\langle \texttt{@status}, \langle\rangle \rangle\colon status, \langle \texttt{content-digest}, \langle\rangle \rangle\colon \mathsf{hash}(body)] +^{\langle\rangle}$ $[\texttt{tag}\colon \texttt{fapi-2-response}, \texttt{keyid}\colon keyId']$ for some $keyId'$; however, the dictionaries may contain additional elements.
(IV) There exists a domain $rsDom \in S^n(c).\texttt{rsSigKeys}$ such that $S^n(c).\texttt{rsSigKeys}[rsDom] \equiv \mathsf{pub}(rsSigKey)$.
(V) If process $rs := \mathsf{dom}^{-1}(rsDom)$ is honest in S^n, then there is a processing step $Q = s \xrightarrow[rs \to E_{out}]{} s'$, 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 13). 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 41, which only happens if the input event to the current processing step is an encrypted HTTP response (see Lines 19ff. of Algorithm 41), i.e., matches $\langle *, *, \mathsf{enc_s}(\langle \mathtt{HTTPResp}, *, *, *, * \rangle, *) \rangle$. 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 41) input message's `body` component under the key `resource` (Line 108 of Algorithm 3). Hence, $\mathsf{dec_s}(m, k).\mathtt{body}[\mathtt{resource}]$ (for the "correct" k) is equivalent to *resource* as stored in the `sessions` state subterm.

Signature Check During P. From (1) we have
$S^n(c).\mathtt{sessions}[sessionId][\mathtt{expect\_signed\_resource\_res}] \equiv \top$.
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, $S^n(c).\mathtt{sessions}[sessionId][\mathtt{expect\_signed\_resource\_res}] \equiv \top$ 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. $headers[\mathtt{Signature\text{-}Input}][\mathtt{res}]$ 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 $\langle \mathtt{@status}, \langle\rangle \rangle$, $\langle \mathtt{content\text{-}digest}, \langle\rangle \rangle \in^{\langle\rangle} headers[\mathtt{Signature\text{-}Input}][\mathtt{res}].1$, and $headers[\mathtt{Signature\text{-}Input}][\mathtt{res}].2[\mathtt{tag}] \equiv \mathtt{fapi\text{-}2\text{-}response}$ – 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).

- $\mathtt{res} \in headers[\mathtt{Signature}]$ Proof by contradiction: if $\mathtt{res} \notin headers[\mathtt{Signature}]$, then $headers[\mathtt{Signature}][\mathtt{res}] \equiv \langle\rangle$ (Definition 47). extractmsg($\langle\rangle$) in Line 97 of Algorithm 3 is undefined (Figure 10), and therefore P is not in ρ. This is a contradiction to (I).
- $headers[\mathtt{Signature}][\mathtt{res}] \sim \mathsf{sig}(*, *)$ Proof by contradiction: assume $headers[\mathtt{Signature}][\mathtt{res}] \nsim \mathsf{sig}(*, *)$. Therefore $\forall k.\ \mathsf{checksig}(headers[\mathtt{Signature}][\mathtt{res}], k) \not\equiv \top$ in Line 106 of Algorithm 3, hence P stops without parameters, which is a contradiction to (I).
- $signatureBase[\langle \mathtt{@status}, \langle\rangle \rangle] \equiv status$ We have $\langle \mathtt{@status}, \langle\rangle \rangle \in^{\langle\rangle} headers[\mathtt{Signature\text{-}Input}][\mathtt{res}].1$ from (II). Therefore, IS_COMPONENT_EQUAL (Algorithm 1) in Line 103 of Algorithm 3 gets called with $\langle \mathtt{@status}, \langle\rangle \rangle$ 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 $signatureBase[\langle \mathtt{@status}, \langle\rangle \rangle] \equiv status$.
- $signatureBase[\langle \mathtt{content\text{-}digest}, \langle\rangle \rangle] \equiv \mathsf{hash}(body)$ From Line 92 of Algorithm 3 and (I) (P does not stop without parameters), we get $headers[\mathtt{Content\text{-}Digest}] \equiv \mathsf{hash}(body)$. With

the same argumentation as above for @status, we get $signatureBase[\langle \texttt{content-digest}, \langle\rangle \rangle] \equiv headers[\texttt{Content-Digest}]$. Combining these, we get $signatureBase[\langle \texttt{content-digest}, \langle\rangle \rangle] \equiv \mathsf{hash}(body)$.

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 $\mathsf{pub}(rsSigKey)$ (see Figure 10). Furthermore, this value is taken from c's rsSigKeys state subterm in Line 96 of Algorithm 3 with a key *rsDom*. Note: if $rsDom \notin S(c).\texttt{rsSigKeys}$, the value of *pubKey* would be $\langle\rangle \nsim \mathsf{pub}(*)$.

(V) From Definition 13, we have $\mathsf{dom}^{-1}(rsDom) \in \mathsf{RS}$ for all $rsDom \in S^0(c).\texttt{rsSigKeys}$. However, since an honest client never changes the contents of its rsSigKeys state subterm, we also get $\mathsf{dom}^{-1}(rsDom) \in \mathsf{RS}$ for all $rsDom \in S(c).\texttt{rsSigKeys}$, and hence, we can assume that $rs := \mathsf{dom}^{-1}(rsDom) \in \mathsf{RS}$ is an honest resource server (in S^n).
Furthermore, Definition 13 gives us $S(c).\texttt{rsSigKeys}[rsDom] \equiv \mathsf{pub}(\mathsf{signkey}(rs))$. This allows us to apply Lemma 10: only *rs* can derive *rsSigKey*, and hence, only *rs* can have created a term $\mathsf{sig}(signatureBase, rsSigKey)$ (see Figure 10). An honest resource server only creates signatures in Line 6 of Algorithm 20, and since c processes $\mathsf{sig}(signatureBase, rsSigKey)$, i.e., a signature created by *rs*, in P, there must have been a processing step Q during which *rs* created $\mathsf{sig}(signatureBase, rsSigKey)$, i.e., signed *signatureBase*.

□

F.7 Proof of Theorem

Theorem 1 follows from Lemmas 31-39, Lemma 41, and Lemma 42.

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \tag{7}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x \tag{8}$$
$$\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) = \top \tag{9}$$
$$\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x \tag{10}$$
$$\mathsf{checkmac}(\mathsf{mac}(x, y), y) = \top \tag{11}$$
$$\mathsf{extractmsg}(\mathsf{mac}(x, y)) = x \tag{12}$$
$$\pi_i(\langle x_1, \ldots, x_n \rangle) = x_i \quad \text{if } 1 \leq i \leq n \tag{13}$$
$$\pi_j(\langle x_1, \ldots, x_n \rangle) = \diamond \quad \text{if } j \notin \{1, \ldots, n\} \tag{14}$$
$$\pi_j(t) = \diamond \quad \text{if } t \text{ is not a sequence} \tag{15}$$

Fig. 10. Equational theory for Σ.

G TECHNICAL DEFINITIONS

Here, we provide technical definitions of the WIM. These follow the descriptions in [24, 33, 34, 35, 36, 37, 38].

G.1 Terms and Notations

Definition 38 (Signature Σ). *We define the signature Σ, over which we will define formal terms, as the union of the following pairwise disjoint sets:*

Constants $C = \mathbb{S} \cup \mathsf{IPs} \cup \{\bot, \top, \diamond\}$ *with the three sets pairwise disjoint. $\mathbb{S}$ is the set of all (ASCII) strings, including the empty string ε.* 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:* $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot)$, $\mathsf{mac}(\cdot,\cdot)$, $\mathsf{checkmac}(\cdot,\cdot)$, $\mathsf{extractmsg}(\cdot)$, $\mathsf{hash}(\cdot)$.

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

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

Definition 39 (Nonces and Terms). *By $X = \{x_0, x_1, \ldots\}$ we denote a set of variables and by $\mathcal{N}$ we denote an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of* terms *over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X \cup C$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol for some $n \geq 0$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*

By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by the theory associated with Σ (see Figure 10). For example, we have that $\pi_1(\mathsf{dec_a}(\mathsf{enc_a}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{pub}(k)), k)) \equiv \mathsf{a}$.

Definition 40 (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 $V_{process} = \{v_1, v_2, \ldots\}$ of variables (called placeholders). The set $\mathcal{M}^v := \mathcal{T}_{\mathcal{N}}(V_{process})$ is called the set of protomessages*, i.e., messages that can contain placeholders.*

Example 1. *For example, $k \in \mathcal{N}$ and $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message*

$\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k))$ *is interpreted to be the message* $\langle a, b, c\rangle$ *(the sequence of constants* a*,* b*,* c*) encrypted by the public key* $\mathsf{pub}(k)$*.*

Definition 41 (Events and Protoevents). *An* event (over IPs and $\mathcal{M}$) *is a term of the form* $\langle a, f, m\rangle$*, for* $a, f \in \mathsf{IPs}$ *and* $m \in \mathcal{M}$*, where* a *is interpreted to be the receiver address and* f *is the sender address. We denote by* $\mathcal{E}$ *the set of all events. Events over* IPs *and* $\mathcal{M}^\nu$ *are called* protoevents *and are denoted* $\mathcal{E}^\nu$*. By* $2^{\mathcal{E}\langle\rangle}$ *(or* $2^{\mathcal{E}^\nu\langle\rangle}$*, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g.,* $\langle\rangle$*,* $\langle\langle a, f, m\rangle, \langle a', f', m'\rangle, \dots\rangle$*, etc.).*

Definition 42 (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 10. For a term* t*, we denote its normal form as* $t{\downarrow}$*.*

Definition 43 (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* $t \mathrel{\dot\sim} patterns$ *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* $\langle\top, 42\rangle \sim p$*,* $\langle\bot, 42\rangle \nsim p$*, and*

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

Definition 44 (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 45 (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:*

- $t \subset^{\langle\rangle} s \iff t_1, \dots, t_n \in s$
- $x \in^{\langle\rangle} t \iff \exists i\colon\ t_i = x$
- $t +^{\langle\rangle} y := \langle t_1, \dots, t_n, y\rangle$
- $t \cup r := \langle t_1, \dots, t_n, r_1, \dots, r_m\rangle$
- $t -^{\langle\rangle} y := \begin{cases} \langle t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n\rangle & \text{if } \exists i\colon\ t_i = x \text{ (i.e., } y \in^{\langle\rangle} t\text{)} \\ t & \text{otherwise (i.e., } y \notin^{\langle\rangle} t\text{)} \end{cases}$

  If y *occurs more than once in* t*,* $t -^{\langle\rangle} y$ *non-deterministically removes one of the occurrences.*
- $t -^{\langle\rangle *} y$ *is* t *with all occurrences of* y *removed.*
- $|t| := n$*. If* t' *is not a sequence, we set* $|t'| := \diamond$*.*
- *For a finite set* M *with* $M = \{m_1, \dots, m_n\}$ *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 46 (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* $[k_1\colon v_1, \dots, k_n\colon v_n]$ *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 47 (Operations on Dictionaries). *Let $z = [k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n]$ 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:*

- $t \in z \iff \exists i \in \{1, \ldots, n\} : k_i = t$
- $z[t] := \begin{cases} v_i & \text{if } \exists k_i \in z : t = k_i \\ \langle\rangle & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- $z - t := \begin{cases} [k_1 : v_1, \ldots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1}, \ldots, k_n : v_n] & \text{if } \exists k_i \in z : t = k_i \\ z & \text{otherwise (i.e., if } t \notin z) \end{cases}$
- *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 $t = \langle t_1, \ldots, t_n\rangle$, we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 48 (Pointers). *A* pointer *is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.*

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

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

Definition 49 (Concatenation of Sequences). *For a sequence $a = \langle a_1, \ldots, a_i\rangle$ and a sequence $b = \langle b_1, b_2, \ldots\rangle$, we define the* concatenation *as $a \cdot b := \langle a_1, \ldots, a_i, b_1, b_2, \ldots\rangle$.*

Definition 50 (Subtracting from Sequences). *For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a non-deterministically chosen occurence of that element in X is removed.*

G.2 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.

G.2.1 URLs.

Definition 51. *A* URL *is a term of the form*

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

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

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

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 48):

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

G.2.2 Origins.

Definition 52. *An* origin *is a term of the form* $\langle host, protocol \rangle$ *with* $host \in \mathsf{Doms}$ *and* $protocol \in \{\mathsf{P}, \mathsf{S}\}$. *We write* Origins *for the set of all origins.*

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

G.2.3 Cookies.

Definition 53. *A* cookie *is a term of the form* $\langle name, content \rangle$ *where* $name \in \mathcal{T}_{\mathcal{N}}$, *and content is a term of the form* $\langle value, secure, session, httpOnly \rangle$ *where* $value \in \mathcal{T}_{\mathcal{N}}$, $secure, session, httpOnly \in \{\top, \bot\}$. *As name is a term, it may also be a sequence consisting of two parts. If the name consists of two parts, we call the first part of the sequence (i.e.,* $name.1$*) the prefix of the name. We write* Cookies *for the set of all cookies and* $\mathsf{Cookies}^{v}$ *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.[13] 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 [12]) of a cookie is set (i.e., *name* consists of two parts and $name.1 \equiv \mathtt{\_\_Host}$), 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 $\langle name, value \rangle$.

G.2.4 HTTP Messages.

Definition 54. *An* HTTP request *is a term of the form shown in (16). An* HTTP response *is a term of the form shown in (17).*

$$\langle \mathsf{HTTPReq}, nonce, method, host, path, parameters, headers, body \rangle \tag{16}$$

$$\langle \mathsf{HTTPResp}, nonce, status, headers, body \rangle \tag{17}$$

The components are defined as follows:

- $nonce \in \mathcal{N}$ *serves to map each response to the corresponding request.*
- $method \in \mathsf{Methods}$ *is one of the HTTP methods.*
- $host \in \mathsf{Doms}$ *is the host name in the HOST header of HTTP/1.1.*
- $path \in \mathbb{S}$ *indicates the resource path at the server side.*
- $status \in \mathbb{S}$ *is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard).*
- $parameters \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ *contains URL parameters.*
- $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ *contains request/response headers. The dictionary elements are terms of one of the following forms:*

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

- $\langle \texttt{Origin}, o \rangle$ *where* o *is an origin,*
- $\langle \texttt{Set-Cookie}, c \rangle$ *where* c *is a sequence of cookies,*
- $\langle \texttt{Cookie}, c \rangle$ *where* $c \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ *(note that in this header, only names and values of cookies are transferred),*
- $\langle \texttt{Location}, l \rangle$ *where* $l \in$ URLs*,*
- $\langle \texttt{Referer}, r \rangle$ *where* $r \in$ URLs*,*
- $\langle \texttt{Strict-Transport-Security}, \top \rangle$*,*
- $\langle \texttt{Authorization}, \langle \textit{username}, \textit{password} \rangle \rangle$ *where* $\textit{username}, \textit{password} \in \mathbb{S}$ *(this header models the 'Basic' HTTP Authentication Scheme, see [81]),*
- $\langle \texttt{ReferrerPolicy}, p \rangle$ *where* $p \in \{\texttt{noreferrer}, \texttt{origin}\}$*.*

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

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

Example 6 (HTTP Request and Response).

$$r := \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, \texttt{/show}, \langle\langle \texttt{index}, 1 \rangle\rangle, \\ [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S} \rangle], \langle \texttt{foo}, \texttt{bar} \rangle\rangle \tag{18}$$

$$s := \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top \rangle\rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x \rangle\rangle \tag{19}$$

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* $\langle \texttt{foo}, \texttt{bar} \rangle$*. 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 G.11) and the scripts initial state* x*.*

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 55. *An* encrypted HTTP request *is of the form* $\text{enc}_\text{a}(\langle m, k' \rangle, k)$*, where* $k \in \mathit{terms}$*,* $k' \in \mathcal{N}$*, and* $m \in$ HTTPRequests*. The corresponding* encrypted HTTP response *would be of the form* $\text{enc}_\text{s}(m', k')$*, where* $m' \in$ 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.

$$\text{enc}_\text{a}(\langle r, k' \rangle, \text{pub}(k_{\textit{example.com}})) \tag{20}$$

$$\text{enc}_\text{s}(s, k') \tag{21}$$

The term (20) shows an encrypted request (with r *as in (18)). It is encrypted using the public key* $\text{pub}(k_{\textit{example.com}})$*. 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).*

G.2.5 DNS Messages.

Definition 56. *A* DNS request *is a term of the form* $\langle \texttt{DNSResolve}, \textit{domain}, \textit{nonce} \rangle$ *where* $\textit{domain} \in$ Doms*,* $\textit{nonce} \in \mathcal{N}$*. We call the set of all DNS requests* DNSRequests*.*

Definition 57. *A* DNS response *is a term of the form* $\langle \texttt{DNSResolved}, \textit{domain}, \textit{result}, \textit{nonce} \rangle$ *with* $\textit{domain} \in$ Doms*,* $\textit{result} \in$ IPs*,* $\textit{nonce} \in \mathcal{N}$*. 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.

G.3 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 58 (Generic Atomic Processes and Systems). *A* (generic) atomic process *is a tuple*

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

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

Definition 59 (Configurations). *A* configuration of a system $\mathcal{P}$ *is a tuple* (S, E, N) *where the state of the system* S *maps every atomic process* $p \in \mathcal{P}$ *to its current state* $S(p) \in Z^p$*, the sequence of waiting events* E *is an infinite sequence*[14] $(e_1, e_2, \dots)$ *of events waiting to be delivered, and* N *is an infinite sequence of nonces* $(n_1, n_2, \dots)$*.*

Definition 60 (Processing Steps). *A* processing step of the system $\mathcal{P}$ *is of the form*

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

where

(1) (S, E, N) *and* (S', E', N') *are configurations of* $\mathcal{P}$*,*
(2) $e_{in} = \langle a, f, m \rangle \in E$ *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) $E_{out}^{\nu} \subseteq 2^{\mathcal{E}^{\nu}\langle\rangle}$ *of protoevents,*
(2) a term $s^{\nu} \in \mathcal{T}_{\mathcal{N}}(V_{process})$*,*
(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) $((e_{in}, S(p)), (E_{out}^{\nu}, s^{\nu})) \in R^p$ *and* $a \in I^p$*,*
(2) $E_{out} = E_{out}^{\nu}[\eta_1/v_1, \dots, \eta_i/v_i]$*,*
(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) $E' = E_{out} \cdot (E \setminus \{e_{in}\})$*,*
(5) $N' = N \setminus N^{\nu}$*.*

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

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

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

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 61 (Runs). *Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A* run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0 *is a finite sequence of configurations $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s_0^p$ 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).*

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 $\langle a, a, \mathtt{TRIGGER}\rangle$ for each $a \in \mathsf{IPs}$, interleaved by address.

G.4 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 62 (Deriving Terms). *Let M be a set of ground terms. We say that* a term m can be derived from M with placeholders V *if there exist $n \geq 0$, $m_1, \ldots, m_n \in M$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \ldots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.*

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

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

Definition 63 (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, $s \in \mathcal{T}_\mathcal{N}$, $s' \in \mathcal{T}_\mathcal{N}(V_{process})$, with $((e, s), (E, s')) \in R^p$ it holds true that E, $s' \in d_{V_{process}}(\{e, s\})$.*

G.5 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 64 (Atomic Attacker Process). *An* (atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$ *is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e, and $s \in \mathcal{T}_\mathcal{N}$ we have that $((e, s), (E, s')) \in R$ iff $s' = \langle e, E, s\rangle$ and $E = \langle\langle a_1, f_1, m_1\rangle, \ldots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_1, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{process}}(\{e, s\})$.*

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.

G.6 Notations for Functions and Algorithms

When describing algorithms, we use the following notations:

G.6.1 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** $\langle \mathtt{Constant}, x, y\rangle \equiv t$ **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 $\mathtt{Constant} \equiv \pi_1(t)$ and if $|\langle \mathtt{Constant}, x, y\rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

G.6.2 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.

G.6.3 Stop without output. We write **stop** (without further parameters) to denote that there is no output and no change in the state.

G.6.4 Placeholders. In several places throughout the algorithms we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 39). Table 2 shows a list of some of the placeholders, generally denoted by ν with some subscript to distinguish between multiple fresh values.

G.6.5 Abbreviations for URLs and Origins. We sometimes use an abbreviation for URLs. We write URL^d_{path} to describe the following URL term: $\langle \mathsf{URL}, \mathsf{S}, d, path, \langle\rangle\rangle$. If the domain d belongs to some distinguished process P and it is the only domain associated to this process, we may also write $\mathsf{URL}^{\mathsf{P}}_{path}$. For a (secure) origin $\langle d, \mathsf{S}\rangle$ of some domain d, we also write origin_d. Again, if the domain d belongs to some distinguished process P and d is the only domain associated to this process, we may write $\mathsf{origin}_{\mathsf{P}}$.

G.7 Browsers

Here, we present the formal model of browsers.

G.7.1 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).

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
$\nu_{10}, \dots$	Algorithm 28, replacement for placeholders in script output

Table 2. List of placeholders used in browser algorithms.

Definition 65 (Placeholders for Scripts). *By $V_{script} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.*

Definition 66 (Scripts). *A* script *is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{script})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{script})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{script}}(s)$.*

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

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

Definition 67 (Attacker Script). *The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{att} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{script}}(s)\}$.*

G.7.2 Web Browser State. Before we can define the state of a Web browser, we first have to define windows and documents.

Definition 68. *A* window *is a term of the form $w = \langle nonce, documents, opener \rangle$ with $nonce \in \mathcal{N}$, $documents \subset^{\langle\rangle}$* Documents *(defined below), $opener \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle}$ documents if documents is not empty (we then call d the* active document of w*). We write* Windows *for the set of all windows. We write $w.\texttt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else.*

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

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

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

Definition 69. *A* document *d is a term of the form*

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

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

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

Definition 70. *For two window terms* w *and* w' *we write*

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

if $w \in^{\langle\rangle} w'$.activedocument.subwindows. *We write* $\xrightarrow{childof^+}$ *for the transitive closure and we write* $\xrightarrow{childof^*}$ *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 71. *A reference for a normal HTTP(S) request is a sequence of the form* $\langle$REQ*, nonce*$\rangle$*, where nonce is a window reference. A reference for a XMLHttpRequest is a sequence of the form* $\langle$XHR*, nonce, xhrreference*$\rangle$*, where nonce is a document reference and xhrreference is some nonce that was chosen by the script that initiated the request.*

We can now define the set of states of Web browsers. Note that we use the dictionary notation that we introduced in Definition 46.

Definition 72. *The* set of states $Z_{webbrowser}$ of a Web browser atomic Dolev-Yao process *consists of the terms of the form*

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

with the subterms as follows:

- *windows* $\subset^{\langle\rangle}$ Windows *contains a list of window terms (modeling top-level windows, or browser tabs) which contain documents, which in turn can contain further window terms (iframes).*
- *ids* $\subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ *is a list of identities that are owned by this browser (i.e., belong to the user of the browser).*
- *secrets* $\in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$ *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.*
- *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* $\subset^{\langle\rangle}$ Doms *stores the list of domains that the browser only accesses via TLS (strict transport security).*
- *DNSaddress* $\in$ IPs *defines the IP address of the DNS server.*

- *pendingDNS* $\in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ *contains one pairing per unanswered DNS query of the form* $\langle reference, request, url\rangle$. *In these pairings, reference is an HTTP(S) request reference (as above), request contains the HTTP(S) message that awaits DNS resolution, and url contains the URL of said HTTP request. The pairings in pendingDNS are indexed by the DNS request/response nonce.*
- *pendingRequests* $\in \mathcal{T}_{\mathcal{N}}$ *contains pairings of the form* $\langle reference, request, url, key, f\rangle$ *with reference, request, and url as in pendingDNS, key is the symmetric encryption key if HTTPS is used or* $\perp$ *otherwise, and* f *is the IP address of the server to which the request was sent.*
- *isCorrupted* $\in \{\perp, \mathtt{FULLCORRUPT}, \mathtt{CLOSECORRUPT}\}$ *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* $\in \left[\mathsf{Doms} \times \mathcal{N}\right]$ *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).

G.7.3 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.

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

Subwindows and Docs. Given a browser state s, $\mathsf{Subwindows}(s)$ denotes the set of all pointers[15] to windows in the window list $s.\mathtt{windows}$ and (recursively) the subwindows of their active documents. We exclude subwindows of inactive documents and their subwindows. With $\mathsf{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\mathsf{Subwindows}(s)$.

Definition 73. *For a browser state s we denote by* $\mathsf{Subwindows}(s)$ *the minimal set of pointers that satisfies the following conditions: (1) For all windows* $w \in^{\langle\rangle} s.\mathtt{windows}$ *there is a* $\overline{p} \in \mathsf{Subwindows}(s)$ *such that* $s.\overline{p} = w$. *(2) For all* $\overline{p} \in \mathsf{Subwindows}(s)$, *the active document d of the window* $s.\overline{p}$ *and every subwindow w of d there is a pointer* $\overline{p'} \in \mathsf{Subwindows}(s)$ *such that* $s.\overline{p'} = w$.

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

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

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

Definition 74. *Let s be a browser state and d a document. By* $\mathsf{Clean}(s, d)$ *we denote the term that equals* $s.\mathtt{windows}$ *but with (1) all inactive documents removed (including their subwindows etc.), (2) all subterms that represent non-same-origin documents w.r.t. d replaced by a limited document d' with the same nonce and the same subwindow list, and (3) the values of the subterms* $\mathtt{headers}$ *for all documents set to* $\langle\rangle$. *(Note that non-same-origin documents on all levels are replaced by their corresponding limited document.)*

CookieMerge. The function CookieMerge merges two sequences of cookies together: When used in the browser, *oldcookies* is the sequence of existing cookies for some origin, *newcookies* is a

[15] Recall the definition of a pointer in Definition 48.

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

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

Definition 76. *For a sequence of cookies (with pairwise different names) oldcookies, a cookie c, and a string protocol* $\in \{\mathsf{P}, \mathsf{S}\}$ *(denoting whether the HTTP response was received from an insecure or a secure origin), the sequence* AddCookie(*oldcookies*, *c*, *protocol*) *is defined by the following algorithm: Let m* := *oldcookies. If* $(c.\mathtt{name.1} \equiv \mathtt{\_\_Host}) \wedge \neg((protocol \equiv \mathsf{S}) \wedge (c.\mathtt{secure} \equiv \top))$*, then return m, else: Remove any c′ from m that has* $c.\mathtt{name} \equiv c'.\mathtt{name}$*. Append c to m and return m.*

NavigableWindows. The function NavigableWindows returns a set of windows that a document is allowed to navigate. We closely follow [7], Section 5.1.4 for this definition.

Definition 77. *The set* NavigableWindows$(\overline{w}, s')$ *is the set* $\overline{W} \subseteq$ Subwindows(s') *of pointers to windows that the active document in* $\overline{w}$ *is allowed to navigate. The set* $\overline{W}$ *is defined to be the minimal set such that for every* $\overline{w'} \in$ Subwindows(s') *the following is true:*

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

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 $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When it is given a window reference (nonce) *window*, this function returns a pointer to a selected window term in s':
  - If *window* is the string _BLANK, a new window is created and a pointer to that window is returned.

Algorithm 22 Web Browser Model: Determine window for navigation.

```
1: function GETNAVIGABLEWINDOW(w̄, window, noreferrer, s′)
2:     if window ≡ _BLANK then    → Open a new window when _BLANK is used
3:         if noreferrer ≡ ⊤ then
4:             let w′ := ⟨v9, ⟨⟩, ⊥⟩
5:         else
6:             let w′ := ⟨v9, ⟨⟩, s′.w̄.nonce⟩
7:         let s′.windows := s′.windows +^⟨⟩ w′
           ↪ and let w̄′ be a pointer to this new element in s′
8:         return w̄′
9:     let w̄′ ← NavigableWindows(w̄, s′) such that s′.w̄′.nonce ≡ window
       ↪ if possible; otherwise return w̄
10:    return w̄′
```

Algorithm 23 Web Browser Model: Determine same-origin window.

```
1: function GETWINDOW(w̄, window, s′)
2:     let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
       ↪ if possible; otherwise return w̄
3:     if s′.w̄′.activedocument.origin ≡ s′.w̄.activedocument.origin then
4:         return w̄′
5:     return w̄
```

Algorithm 24 Web Browser Model: Cancel pending requests for given window.

```
1: function CANCELNAV(reference, s′)
2:     remove all ⟨reference, req, url, key, f⟩ from s′.pendingRequests for any req, url, key, f
3:     remove all ⟨x, ⟨reference, message, url⟩⟩ from s′.pendingDNS
       ↪ for any x, message, url
4:     return 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 71. *url* contains the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). *origin* is the origin header value that is to be added to the HTTP request.

Algorithm 25 Web Browser Model: Prepare headers, do DNS resolution, save message.

function HTTP_SEND(*reference*, *message*, *url*, *origin*, *referrer*, *referrerPolicy*, *a*, s')
  if *message*.host $\in^{\langle\rangle}$ s'.sts **then**
    let *url*.protocol := S
  let *cookies* := $\langle\{\langle c.\text{name}, c.\text{content.value}\rangle \mid c \in^{\langle\rangle} s'.\text{cookies}[message.\text{host}] \hookrightarrow \wedge (c.\text{content.secure} \equiv \top \implies (url.\text{protocol} \equiv \text{S}))\}\rangle$
  let *message*.headers[Cookie] := *cookies*
  if *origin* $\not\equiv \bot$ **then**
    let *message*.headers[Origin] := *origin*
  if *referrerPolicy* ≡ no-referrer **then**
    let *referrer* := $\bot$
  if *referrer* $\not\equiv \bot$ **then**
    if *referrerPolicy* ≡ origin **then**
      let *referrer* := $\langle$URL, *referrer*.protocol, *referrer*.host, /, $\langle\rangle$, $\bot\rangle$
        → Referrer stripped down to origin.
    let *referrer*.fragment := $\bot$
      → Browsers do not send fragment identifiers in the Referer header.
    let *message*.headers[Referer] := *referrer*
  let s'.pendingDNS[v_8] := $\langle$*reference*, *message*, *url*$\rangle$
  stop $\langle\langle s'.\text{DNSaddress}, a, \langle\text{DNSResolve}, message.\text{host}, v_8\rangle\rangle\rangle$, s'

Algorithm 26 Web Browser Model: Navigate a window backward.

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

Algorithm 27 Web Browser Model: Navigate a window forward.

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

- 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 $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted.

Algorithm 28 Web Browser Model: Execute a script.

1: **function** RUNSCRIPT($\overline{w}$, $\overline{d}$, a, s')
2: **let** *tree* := Clean(s', $s'.\overline{d}$)
3: **let** *cookies* := $\langle \{ \langle c.\texttt{name}, c.\texttt{content.value} \rangle | c \in^{\langle\rangle} s'.\texttt{cookies} \left[s'.\overline{d}.\texttt{origin.host} \right]$
↪ $\wedge$ $c.\texttt{content.httpOnly} \equiv \bot$
↪ $\wedge \left(c.\texttt{content.secure} \equiv \top \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S} \right) \right) \} \rangle$
4: **let** *tlw* $\leftarrow$ s'.windows **such that** *tlw* is the top-level window containing $\overline{d}$
5: **let** *sessionStorage* := $s'.\texttt{sessionStorage} \left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce} \rangle \right]$
6: **let** *localStorage* := $s'.\texttt{localStorage} \left[s'.\overline{d}.\texttt{origin} \right]$
7: **let** *secrets* := $s'.\texttt{secrets} \left[s'.\overline{d}.\texttt{origin} \right]$
8: **let** R := $\text{script}^{-1}(s'.\overline{d}.\texttt{script})$ **if possible; otherwise stop**
9: **let** *in* := $\langle tree, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}, s'.\overline{d}.\texttt{scriptinputs}, cookies,$
↪ $localStorage, sessionStorage, s'.\texttt{ids}, secrets \rangle$
10: **let** *state*$'$ $\leftarrow$ $\mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, *cookies*$'$ $\leftarrow$ Cookies^{v}, *localStorage*$'$ $\leftarrow$ $\mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
↪ *sessionStorage*$'$ $\leftarrow$ $\mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, *command* $\leftarrow$ $\mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
↪ *out* := $\langle$*state*$'$, *cookies*$'$, *localStorage*$'$, *sessionStorage*$'$, *command*$\rangle$
↪ **such that** *out* := $out^{\lambda}[v_{10}/\lambda_1, v_{11}/\lambda_2, \dots]$ **with** $(in, out^{\lambda}) \in R$
11: **let** $s'.\texttt{cookies} \left[s'.\overline{d}.\texttt{origin.host} \right]$:=
↪ $\langle \text{CookieMerge}(s'.\texttt{cookies} \left[s'.\overline{d}.\texttt{origin.host} \right], cookies', s'.\overline{d}.\texttt{origin.protocol}) \rangle$
12: **let** $s'.\texttt{localStorage} \left[s'.\overline{d}.\texttt{origin} \right]$:= *localStorage*$'$
13: **let** $s'.\texttt{sessionStorage} \left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce} \rangle \right]$:= *sessionStorage*$'$
14: **let** $s'.\overline{d}.\texttt{scriptstate}$:= *state*$'$
15: **let** *referrer* := $s'.\overline{d}.\texttt{location}$
16: **let** *referrerPolicy* := $s'.\overline{d}.\texttt{headers[ReferrerPolicy]}$
17: **let** *docorigin* := $s'.\overline{d}.\texttt{origin}$
18: **switch** *command* **do**
19: **case** $\langle \texttt{HREF}, url, hrefwindow, noreferrer \rangle$
20: **let** $\overline{w'}$:= GETNAVIGABLEWINDOW($\overline{w}$, *hrefwindow*, *noreferrer*, s')
21: **let** *reference* := $\langle \texttt{REQ}, s'.\overline{w'}.\texttt{nonce} \rangle$
22: **let** *req* := $\langle \texttt{HTTPReq}, v_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle \rangle$
23: **if** *noreferrer* $\equiv \top$ **then**
24: **let** *referrerPolicy* := noreferrer
25: **let** s' := CANCELNAV(*reference*, s')
26: **call** HTTP_SEND(*reference*, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, a, s')
27: **case** $\langle \texttt{IFRAME}, url, window \rangle$
28: **if** *window* $\equiv$ _SELF **then**
29: **let** $\overline{w'}$:= $\overline{w}$
30: **else**
31: **let** $\overline{w'}$:= GETWINDOW($\overline{w}$, *window*, s')
32: **let** *req* := $\langle \texttt{HTTPReq}, v_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle \rangle$
33: **let** w' := $\langle v_5, \langle\rangle, \bot \rangle$
34: **let** $s'.\overline{w'}.\texttt{activedocument.subwindows} := s'.\overline{w'}.\texttt{activedocument.subwindows} +^{\langle\rangle} w'$
35: **call** HTTP_SEND($\langle \texttt{REQ}, v_5 \rangle$, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, a, s')

→ Algorithm continues on next page.

```
36:     case ⟨FORM, url, method, data, hrefwindow⟩
37:         if method ∉ {GET, POST} then
38:             stop
39:         let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s′)
40:         let reference := ⟨REQ, s′.w̄′.nonce⟩
41:         if method = GET then
42:             let body := ⟨⟩
43:             let parameters := data
44:             let origin := ⊥
45:         else
46:             let body := data
47:             let parameters := url.parameters
48:             let origin := docorigin
49:         let req := ⟨HTTPReq, v4, method, url.host, url.path, parameters, ⟨⟩, body⟩
50:         let s′ := CANCELNAV(reference, s′)
51:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
52:     case ⟨CIBAFORM, url, method, data, hrefwindow, clientDomain, cibaBindingMessage⟩
            → Custom CIBA FORM command: When starting a CIBA flow, the client returns a bind-
              ing message. When authenticating at the AS, the end-user has to make sure that they
              receive the same value. For modeling this behavior, we extend the browser state by
              the cibaBindingMessages subterm and define this command which first checks if the
              cibaBindingMessage is stored by the browser and then continues as the FORM command.
              Note that this command is a modeling artifact.
53:         if ⟨clientDomain, cibaBindingMessage⟩ ∉^⟨⟩ s′.cibaBindingMessages then
54:             stop
55:         if method ∉ {GET, POST} then
56:             stop
57:         let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s′)
58:         let reference := ⟨REQ, s′.w̄′.nonce⟩
59:         if method = GET then
60:             let body := ⟨⟩
61:             let parameters := data
62:             let origin := ⊥
63:         else
64:             let body := data
65:             let parameters := url.parameters
66:             let origin := docorigin
67:         let req := ⟨HTTPReq, v4, method, url.host, url.path, parameters, ⟨⟩, body⟩
68:         let s′ := CANCELNAV(reference, s′)
69:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
70:     case ⟨SETSCRIPT, window, script⟩
71:         let w̄′ := GETWINDOW(w̄, window, s′)
72:         let s′.w̄′.activedocument.script := script
73:         stop ⟨⟩, s′
74:     case ⟨SETSCRIPTSTATE, window, scriptstate⟩
75:         let w̄′ := GETWINDOW(w̄, window, s′)
76:         let s′.w̄′.activedocument.scriptstate := scriptstate
77:         stop ⟨⟩, s′
```

→ Algorithm continues on next page.

```
78:     case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
79:         if method ∈ {CONNECT, TRACE, TRACK} ∨ xhrreference ∉ V_process ∪ {⊥} then
80:             stop
81:         if url.host ≠ docorigin.host ∨ url.protocol ≠ docorigin.protocol then
82:             stop
83:         if method ∈ {GET, HEAD} then
84:             let data := ⟨⟩
85:             let origin := ⊥
86:         else
87:             let origin := docorigin
88:         let req := ⟨HTTPReq, v_4, method, url.host, url.path, url.parameters, ⟨⟩, data⟩
89:         let reference := ⟨XHR, s′.d̄.nonce, xhrreference⟩
90:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
91:     case ⟨BACK, window⟩
92:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
93:         call NAVBACK(w̄′, s′)
94:     case ⟨FORWARD, window⟩
95:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
96:         call NAVFORWARD(w̄′, s′)
97:     case ⟨CLOSE, window⟩
98:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
99:         remove s′.w̄′ from the sequence containing it
100:        stop ⟨⟩, s′
101:    case ⟨POSTMESSAGE, window, message, origin⟩
102:        let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
103:        if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
            ↪ ∧ (origin ≠ ⊥ ⟹ s′.w̄′.documents.j̄.origin ≡ origin) then
104:            let s′.w̄′.documents.j̄.scriptinputs := s′.w̄′.documents.j̄.scriptinputs
                ↪ +^⟨⟩ ⟨POSTMESSAGE, s′.w̄.nonce, docorigin, message⟩
105:        stop ⟨⟩, s′
106:    case else
107:        stop
```

- 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 71. *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).

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

Definition 78. *The pair* $((\langle a, f, m\rangle, s), (M, s'))$ *belongs to* $R_{webbrowser}$ *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'*.*

Algorithm 29 Web Browser Model: Process an HTTP response.

```
1: function PROCESSRESPONSE(response, reference, request, requestUrl, a, f, s′)
2:     if Set-Cookie ∈ response.headers then
3:         for each c ∈⟨⟩ response.headers [Set-Cookie], c ∈ Cookies do
4:             let s′.cookies [request.host]
                   ↪ := AddCookie(s′.cookies [request.host], c, requestUrl.protocol)
5:     if Strict-Transport-Security ∈ response.headers ∧ requestUrl.protocol ≡ S then
6:         let s′.sts := s′.sts +⟨⟩ request.host
7:     if Referer ∈ request.headers then
8:         let referrer := request.headers[Referer]
9:     else
10:        let referrer := ⊥
11:    if Location ∈ response.headers ∧ response.status ∈ {303, 307} then
12:        let url := response.headers [Location]
13:        if url.fragment ≡ ⊥ then
14:            let url.fragment := requestUrl.fragment
15:        let method′ := request.method
16:        let body′ := request.body
17:        if Origin ∈ request.headers
               ↪ ∧ request.headers[Origin] ≠ ◇
               ↪ ∧ (⟨url.host, url.protocol⟩ ≡ ⟨request.host, requestUrl.protocol⟩
                   ↪ ∨ ⟨request.host, requestUrl.protocol⟩ ≡ request.headers[Origin]) then
18:            let origin := request.headers[Origin]
19:        else
20:            let origin := ◇
21:        if response.status ≡ 303 ∧ request.method ∉ {GET, HEAD} then
22:            let method′ := GET
23:            let body′ := ⟨⟩
24:        if ∃ w̄ ∈ Subwindows(s′) such that s′.w̄.nonce ≡ π2(reference) then   → Do not redirect XHRs.
25:            let req := ⟨HTTPReq, v6, method′, url.host, url.path, url.parameters, ⟨⟩, body′⟩
26:            let referrerPolicy := response.headers[ReferrerPolicy]
27:            call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
28:        else
29:            stop ⟨⟩, s′
               → This algorithm is continued on the next page.
```

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).

G.8 Definition of Web Browsers

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

Definition 79 (Web Browser atomic Dolev-Yao Process). *A Web browser atomic Dolev-Yao process is an atomic Dolev-Yao process of the form $p = (I^p, Z_{webbrowser}, R_{webbrowser}, s_0^p)$ for a set I^p of addresses, $Z_{webbrowser}$ and $R_{webbrowser}$ as defined above, and an initial state $s_0^p \in Z_{webbrowser}$.*

Definition 80 (Web Browser Initial State). *An initial state $s_0^p \in Z_{webbrowser}$ for a browser process p is a Web browser state (Definition 72) with the following properties:*

- s_0^p.windows $\equiv \langle\rangle$

30: **switch** $\pi_1(\mathit{reference})$ **do**
31: **case** REQ
32: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\mathsf{nonce} \equiv \pi_2(\mathit{reference})$ **if possible;**
↪ **otherwise stop** → normal response
33: **if** $\mathit{response}.\mathsf{body} \nsim \langle *, * \rangle$ **then**
34: **stop** $\langle\rangle, s'$
35: **let** $\mathit{script} := \pi_1(\mathit{response}.\mathsf{body})$
36: **let** $\mathit{scriptstate} := \pi_2(\mathit{response}.\mathsf{body})$
37: **let** $d := \langle v_7, \mathit{requestUrl}, \mathit{response}.\mathsf{headers}, \mathit{referrer}, \mathit{script}, \mathit{scriptstate}, \langle\rangle, \langle\rangle, \top \rangle$
38: **if** $s'.\overline{w}.\mathsf{documents} \equiv \langle\rangle$ **then**
39: **let** $s'.\overline{w}.\mathsf{documents} := \langle d \rangle$
40: **else**
41: **let** $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}.\mathsf{documents}.\overline{i}.\mathsf{active} \equiv \top$
42: **let** $s'.\overline{w}.\mathsf{documents}.\overline{i}.\mathsf{active} := \bot$
43: **remove** $s'.\overline{w}.\mathsf{documents}.(\overline{i}+1)$ and all following documents
↪ from $s'.\overline{w}.\mathsf{documents}$
44: **let** $s'.\overline{w}.\mathsf{documents} := s'.\overline{w}.\mathsf{documents} +^{\langle\rangle} d$
45: **stop** $\langle\rangle, s'$
46: **case** XHR
47: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s'), \overline{d}$ **such that** $s'.\overline{d}.\mathsf{nonce} \equiv \pi_2(\mathit{reference})$
↪ $\wedge\, s'.\overline{d} = s'.\overline{w}.\mathsf{activedocument}$ **if possible; otherwise stop**
→ process XHR response
48: **let** $\mathit{headers} := \mathit{response}.\mathsf{headers} - \mathsf{Set\text{-}Cookie}$
49: **let** $s'.\overline{d}.\mathsf{scriptinputs} := s'.\overline{d}.\mathsf{scriptinputs} +^{\langle\rangle}$
$\langle \mathsf{XMLHTTPREQUEST}, \mathit{headers}, \mathit{response}.\mathsf{body}, \pi_3(\mathit{reference}) \rangle$
50: **stop** $\langle\rangle, s'$

- $s_0^p.\mathsf{ids} \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ *(intended to be constrained by instantiations of the Web Infrastructure Model)*
- $s_0^p.\mathsf{secrets} \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ *(intended to be constrained by instantiations of the Web Infrastructure Model)*
- $s_0^p.\mathsf{cookies} \equiv \langle\rangle$
- $s_0^p.\mathsf{localStorage} \equiv \langle\rangle$
- $s_0^p.\mathsf{sessionStorage} \equiv \langle\rangle$
- $s_0^p.\mathsf{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ *(intended to be constrained by instantiations of the Web Infrastructure Model)*
- $s_0^p.\mathsf{sts} \equiv \langle\rangle$
- $s_0^p.\mathsf{DNSaddress} \in \mathsf{IPs}$ *(note that this includes the possibility of using an attacker-controlled address)*
- $s_0^p.\mathsf{pendingDNS} \equiv \langle\rangle$
- $s_0^p.\mathsf{pendingRequests} \equiv \langle\rangle$
- $s_0^p.\mathsf{isCorrupted} \equiv \bot$
- $s_0^p.\mathsf{cibaBindingMessages} \equiv \langle\rangle$
- $s_0^p.\mathsf{tlskeys} \equiv \mathit{tlskeys}^p$ *(see Appendix C.3)*

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

Algorithm 30 Web Browser Model: Main Algorithm.

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

1: **let** $s' := s$
2: **if** $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
3: **let** $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests}\rangle$ → Collect incoming messages
4: **let** $m' \leftarrow d_V(s')$
5: **let** $a' \leftarrow \mathsf{IPs}$
6: **stop** $\langle\langle a', a, m'\rangle\rangle, s'$
7: **if** $m \equiv \texttt{TRIGGER}$ **then** → A special trigger message.
8: **let** *switch* $\leftarrow \{\texttt{script}, \texttt{urlbar}, \texttt{reload}, \texttt{forward}, \texttt{back}\}$
9: **if** *switch* $\equiv \texttt{script}$ **then** → Run some script.
10: **let** $\overline{w} \leftarrow \text{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
11: **let** $\overline{d} := \overline{w} +^{\langle\rangle} \texttt{activedocument}$
12: **call** $\text{RUNSCRIPT}(\overline{w}, \overline{d}, a, s')$
13: **else if** *switch* $\equiv \texttt{urlbar}$ **then** → Create some new request.
14: **let** *newwindow* $\leftarrow \{\top, \bot\}$
15: **if** *newwindow* $\equiv \top$ **then** → Create a new window.
16: **let** *windownonce* $:= v_1$
17: **let** $w' := \langle \textit{windownonce}, \langle\rangle, \bot\rangle$
18: **let** $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
19: **else** → Use existing top-level window.
20: **let** $\overline{tlw} \leftarrow \mathbb{N}$ **such that** $s'.\overline{tlw}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some top-level window.
21: **let** *windownonce* $:= s'.\overline{tlw}.\textit{nonce}$
22: **let** *protocol* $\leftarrow \{\mathsf{P}, \mathsf{S}\}$
23: **let** *host* $\leftarrow \mathsf{Doms}$
24: **let** *path* $\leftarrow \mathbb{S}$
25: **let** *fragment* $\leftarrow \mathbb{S}$
26: **let** *parameters* $\leftarrow [\mathbb{S} \times \mathbb{S}]$
27: **let** *body* $:= \langle\rangle$
28: **let** *startciba* $\leftarrow \{\top, \bot\}$
29: **if** *startciba* $\equiv \top$ **then**
30: **let** $\textit{body}[\texttt{authServ}] \leftarrow \mathsf{Doms}$
31: **let** $\textit{body}[\texttt{identity}] \leftarrow s'.\texttt{ids}$
32: **let** *url* $:= \langle \mathsf{URL}, \textit{protocol}, \textit{host}, \textit{path}, \textit{parameters}, \textit{fragment}\rangle$
33: **let** *req* $:= \langle \mathsf{HTTPReq}, v_2, \mathsf{GET}, \textit{host}, \textit{path}, \textit{parameters}, \langle\rangle, \textit{body}\rangle$
34: **call** $\text{HTTP\_SEND}(\langle \mathsf{REQ}, \textit{windownonce}\rangle, \textit{req}, \textit{url}, \bot, \bot, \bot, a, s')$
35: **else if** *switch* $\equiv \texttt{reload}$ **then** → Reload some document.
36: **let** $\overline{w} \leftarrow \text{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
37: **let** *url* $:= s'.\overline{w}.\texttt{activedocument.location}$
38: **let** *req* $:= \langle \mathsf{HTTPReq}, v_2, \mathsf{GET}, \textit{url}.\texttt{host}, \textit{url}.\texttt{path}, \textit{url}.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
39: **let** *referrer* $:= s'.\overline{w}.\texttt{activedocument.referrer}$
40: **let** $s' := \text{CANCELNAV}(s'.\overline{w}.\text{nonce}, s')$
41: **call** $\text{HTTP\_SEND}(\langle \mathsf{REQ}, s'.\overline{w}.\text{nonce}\rangle, \textit{req}, \textit{url}, \bot, \textit{referrer}, \bot, a, s')$
42: **else if** *switch* $\equiv \texttt{forward}$ **then**
43: **let** $\overline{w} \leftarrow \text{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
44: **call** $\text{NAVFORWARD}(\overline{w}, s')$

→ Algorithm continues on next page.

```
45:     else if switch ≡ back then
46:         let w̄ ← Subwindows(s′) such that s′.w̄.documents ≠ ⟨⟩
              ↪ if possible; otherwise stop    → Pointer to some window.
47:         call NAVBACK(w̄, s′)
48: else if m ≡ FULLCORRUPT then    → Request to corrupt browser
49:     let s′.isCorrupted := FULLCORRUPT
50:     stop ⟨⟩, s′
51: else if m ≡ CLOSECORRUPT then    → Close the browser
52:     let s′.secrets := ⟨⟩
53:     let s′.windows := ⟨⟩
54:     let s′.pendingDNS := ⟨⟩
55:     let s′.pendingRequests := ⟨⟩
56:     let s′.sessionStorage := ⟨⟩
57:     let s′.cookies ⊂^⟨⟩ Cookies such that
          ↪ (c ∈^⟨⟩ s′.cookies) ⟺ (c ∈^⟨⟩ s.cookies ∧ c.content.session ≡ ⊥)
58:     let s′.isCorrupted := CLOSECORRUPT
59:     stop ⟨⟩, s′
60: else if ∃ ⟨reference, request, url, key, f⟩ ∈^⟨⟩ s′.pendingRequests such that
          ↪ π1(dec_s(m, key)) ≡ HTTPResp then    → Encrypted HTTP response
61:     let m′ := dec_s(m, key)
62:     if m′.nonce ≠ request.nonce then
63:         stop
64:     remove ⟨reference, request, url, key, f⟩ from s′.pendingRequests
65:     if binding_message ∈^⟨⟩ m′.body then
66:         let s′.cibaBindingMessages := s′.cibaBindingMessages +^⟨⟩
              ↪ ⟨request.host, m′.body[binding_message]⟩
67:     call PROCESSRESPONSE(m′, reference, request, url, a, f, s′)
68: else if π1(m) ≡ HTTPResp ∧ ∃ ⟨reference, request, url, ⊥, f⟩ ∈^⟨⟩ s′.pendingRequests such that
      ↪ m.nonce ≡ request.nonce then    → Plain HTTP Response
69:     remove ⟨reference, request, url, ⊥, f⟩ from s′.pendingRequests
70:     call PROCESSRESPONSE(m, reference, request, url, a, f, s′)
71: else if m ∈ DNSResponses then    → Successful DNS response
72:     if m.nonce ∉ s.pendingDNS ∨ m.result ∉ IPs
          ↪ ∨ m.domain ≠ s.pendingDNS[m.nonce].request.host then
73:         stop
74:     let ⟨reference, message, url⟩ := s.pendingDNS[m.nonce]
75:     if url.protocol ≡ S then
76:         let s′.pendingRequests := s′.pendingRequests
              ↪ +^⟨⟩ ⟨reference, message, url, v3, m.result⟩
77:         let message := enc_a(⟨message, v3⟩, s′.keyMapping [message.host])
78:     else
79:         let s′.pendingRequests := s′.pendingRequests
              ↪ +^⟨⟩ ⟨reference, message, url, ⊥, m.result⟩
80:     let s′.pendingDNS := s′.pendingDNS − m.nonce
81:     stop ⟨⟨m.result, a, message⟩⟩, s′
```

→ Algorithm continues on next page.

82: **else if** $\exists\, m_{\text{dec}}, k, k', \mathit{inDomain}$ **such that** $\langle m_{\text{dec}}, k\rangle \equiv \text{dec}_{\text{a}}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\texttt{tlskeys}$ **then**
→ For modelling CIBA, we allow the browser to receive requests. By this, the AS can contact its users and ask to give their consent for a given CIBA flow
83: **let** $n, \mathit{method}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}$ **such that**
↪ $\langle \texttt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop**
84: **call** PROCESS_ENCRYPTED_PUSH_MSG$(m_{\text{dec}}, k, a, f, s')$
85: **else if** $m \in$ HTTPRequests **then**
86: **call** PROCESS_PLAIN_PUSH_MSG(m, a, f, s')
87: **stop**

Algorithm 31 Web Browser Model: Processing Encrypted Push Messages

1: **function** PROCESS_ENCRYPTED_PUSH_MSG(m, k, a, f, s') → m is the decrypted 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 browser atomic DY process.
2: **if** m.path ≠ /start-ciba-authentication **then stop**
3: **let** $\mathit{newwindow} \leftarrow \{\top, \bot\}$ → Choose whether to visit AS in new or existing browser window
4: **if** $\mathit{newwindow} \equiv \top$ **then** → Create a new window.
5: **let** $\mathit{windownonce} := \nu_1$
6: **let** $w' := \langle \mathit{windownonce}, \langle\rangle, \bot\rangle$
7: **let** $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
8: **else** → Use existing top-level window.
9: **let** $\overline{\mathit{tlw}} \leftarrow \mathbb{N}$ **such that** $s'.\overline{\mathit{tlw}}.\texttt{documents} \neq \langle\rangle$
↪ **if possible; otherwise stop** → Pointer to some top-level window.
10: **let** $\mathit{windownonce} := s'.\overline{\mathit{tlw}}.\mathit{nonce}$
11: **let** $\mathit{url} := m.\texttt{body[ciba\_url]}$
12: **let** $\mathit{req} := \langle \texttt{HTTPReq}, \nu_{\text{ciba\_req}}, \texttt{POST}, \mathit{url}.\texttt{host}, \varepsilon, \langle\rangle, \langle\rangle, m.\texttt{body}\rangle$
13: **call** HTTP_SEND$(\langle \texttt{REQ}, \mathit{windownonce}\rangle, \mathit{req}, \mathit{url}, \bot, \bot, \bot, a, s')$

Algorithm 32 Web Browser Model: Processing Plaintext Push Messages

1: **function** PROCESS_PLAIN_PUSH_MSG(m, a, f, s') → m is the message, a is the receiver, f the sender of the message. s' is the current state of the browser atomic DY process.
2: **stop** $\langle\rangle, s'$ → Default implementation as no-op.

G.9 Helper Functions

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

CHOOSEINPUT (Algorithm 33). 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$(s', \mathit{scriptinputs})$ is used, where s' denotes the scripts current state. It saves the indexes of already handled messages in the scriptstate s' and chooses a yet unhandled input message from *scriptinputs*. The index of this message is then saved in the scriptstate (which is returned to the script).

CHOOSEFIRSTINPUTPAT (Algorithm 34). Similar to the function CHOOSEINPUT above, we define the function CHOOSEFIRSTINPUTPAT. This function takes the term *scriptinputs*, which as

Algorithm 33 Function to retrieve an unhandled input message for a script.

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

Algorithm 34 Function to extract the first script input message matching a specific pattern.

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

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.

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 $\bot$.

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*.

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

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*.

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

Algorithm 35 Relation of a DNS server R^d.

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

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

and returns its active document's nonce. If there is no such window or no active document, it returns *docnonce*.

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.

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*.

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.

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

G.10 DNS Servers

Definition 81. *A* DNS server *d (in a flat DNS model) is modeled in a straightforward way as an atomic DY process* $(I^d, \{s_0^d\}, R^d, s_0^d)$. *It has a finite set of addresses* I^d *and its initial (and only) state* s_0^d *encodes a mapping from domain names to addresses of the form*

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

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

The relation $R^d \subseteq (\mathcal{E} \times \{s_0^d\}) \times (2^{\mathcal{E}} \times \{s_0^d\})$ of *d* above is defined by Algorithm 35.

G.11 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 82. *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* $p' \neq p$, $p' \in \mathsf{Hon} \cup \mathsf{Web}$. *Hence, a Web attacker cannot listen to traffic intended for other processes. Also, we require that* $A = I^p$, *i.e., a Web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on* I^p*) and may spoof all addresses (i.e., the set* A *may be* IPs*).*

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

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

Finally, E^0 *is an (infinite) sequence of events, containing an infinite number of events of the form* $\langle a, a, \mathtt{TRIGGER}\rangle$ *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.

G.12 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 83 (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:* $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$ *(both containing arbitrary terms),* $\mathit{DNSaddress} \in \mathsf{IPs}$ *(containing the IP address of a DNS server),* $\mathit{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ *(containing a mapping from domains to public keys),* $\mathit{tlskeys} \in \left[\mathsf{Doms} \times \mathcal{N}\right]$ *(containing a mapping from domains to private keys), and* $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$ *(either* $\bot$ *if the server is not corrupted, or an arbitrary term otherwise).*

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

Let v_{n0} and v_{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 36–40, and the main relation in Algorithm 41.

Algorithm 36 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** $s'.\mathtt{pendingDNS}[v_{n0}] := \langle \mathit{reference}, \mathit{message}\rangle$
3: **stop** $\langle\langle s'.\mathtt{DNSaddress}, a, \langle \mathtt{DNSResolve}, \mathit{message}.\mathtt{host}, v_{n0}\rangle\rangle, s'$

Algorithm 37 Generic HTTPS Server Model: Default HTTPS response handler.

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

Algorithm 38 Generic HTTPS Server Model: Default trigger event handler.

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

Algorithm 39 Generic HTTPS Server Model: Default HTTPS request handler.

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

Algorithm 40 Generic HTTPS Server Model: Default handler for other messages.

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

Algorithm 41 Generic HTTPS Server Model: Main relation of a generic HTTPS server

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

G.13 General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [36].

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

Definition 84 (Emitting Events). *Given an atomic process p, an event e, and a finite run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \ldots)$ we say that p* emits *e iff there is a processing step in ρ of the form*

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

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

Definition 85. *We say that a term t* is derivably contained in (a term) *t'* for (a set of DY processes) *P* (in a processing step $s_i \to s_{i+1}$ of a run $\rho = (s_0, s_1, \ldots)$) *if t is derivable from t' with the knowledge available to P, i.e.,*

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

Definition 86. *We say that* a set of processes P leaks a term t (in a processing step $s_i \to s_{i+1}$) to a set of processes P' *if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.*

Definition 87. *We say that a DY process p* created *a message m in a processing step*

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

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

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

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

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

Definition 88. *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 function* PROCESSRESPONSE, *passing the message and the request (see Algorithm 29).*

Definition 89. *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 90. *Let $N \subseteq \mathcal{N}$, $t \in \mathcal{T}_N(X)$, and $k \in \mathcal{T}_N(X)$. We say that k* appears only as a public key *in t, if*

(1) If $t \in N \cup X$, then $t \neq k$

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

Definition 91. *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 92. *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 41).*

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

Lemma 43. *If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b*

(I) emits an HTTPS request of the form

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

(where req is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and

(II) in the initial state s_0, for all processes $p \in \mathcal{W} \setminus \{u\}$, the private key k' appears only as a public key in $S^0(p)$, and

(III) u never leaks k',

then all of the following statements are true:

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

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

(3) The value of the host header in req is the domain that is assigned the public key $\mathsf{pub}(k')$ in the browsers' keymapping s_0.keyMapping (in its initial state).

(4) If b accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and b is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ prior to s_j, then u created the HTTPS response m' to the HTTPS request m, i.e., the nonce of the HTTP request req is not known to any atomic process p, except for the atomic processes b and u.

Proof. **(1)** follows immediately from the preconditions.

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

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

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

The browser is honest in s_i. From the definition of the browser b, we see that the key k is always chosen as a fresh nonce (placeholder ν_3 in Lines 71ff. of Algorithm 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., $\nexists\, s' = (S',E') \in \rho : k \in d_{NP}(S'(p))$).*

Lemma 44. *If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\texttt{windows}$ with the origin $\langle dom, \mathsf{S}\rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then b extracted (in Line 37 in Algorithm 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 $S_i(b).\texttt{keyMapping}[dom] = S_0(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ for p (the key mapping cannot be altered during a run), we can see that *req* was encrypted using a public key that matches a private key which is only (if at all) known to p. With Lemma 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 $\mathcal{WS}$ an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, \mathsf{S}\rangle$ where $S_i(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$*

being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then

- *that request was initiated by a script that b extracted (in Line 37 in Algorithm 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

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

(where req is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and

(II) in the initial state s_0, for all processes $p \in \mathcal{W} \setminus \{u\}$, the private key k' appears only as a public key in $S^0(p)$,

(III) u never leaks k',

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

(V) the instance model defined on top of the HTTPS server does not emit messages in HTTPSRequests*,*

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

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

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

then all of the following statements are true:

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

(2) If there is a processing step $s_j \rightarrow s_{j+1}$ 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 $\mathtt{pub}(k')$ in $S^0(s).\mathtt{keyMapping}$ (i.e., in the initial state of s).

(4) If s accepts a response (say, m') to m in a processing step $s_j \rightarrow s_{j+1}$ and s is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ prior to s_j, then u created the HTTPS response m' to the HTTPS request m, i.e., the nonce of the HTTP request req is not known to any atomic process p, except for the atomic processes s and u.

Proof. **(1)** follows immediately from the 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 $\mathtt{pub}(x)$, 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 41:

- The message emitted in Line 3 of Algorithm 36 has a different message structure
- As s is honest, it does not send the message of Line 6 of Algorithm 41
- 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 v_{n1} in Algorithm 41, is only stored in the *pendingRequests* subterm of the state of s, i.e., in $S^{i+1}(s).\mathtt{pendingRequests}$. Other than that, s only accesses this value in Line 19 of Algorithm 41, 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 41 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 41, 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 36, where a DNS request is sent, and in Line 18 of Algorithm 41, 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 41 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'.

□