CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Formalizing and Evaluating mAuth: Secure Authorization and Authentication Protocol for Native Apps

Master's thesis in Computer science and engineering

JESPER HERMENIUS
SIMON STRÖMBÄCK OLOFSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Formalizing and Evaluating mAuth: Secure Authorization and Authentication Protocol for Native Apps

JESPER HERMENIUS
SIMON STRÖMBÄCK OLOFSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Formalizing and Evaluating mAuth:
Secure Authorization and Authentication Protocol for Native Apps

JESPER HERMENIUS
SIMON STRÖMBÄCK OLOFSSON

Supervisor: Gerardo Schneider, Department of Data Science and AI
Advisors: Miranda Aldrin, Fredrik Hamrefors, Omegapoint AB
Examiner: Romaric Duvignau, Department of Computer and Network Systems

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Formalizing and Evaluating mAuth:
Secure Authorization and Authentication Protocol for Native Apps

JESPER HERMENIUS
SIMON STRÖMBÄCK OLOFSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

OAuth was designed for the browser: its canonical authorisation-code flow assumes a redirect through a system user-agent, an assumption that fits poorly with native mobile applications, where every redirect crosses an OS boundary and breaks the in-app experience. In 2024, Hamrefors and Törnkvist proposed *mAuth*, a redirect-less, mobile-native protocol that preserves OAuth's authorisation guarantees while relocating user authentication into the device's own secure hardware. mAuth supports three authentication ceremonies: a CIBA-style backchannel, a FIDO2 challenge-response and ROPC fallback. Every issued token is bound to a hardware-backed instance key, by means of platform attestation and DPoP. Its original presentation, however, argued for security informally and left several mechanisms underspecified.

This thesis formalizes mAuth against a set of normative standards based around the OAuth architecture, and tightens the underspecified parts. A symbolic model of mAuth using the protocol verification tool *Tamarin* is also constructed. The three theories with authentication variants CIBA, FIDO2 and ROPC are built, with a set of lemmas establishing relevant normative security properties within the symbolic model. The results therefore provide assurance relative to the abstractions and adversary model encoded in Tamarin, rather than a guarantee of security in deployment.

Keywords: Computer Science, Engineering, Thesis, Native App, mAuth, Authentication, Authorization, Security, Tamarin, Formalization.

Acknowledgements

Many thanks to the people that supported us throughout the project:

- Miranda Aldrin
- Fredrik Hamrefors
- Gerardo Schneider
- Romaric Duvignau

Jesper Hermenius, Simon Strömbäck Olofsson, Gothenburg, 2026-06-22

Contents

List of Figures

List of Tables

1

Introduction

1.1 Brief Background

OAuth was released at the end of 2007 to fill a gap in identity-aware web applications: how to let one application access part of a user's account on another service without that user handing over their login credentials. The original specification [1] and its successor OAuth 2.1 [2] have since become one of the standards for delegated authorization on the web.

OAuth was, however, designed for the browser, and its canonical authorization-code flow assumes a redirect through the user agent. That assumption fits unintuitively with native mobile applications, where each redirect crosses an OS boundary, opens a system browser, and exposes the user to an inconsistent experience between different apps. In 2024, Hamrefors and Törnkvist proposed *mAuth* [3], a redirect-less, mobile-native protocol that preserves OAuth's authorization guarantees while moving user authentication into the device's own secure hardware. mAuth supports three user-authentication ceremonies: a CIBA-style backchannel, a FIDO2 challenge-response, and a ROPC fallback. Every issued token is bound to a hardware-backed instance key by means of platform attestation and DPoP.

The original mAuth proposal argues for its security informally and leaves a number of mechanisms underspecified at the level of a real implementation. This work aims to formalize the original mAuth and assess its security through a symbolic prover.

1.2 Problem statement

This thesis asks: *What does mAuth require to be formalized, and once formalized against a set of normative standards, does it satisfy the security properties those standards require?*

1.3 Limitations

UX

The previous thesis discussed the user experience of the protocol. This thesis does not pursue any UX analysis. The focus is instead placed on proving security formally rather than reasoning about it informally.

Formalization exclusions

The formalization focuses on protocol-level properties. The following classes of concern are explicitly out of scope and are not represented in the symbolic model:

- *Implementation-level details* such as programming-language semantics, memory safety, and concurrency issues.
- *Cryptographic implementation vulnerabilities*, including side-channel attacks, timing attacks, and weak randomness sources.
- *Physical attacks* on devices, including hardware tampering and extraction of secrets through physical means.
- *Vulnerabilities in underlying operating systems*, firmware, or hardware security modules beyond the compromise scenarios the model encodes as explicit reveal events.
- *Attacks on trusted components* such as the authorization server or the attestation authority, beyond what is explicitly modeled as a compromise event.
- *Denial-of-service attacks.*
- *Phishing and social engineering.*
- *Correctness of external services and dependencies* not explicitly represented in the formal model. In particular, the attestation verification does not check for revoked certificates.

1.4 Outline

2. Background Establishes the technical groundwork: the cryptographic primitives mAuth uses, the OAuth 2.1 ecosystem and its extensions, the FIDO2 family, the original mAuth proposal, and the symbolic-verification framework in which the formalization is carried out.

3. Formalized mAuth Presents the formalized mAuth protocol. It walks the full 19-step flow alongside its three authentication variants, names the formalizations of the underspecified mechanisms in the original design and lists the new additions.

4. Tamarin Model Describes the Tamarin model: channel model, setup rules, the realisation of each protocol step as one or more Tamarin rules, the action labels and restrictions, the leak-and-reveal compromise events, and the modelling abstractions through which the formalized flow is mapped to the symbolic level.

5. Results Reports the verification results: every lemma's status across each of the three theories.

6. Alternative mAuth Variants Analyses alternative variants of the underlying mechanisms — mTLS in place of DPoP, HTTP Message Signatures in place of DPoP, and verifiable credentials in place of platform attestation.

7. Related work Shines light on protocol's that, similarly to mAuth, keeps the authentication within the native environment.

8. Discussion Addresses three sufficiency arguments: that the adopted standards together produce a coherent specification of the protocol, that the Tamarin model is a faithful symbolic encoding of the formalized mAuth and not an unrelated artefact, and that the proven lemmas cover every security property mandated by the adopted security baselines.

9. Conclusion Summarizes the findings, answers the problem statement and lists directions for future work.

Appendix The appendix is divided into two chapters, the first being an RFC for the mAuth protocol. The second containing the Tamarin-code for all three variants (CIBA, FIDO2, ROPC).

2

Background

This chapter establishes the technical groundwork on which the rest of the thesis builds.

2.1 Cryptographic Primitives

Hash Functions

A hash function is a deterministic algorithm that maps an input of arbitrary length to a fixed-size output called the hash value. SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that converts any input data into a unique, fixed-size 256-bit string.

Symmetric and Asymmetric Cryptography

Symmetric cryptography uses a single shared key for both encryption and decryption. Asymmetric, or public-key, cryptography instead uses a mathematically related key pair: a public key that can be distributed freely and a private key that is held only by its owner. Anything encrypted with the public key can be decrypted only with the private key, and a signature produced with the private key can be verified by anyone holding the public key.

Digital Signatures

A digital signature is a way to digitally show authenticity, integrity, and non-repudiation of data. It can serve as a fingerprint in the digital world to sign that a specific sender really sent the message and that it has not been tampered with. It also means that a party cannot claim they did not sign something once they have signed it.

Public-Key Cryptography

Public-key cryptography relies on two keys for signing/encryption/decryption. One of these keys is publicly known to allow anyone to encrypt a message but only the person with the private key can then decrypt the message and read its contents.

Nonces

A nonce (*number used once*) is a value that is generated freshly for a single use and discarded afterwards.

Password Hashing (Argon2id)

Where a user password is involved, simply computing a cryptographic hash is not enough: the relevant security property is that an attacker in possession of the stored hash cannot brute-force the underlying password in reasonable time. Argon2id [4] is a memory-hard password-hashing function. It is configured with a memory cost, a time cost, and a parallelism parameter that are tuned together so that hashing remains tolerable on legitimate device's while being prohibitive at scale for an attacker.

Elliptic-Curve Cryptography and ES256

The asymmetric algorithm used throughout this protocol is the Elliptic Curve Digital Signature Algorithm [5] (ECDSA) over the NIST P-256 curve, combined with SHA-256 as the message digest. The JOSE algorithm registry identifies this combination as `ES256`. P-256 is chosen because it is the curve mandated by both the iOS Secure Enclave [6] and the Android StrongBox keystore [7], making it the lowest common denominator for hardware-backed instance keys on consumer device's.

2.2 Transport-Layer Security

TLS 1.3

Transport Layer Security version 1.3 (RFC 8446 [8]) is a record-protocol-and-handshake combination that provides confidentiality, integrity, and authenticity between two endpoints.

X.509 Certificates

X.509 is the certificate format used by the public web PKI. A certificate binds a public key to a subject (typically a domain name) and is signed by a Certificate Authority. Certificates are organised into chains that terminate in a root authority that the verifying party already trusts.

Mutual TLS

Standard TLS authenticates only the server to the client. Mutual TLS (mTLS), specified for OAuth in RFC 8705 [9], additionally requires the client to present an X.509 certificate during the handshake, the resulting session is then cryptographically bound to the client's identity.

AEAD Cipher Suites

Authenticated Encryption with Associated Data (AEAD) is a class of symmetric cipher mode that simultaneously provides confidentiality and integrity for a message, and integrity (but not confidentiality) for a small piece of associated data. AEAD is the only mode permitted by TLS 1.3 [8].

2.3 Authentication and Authorization

Authentication

Authentication is the act of verifying that a user is who they claim to be.

Authentication Factors

A factor of authentication is one piece of evidence used to establish identity. The standard categories are knowledge (something the user knows, e.g. a password or PIN), possession (something the user has, e.g. a hardware key), inherence (something the user is, e.g. a fingerprint or face scan).

User Authentication vs Client Authentication

Authenticating a user differs from authenticating a client because the two represent fundamentally different entities within a system. User authentication refers to verifying the identity of an individual person who is interacting with an application. This process ensures that the person is who they claim to be, typically through credentials such as passwords, biometrics, or multi-factor authentication.

In contrast, client authentication refers to verifying the identity of an application or device rather than a human user. A client may be a web application, mobile app, or service that is requesting access to another system, such as an API. Client authentication is commonly implemented using mechanisms such as API keys, client IDs and secrets, or digital certificates. Its purpose is to ensure that only authorized software systems can communicate with protected resources.

While both forms of authentication aim to establish trust, they operate at different levels: user authentication focuses on identifying individuals, whereas client authentication focuses on verifying the legitimacy of systems. In many modern architectures, both types are used together, for example, a client application may first authenticate itself to a server and then authenticate the user on whose behalf it is acting.

Multi-Factor Authentication

Multi-factor authentication (MFA) requires the user to satisfy two or more authentication factors before access is granted, so that the compromise of any single factor is insufficient.

Authorization

Authorization is the act of deciding what an authenticated principal is allowed to do. OAuth [2] defines four roles to make this decision machine-checkable:

Authorization Server (AS)

The AS is the trusted party that authenticates resource owners, authenticates client's, and issues tokens.

Resource Server (RS)

The RS hosts the protected resources or APIs. It does not authenticate user's itself, instead it accepts an access token, verifies the token, and decides whether to release the resource on the basis of the tokens claims.

Client

The client is the application that is requesting access to the protected resource. The client is also the actor that will make API calls when it has been granted access.

Resource Owner (RO)

The resource owner is the one who both owns the data/resources but more importantly, it is responsible for deciding who will have access to the data. This is the person and not the device.

2.4 Tokens and JSON Web Tokens

Token Types

A token is a value that represents a specific authorization decision.

Access Tokens Are short-lived credentials presented to the RS to obtain a protected resource.

Refresh Tokens Are tokens used to obtain new access tokens without requiring the user to re-authenticate.

Sender-constrained tokens Bind a token cryptographically to a key held by the legitimate client. The bearer of the token must prove possession of that key on every use.

Reference tokens A reference token is a high-entropy opaque value that carries no claim of its own, it is a handle by which a token store maps to the real (JWT) access token held server-side. Reference tokens shrink the surface area visible to

user-agent and device-side code, the frontend holds only the reference token, the backend maintains the mapping to the access token.

Sender-constrained vs. bearer tokens A bearer token grants its bearer the rights it names, no further authentication of the holder is required. A sender-constrained token additionally ties to a key, and only a party that can demonstrate possession of that key may use the token.

JSON Web Tokens and JOSE

A JSON Web Token (JWT, RFC 7519 [10]) is a compact, URL-safe representation of a JSON payload protected by a JSON Web Signature (JWS, RFC 7515 [11]) or a JSON Web Encryption (JWE, RFC 7516 [12]). A JWS-protected JWT consists of three base64url-encoded segments: a JOSE *header* identifying the algorithm and key, a *payload* containing claims, and a cryptographic *signature* computed over the concatenation of the encoded header and payload.

Standard JWT Claims

RFC 7519 [10] defines a small set of registered claim names:

`iss` Issuer. The principal that made the JWT.

`sub` Subject. The principal that the JWT is about, typically a user identifier.

`aud` Audience. The recipient the JWT is intended for. A recipient that does not identify itself in the `aud` value must reject the token.

`exp` Expiration time, as a Unix timestamp. The token must not be accepted at or after this instant.

`iat` Issued-at time. Used for freshness checks.

`jti` JWT ID. A unique identifier for the token, used by the verifier to detect replay.

The JOSE Header

The JOSE header of a JWS-protected [11] JWT identifies the signing algorithm and the verification key. The header parameters relevant here are:

`alg` The signing algorithm, e.g. `ES256`.

`typ` The token type. Profiles set distinct `typ` values to prevent cross-context substitution: `at+jwt` for access tokens (RFC 9068 [13]), `dpop+jwt` for DPoP proofs (RFC 9449 [14]), `client-attestation+jwt` for client attestation JWTs.

`jwk` A public key, used when the signer is not previously enrolled at the verifier — for instance, the instance public key carried in the header of a DPoP proof.

The Confirmation Claim (`cnf`)

RFC 7800 [15] defines the `cnf` (confirmation) claim, which lets a JWT name the key whose holder is entitled to use the token. Two forms appear in this protocol:

`cnf.jwk` The full public key in JWK form. Used in the Client Attestation JWT to commit the issued attestation to a specific instance key.

`cnf.jkt` The JWK SHA-256 thumbprint (RFC 7638 [16]) of the bound key. Used in DPoP-bound access tokens (RFC 9449 [14]) so that the resource server can compare the access tokens `cnf.jkt` against the DPoP proofs `jwk` thumbprint on each request.

JWK Thumbprints (RFC 7638)

A JWK thumbprint [16] is a digest of a JSON Web Key computed using a standardized, canonical transformation of selected JWK members. The specification defines which members of a JWK are included in the computation, how those members are serialized into a canonical JSON representation, and how that representation is converted into a UTF-8 byte sequence for hashing. The resulting byte sequence is then hashed using a cryptographic hash function. The output digest can be used as a stable identifier for the JWK, enabling key identification or selection based on the thumbprint value.

JWT-Bearer Client Authentication (RFC 7523)

RFC 7523 [17] defines a client-authentication method in which the client signs a JWT (a *client assertion*) with its registered key and sends it to the AS together with `client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer`. OpenID Connect Core 1.0 [18] standardises this method under the name `private_key_jwt`.

JWT Profile for Access Tokens (RFC 9068)

RFC 9068 [13] pins down the structure of a JWT access token: header `typ=at+jwt`.

2.5 Communication Channels

Front-Channel and Back-Channel

A *front-channel* message travels through the user's browser or device, typically as an HTTP redirect or as content rendered to the user. Front-channel messages are visible to the end user and to any extension or piece of software running in the same context, and therefore cannot carry secrets. A *back-channel* message travels directly between two server-side components over a server-to-server TLS connection, the user does not see it and cannot intercept it.

Redirect-Based and Redirect-Less Flows

A redirect-based flow uses a sequence of HTTP redirects through the user's browser to carry small pieces of state between the client and the AS. The standard OAuth [2] authorization-code flow is the canonical example. A redirect-less flow instead sends the relevant state directly between server components without involving the user's browser. mAuth is redirect-less, the App Frontend never visits the authorization endpoint in a browser.

Authorization Code Exchange

The OAuth authorization code is a short-lived, single-use, opaque value issued by the AS as the result of a successful authorization decision. The client exchanges this code at the token endpoint for the actual tokens. The code itself confers no resource access, it is useful only to the party that can satisfy the token-endpoint authentication requirements.

2.6 Key and Device Trust

Key Lifecycle Management

Key lifecycle management refers to the operational handling of cryptographic keys from generation through destruction.

Key generation Generation must use a cryptographically secure random source. The key type and size are dictated by the intended use.

Key storage Entity that keeps the keys both available and confidential.

Key rotation Periodic rotation limits the value of any single compromised key and the cumulative exposure of long-lived material.

Key revocation Revocation invalidates a key that has been, or is suspected to have been, compromised.

Hardware-Backed Keystores

A hardware-backed keystore generates and uses cryptographic keys inside dedicated hardware rather than in main memory. The private key is never exposed to the operating system, even on a fully compromised device.

Device and Application Attestation

Attestation is a cryptographic claim, made by a trusted party, that a piece of software is running on a particular device, in an expected state, and with a particular cryptographic key. The trusted party is the platform's *attestation authority* (AA):

Google for Android (Play Integrity API [19]) and Apple for iOS (App Attest, part of the DeviceCheck framework [20]). The AA inspects device-integrity signals (bootloader state, root flags, emulator detection) and the calling applications identity (signing certificate digest on Android, App ID on iOS), and, on success, signs an attestation statement that binds those claims to a value chosen by the caller.

2.7 OAuth 2.1 and Surrounding Standards

OAuth 2.1

The OAuth framework authorizes an application to access resources without sharing credentials directly. OAuth 2.1 [2] is an in-progress IETF draft that consolidates OAuth 2.0 [1] with the security best practices that have accumulated around it.

OAuth Authorization Code Flow

The authorization-code grant is OAuth's canonical flow. The client sends the user to the AS authorization endpoint, the AS authenticates the user, obtains consent, and returns a single-use authorization code to a registered redirect URI. The client then exchanges the code at the token endpoint for tokens. The flows strength is that the tokens themselves never traverse the user agent.

PKCE

Proof Key for Code Exchange (PKCE) [21] is an extension of the authorization code flow. The intended use for PKCE is client's such as mobile client's or single page applications. PKCE uses a dynamically generated code verifier as well as a code challenge derived from the verifier to protect the authorization code exchange. The client must then present the original verifier which ensures that only the legitimate client can redeem the authorization code.

Client-Initiated Backchannel Authentication (CIBA)

CIBA [22] decouples user authentication from the calling device. The client POSTs the authentication request directly to the AS at the *backchannel-authentication endpoint*, there is no user-agent redirect at any point. The request carries a user hint (`login_hint`, `login_hint_token`, or `id_token_hint`) that lets the AS identify the resource owner without involving the consumption device, the requested `scope`, optionally `acr_values` naming the required authentication-context class, `binding_message` which the AS displays on the authentication device so the user can correlate the prompt with the action they initiated, and `requested_expiry`. When the client is registered, the request is itself a signed JWT carrying `iss`, `aud`, `iat`, `exp`, and `jti`, signed under the client's registered `backchannel_authentication_request_signing_alg`.

The AS responds with an `auth_req_id` and contacts the user's authentication actor

out of band (e.g, Bank-ID). CIBA defines three modes by which the eventual tokens reach the client, advertised at registration as `backchannel_token_delivery_mode`: `poll` (the client polls the token endpoint), `ping` (the AS notifies a client callback when authentication finishes), and `push` (the AS posts tokens directly to a client callback). This profile uses `poll`.

In `poll` mode the client POSTs the `auth_req_id` to the token endpoint with `grant_type=urn:openid:params:grant-type:ciba` and receives one of: `authorization_pending` while the user has not yet finished, `slow_down` if polling too frequently, `expired_token` on timeout, `access_denied` on user refusal, or a normal token response on success.

Resource Owner Password Credentials (ROPC)

ROPC [1] lets the client send the user's username and password directly to the AS token endpoint. Because the client sees the password in the clear, ROPC violates the principle that the user should authenticate to the AS, not to the client, OAuth 2.1 [2] accordingly removes ROPC from the core specification.

DPoP (RFC 9449)

Demonstration of Proof of Possession (DPoP) [14] is the mechanism by which every issued token is sender-constrained. The client generates an asymmetric key pair (the *instance key*), publishes the public half to the AS through the authorization flow, and signs a fresh *DPoP proof JWT* with the private half on every request to either the AS or an RS.

The DPoP Proof JWT

A DPoP proof is a JWT with JOSE header `typ=dpop+jwt`, `alg=ES256`, and an inline `jwk` carrying the instance public key. The payload carries:

`jti` A unique identifier with at least 96 bits of entropy, used by the server to detect replay.

`htm` The HTTP method of the request the proof is attached to.

`htu` The HTTP URI of the same request, with query and fragment stripped.

`iat` The current timestamp, used for freshness.

`nonce` The most recent value returned by the server in a `DPoP-Nonce` response header.

`ath` For protected-resource requests only: `BASE64URL(SHA-256(access_token))`, binding the proof to a specific access token.

The corresponding access token carries the binding back in the form of a `cnf.jkt` claim equal to the SHA-256 JWK thumbprint of the same `jwk`. A verifier accepts a DPoP-bound request only if the `jwk` in the proof header thumbprints to the access

tokens `cnf.jkt`, `htm` and `htu` match the actual request, `ath` matches the presented access token, `iat` is fresh, `jti` has not been seen before, and `nonce` matches the verifiers last issued `DPoP-Nonce`.

Server-Issued DPoP Nonces (RFC 9449 §8)

A DPoP proof whose freshness rests only on `jti` and `iat` leaves a small replay window between proof creation and expiry. RFC 9449 §8 [14] closes it by letting the server require that the proof carry a `nonce` claim equal to a value the server itself chose. The server publishes the value in a `DPoP-Nonce` response header and may rotate it at any time; a stale or missing `nonce` causes the server to reply with HTTP 401 and `error=use_dpop_nonce`, accompanied by a fresh `DPoP-Nonce`, after which the client retries.

The `dpop_jkt` Request Parameter (RFC 9449 §10)

`dpop_jkt` is an authorization-request parameter that carries the SHA-256 JWK thumbprint of the instance public key. By binding the authorization *request* to the instance key.

FAPI 2.0 Security Profile

The Financial-grade API (FAPI) Security Profile 2.0 [23] is the OpenID Foundations hardening profile for financial deployments. It takes the looser configuration freedom of OAuth and pins it to fewer more secure choices per dimension.

HTTP Message Signatures (RFC 9421)

HTTP Message Signatures [24] let a sender select a set of HTTP message components (method, URI, specific headers, body) and produce a single signature that covers all of them. The signature travels in the `Signature` and `Signature-Input` headers. Compared with DPoP, the signed coverage is broader, where DPoP only signs request metadata, HTTP Message Signatures can also cover the request body.

OAuth Security Best Current Practice (RFC 9700)

RFC 9700 is the most recent OAuth security best-current-practice document. It mandates refresh-token rotation, sender-constrained tokens for public client's, audience-restricted access tokens and more.

2.8 OAuth Discovery, Registration, and Lifecycle

Dynamic Client Registration (RFC 7591)

RFC 7591 [25] lets a client register with an AS by POSTing a JSON document of *client metadata* to a registration endpoint. The metadata fields include `client_id`, `grant_types`, `response_types`, `token_endpoint_auth_method`, `jwks` (or `jwks_uri`),

and any profile-specific extensions. The AS stores the resulting metadata as the long-term contract for that client.

Authorization Server Metadata

RFC 8414 [26] defines a discovery document, served at `/.well-known/oauth-authorization-server` (or, for OpenID Connect [18], `/.well-known/openid-configuration`), that advertises every endpoint and capability of the AS: `issuer`, `token_endpoint`, `introspection_endpoint`, `revocation_endpoint`, `jwks_uri`, the set of supported grant types, response types, client-authentication methods, and signing algorithms. The `signed_metadata` field optionally wraps the document in a JWS so that the discovery document itself can be authenticated end-to-end.

Token introspection [27] is the AS endpoint at which a resource server can ask whether a given token (opaque or JWT) is currently valid. The RS POSTs the token together with its own client authentication and the AS replies with `{active: true, ...}` plus the tokens claims (`scope`, `sub`, `aud`, `exp`, `token_type`), or with `{active: false}`.

Token Revocation (RFC 7009)

RFC 7009 [28] defines a revocation endpoint at which a client can declare that an issued token (refresh or access) should no longer be active. The endpoint accepts a `token` parameter together with client authentication. The AS invalidates the indicated token and, in this protocols profile, the entire associated grant chain on suspected refresh-token reuse.

Issuer Identification (RFC 9207)

RFC 9207 [29] addresses the *mix-up attack*, in which a client that trusts more than one AS can be tricked into routing a response issued by one AS as though it came from another. The mitigation is the `iss` response parameter, which carries the issuers URL on every authorization response and which the client checks against the issuer it learned at discovery.

Resource Indicators (RFC 8707)

RFC 8707 [30] introduces the `resource` request parameter, by which the client tells the AS the canonical URL of the resource server it intends to call. The AS must verify that each requested `resource` value lies in the client's registered set, refusing with `error=invalid_target` otherwise.

Rich Authorization Requests (RFC 9396)

Where `scope` carries authorization as a single opaque string, Rich Authorization Requests (RAR) [31] let the client send a structured `authorization_details` array.

Each entry has a `type` that selects a JSON schema registered for that AS, plus additional fields meaningful to that schema, e.g.

```
{
    "type": "payment\_initiation",
        "instructedAmount": { "currency": "EUR", "amount": "12.50" },
        "creditorAccount": {...}
}
```

The resulting authorization is per-transaction rather than per-scope, which narrows the value of a leaked access token.

2.9 Client Attestation for OAuth

The IETF draft *OAuth 2.0 Attestation-Based Client Authentication* [32] defines a way to present a client's device and app-attestation evidence as a pair of JWTs that the AS can verify uniformly across platforms, regardless of the proprietary attestation format used by the platform underneath. The two JWT `"typ"` values are:

Client Attestation JWT. Header `typ=client-attestation+jwt`, signed by an *attestation provider* that the AS already trusts. It carries the platform attestation evidence and a `cnf.jwk` claim that names the instance key the attestation is bound to.

Client Attestation PoP JWT. Signed by the instance private key referenced in the Client Attestation JWTs. It proves to the AS that the party presenting the attestation actually holds the corresponding private key. The PoP carries `aud`, `iat`, `jti`

The two JWTs travel in the `OAuth-Client-Attestation` and `OAuth-Client-Attestation-PoP` HTTP headers respectively. The draft additionally defines a `challenge_endpoint` from which the client retrieves the attestation nonce that the PoP must echo, and the metadata fields `client_attestation_signing_alg_values_supported` that let the AS advertise which algorithms it accepts for the two JWTs.

2.10 FIDO2 and WebAuthn

Overview

FIDO2 is the umbrella name for two complementary specifications: the W3C *Web Authentication* (WebAuthn) Level 3 API [33] which defines how a relying party invokes the authenticator from a web or native context, and the FIDO Alliance *Client to Authenticator Protocol* (CTAP) 2.2 [34] which defines how the platform talks to an external authenticator.

WebAuthn Assertion Structure

A WebAuthn authentication ceremony exchanges three structured objects:

`PublicKeyCredentialRequestOptions` sent from the relying party to the client. The relevant fields are `challenge` (a fresh server-generated nonce of at least 16 bytes), `rpId` (the relying-party identifier), `allowCredentials` (the list of credential IDs the RP accepts for this user), `userVerification` (`required`, `preferred`, or `discouraged`), and `timeout`.

`clientDataJSON` assembled by the platforms WebAuthn client. It contains `type="webauthn.get"`, the `challenge` reflected back from the request options, and the calling origin. For native applications the origin is the apps origin, bound to the RPs domain via the platform-specific mechanisms described below.

`AuthenticatorData` produced by the authenticator. It contains the `rpIdHash` (= `SHA-256(rpId)`), a flags byte carrying the *User Presence* (`UP`) and *User Verification* (`UV`) bits, and a monotonically increasing `signCount`.

The authenticator signs the concatenation
`authenticatorData` ∥ `SHA-256(clientDataJSON)` with the credentials private key. The RP verifies the signature with the public key it registered for that credential ID and additionally checks that the `type` is `webauthn.get`, that the `challenge` matches the one it issued, that the origin is one it permits, that the `rpIdHash` matches its expected `rpId`, that the `UV` bit is set when user verification was required, and that `signCount` has increased.

2.11 The mAuth Protocol

The mAuth [3] protocol is an authorization and authentication protocol based on the OAuth architecture, designed specifically for native mobile applications. Its primary modification compared to traditional OAuth flows is the removal of browser redirects, which are considered unintuitive in mobile environments.

To preserve security despite the removal of redirects, mAuth incorporates several modern OAuth security mechanisms, including Demonstrating Proof-of-Possession (DPoP) [14], client attestation, hardware-backed key storage, and sender-constrained tokens. These mechanisms bind issued tokens to a specific client instance and cryptographic key pair, reducing the impact of token leakage and replay attacks.

The protocol further supports multiple authentication variants, including FIDO2 [33] authentication, Client-Initiated Backchannel Authentication (CIBA) [22], and a constrained first-party Resource Owner Password Credentials (ROPC) [1] flow.

2.12 Symbolic Protocol Verification

The Dolev–Yao Adversary

The standard symbolic Dolev–Yao [35] adversary controls the network and can read, intercept, delete, inject, reorder, and synthesise any message. It cannot, however, break the primitives themselves, they cannot forge a signature without the private key, cannot guess a fresh value, and cannot extract the contents of a hash.

The Tamarin Prover

Tamarin [36] is an automated symbolic-protocol verification tool. A Tamarin theory consists of:

Rules A rule has the shape `[ premises ] -[actions]-> [ conclusions ]` [37]. A rule fires when its premises are present. On firing, those facts are consumed, the action labels are emitted into the trace (for use with lemmas), and the conclusion facts are added.

Facts *Linear* facts are consumed when a rule fires and represent transient state. *Persistent* facts (written with a leading `!`) are not consumed and represent long-term state. *Fresh* facts `Fr(~x)` introduce values that are guaranteed unique across the entire trace, modelling cryptographic randomness.

Action labels and traces A trace is a sequence of action labels emitted by the firing rules, along with the timepoint at which each was emitted. Lemmas reason about traces.

Lemmas Lemmas are the security properties to be verified. A *universal* lemma (`all-traces`) states that a property holds in every trace. An *existential* lemma (`exists-trace`) states that at least one trace satisfies the property. Lemmas can also be tagged with attributes: `[reuse]` caches the lemma as a free assumption during later proofs, `[sources]` marks a lemma that helps Tamarins source resolver close open chains during pre-computation, `[use_induction]` switches the proof strategy from constraint solving to inductive proof.

Restrictions A restriction is a global axiom imposed on every trace. Restrictions are part of the trusted base: their content is assumed, not proven.

Heuristics Tamarin's automated proof search picks goals according to a heuristic. The `S` (smart) heuristic balances depth and breadth.

Tamarin Syntax

A small example illustrates the syntax used. The rule:

```
rule Client_GenerateKey:
    [ Fr(~skC) ]
  --[ Client_Key_In_Hardware(pk(~skC)) ]->
    [ !Client_Key($CID, ~skC, pk(~skC)) ]
```

generates a fresh client secret `~skC`, emits the action label `Client_Key_In_Hardware` so that lemmas can witness hardware key generation, and adds the persistent fact `!Client_Key` so that subsequent rules can use the same key. The variables prefixed with `$` (e.g. `$CID`) are public constants, those prefixed with `~` (e.g. `~skC`) are fresh values introduced by `Fr`.

Another example of Tamarin syntax, this one being a lemma:

```
lemma S1_client_key_secrecy:
all-traces
"All ClientID k #i. Secret_ClientKey(ClientID, k) @ #i
 ==> not (Ex #j. K(k) @ #j)
     | (Ex #r. RevealClientKey(ClientID, k) @ #r)"
```

This one implying: For all traces in the protocol flow, for the values ClientID and k (key), there exists no time-point (`#j`) where the attacker knows the value k (`K(k)`), except for if a Reveal/Leak event takes place.

ProVerif

ProVerif [38] is the closest alternative to Tamarin. It also operates in the symbolic Dolev–Yao model but compiles protocols to clauses and uses a saturation-based resolution procedure. ProVerif tends to be faster on protocols with simple state but struggles with stateful linearity that mAuth depends on.

2.13 Abbreviations

Below is a list of abbreviation that are used in the thesis work alongside their definitions. The abbreviations are in alphabetical order.

AA Attestation Authority

API Application Programming Interface

AS Authorization Server

ath Access Token Hash (DPoP claim)

BCP Best Current Practice

CAT Client Attestation Token

CIBA Client-Initiated Backchannel Authentication

cnf Confirmation (JWT claim)

DPoP Demonstrating Proof of Possession

ECDSA Elliptic Curve Digital Signature Algorithm

ES256 ECDSA using P-256 and SHA-256

FAPI Financial-grade API Security Profile

FIDO Fast Identity Online

htm HTTP Method (DPoP claim)

htu HTTP URI (DPoP claim)

IETF Internet Engineering Task Force

JOSE JavaScript Object Signing and Encryption

JWE JSON Web Encryption

JWK JSON Web Key

JWS JSON Web Signature

JWT JSON Web Token

mAuth Mobile Authorization Protocol

MFA Multi-Factor Authentication

mTLS Mutual Transport Layer Security

OIDC OpenID Connect

PKCE Proof Key for Code Exchange

PKI Public-Key Infrastructure

PoP Proof of Possession

RAR Rich Authorization Requests

RFC Request for Comments

ROPC Resource Owner Password Credentials

RS Resource Server

rpId Relying Party Identifier (WebAuthn)

SDK Software Development Kit

TLS Transport Layer Security

UP User Presence (WebAuthn flag)

UV User Verification (WebAuthn flag)

VC Verifiable Credential

WebAuthn Web Authentication (W3C)

3

Formalized mAuth

The original mAuth proposal sketches a protocol but leaves a number of mechanisms underspecified. This chapter presents the result of the formalized mAuth. The formalization is constructed as a profile of OAuth 2.1, with each adaptation anchored to a published normative source, being an RFC, a FIDO2/WebAuthn recommendation, or an OpenID Foundation specification.

3.1 Protocol Steps

The formalized mAuth protocol is organized into three phases: *pre-flow operations*, the *main flow*, and *lifecycle operations*. Together, these phases define the complete lifecycle of a client instance, from initial registration and trust establishment to token issuance, protected-resource access, token renewal, and revocation.

3.1.1 Pre-flow operations

(M0) Authorization-server metadata discovery

Performed once by the App Backend, it issues
`GET /.well-known/mauth-authorization-server` [26] to the configured issuer host. The response is a JSON document including:

Common to all variants:

- `issuer` equals the expected issuer URL.
- `token_endpoint`, `introspection_endpoint` (RFC 7662 [27]), `revocation_endpoint` (RFC 7009 [28]), `jwks` (RFC 7517 [39]), and `challenge_endpoint` (`draft-ietf-oauth-attestation-based-client-auth` [32]) are HTTPS URLs.
- `dpop_signing_alg_values_supported` contains `ES256` (RFC 9449 §5.1 [14]).
- `grant_types_supported` contains the union of grants required by the variants this AS deployment offers.
- `authorization_details_types_supported` (RFC 9396 [31]).

- `token_endpoint_auth_methods_supported`, `revocation_endpoint_auth_methods_supported`, and `introspection_endpoint_auth_methods_supported` all contain `private_key_jwt`.
- `client_attestation_signing_alg_values_supported` `client_attestation_pop_signing_alg_values_supported`([32]).
- `authorization_response_iss_parameter_supported` is `true` (RFC 9207 [29])

CIBA variant only:

- `backchannel_authentication_endpoint` is an HTTPS URL (CIBA Core 1.0 [22]).
- `backchannel_token_delivery_modes_supported` contains `poll`.

FIDO2 variant only:

- `authorization_endpoint` is a HTTPS URL — targeted by the FIDO2 variant in step (9) for WebAuthn [33] challenge parameters
- `response_types_supported` — code

ROPC variant only: The ROPC variant introduces no new metadata: it reuses `token_endpoint` from the common set, since RFC 6749 §4.3 [1] carries the password grant directly there.

(M0a) JWKS fetch

The App Backend retrieves the AS signing key set from the `jwks_uri` advertised in the M0 metadata (RFC 8414 §2 [26]), over a TLS 1.3 channel (RFC 8446 [8]) whose server certificate matches the issuer host. These keys verify signatures produced by the AS, e.g the AS-signed access token.

(M1) Client registration

The client sends its desired client metadata to the AS registration endpoint (RFC 7591 [25]), the AS registers the client and returns the registered metadata together with an assigned `client_id`.

Client Sends via Backend — Common to all variants:

- `grant_types` contains `refresh_token` (variant-specific grants are found below).
- `revocation_endpoint_auth_method` `introspection_endpoint_auth_method` `token_endpoint_auth_method` — `private_key_jwt` ([22]).

- `jwks` — the client backend's key set of public keys (RFC 7517 [39]), used for `private_key_jwt` client authentication.
- `dpop_bound_access_tokens=true` (ES256) (RFC 9449 §5.2 [14]).
- `client_attestation_signing_alg_value`, `client_attestation_pop_signing_alg_value` [32] — asymmetric JWS algorithms for the Client Attestation and Client Attestation PoP JWTs.
- `authorization_details_types` — RAR types the client request to use (RFC 9396 [31]).

Client Sends via Backend — CIBA variant only:

- `grant_types` additionally contains `urn:openid:params:grant-type:ciba` (CIBA Core 1.0 §4 [22]).
- `backchannel_token_delivery_mode` — `poll`.
- `backchannel_authentication_request_signing_alg` — an asymmetric JWS algorithm (CIBA Core 1.0).

Client Sends via Backend — FIDO2 variant only:

- `grant_types` additionally contains `authorization_code`.
- `response_types` — `code` only (RFC 6749 §3.1.1 [1]).

Client Sends via Backend —ROPC variant only:

- `grant_types` additionally contains `password` (RFC 6749 §4.3 [1]).
- `first_party` flag — gates eligibility for the ROPC variant, client's without this flag MUST be refused at step (10).

AS responds with:

- `client_id` assigned by the AS.
- The registered client metadata, echoing the values above as accepted.

Flowchart of pre-flow

The figure below (3.1) showcases the sequence of the pre-flow including metadata discovery, jwks fetch and client registration:

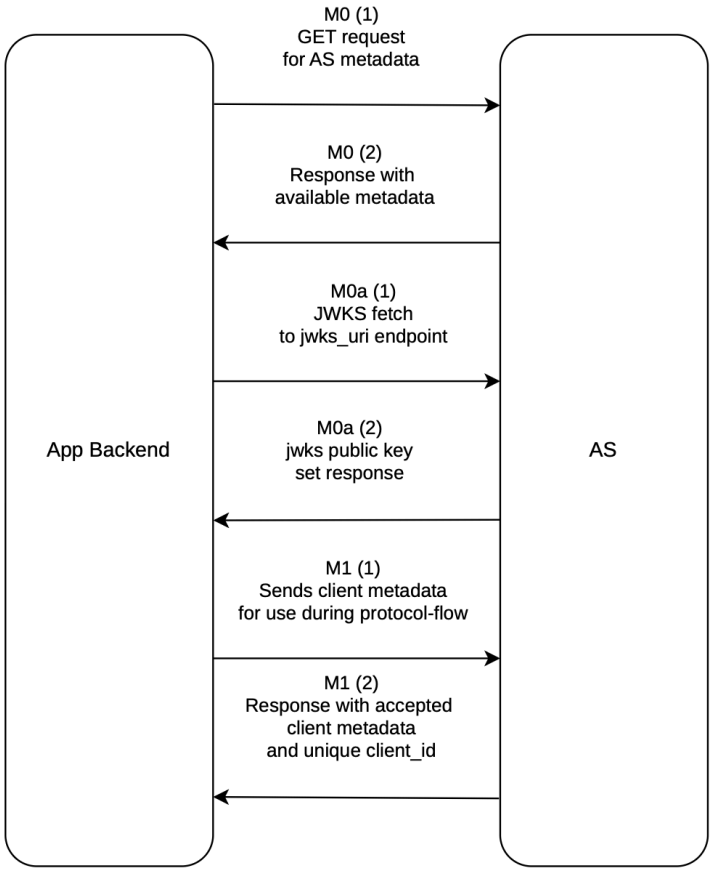

Figure 3.1: Pre-flow sequence.

3.1.2 Main Flow

(1) Login prompt

User prompts the login within the app, for ROPC the username and password credentials are entered.

(2) Attestation-challenge request

Actors: App Frontend → App Backend → AS.
Transport: TLS 1.3, AEAD cipher, forward secrecy

The App Frontend issues an HTTP POST to a frontend-to-backend endpoint carrying `client_id` (RFC 6749 §2.2 [1]). The backend authenticates itself to the AS per the registered `token_endpoint_auth_method` (`private_key_jwt`), following RFC 7523 [17] and RFC 8725 [40] — explicit `typ`, short `exp`, single `aud` equal to the AS issuer, and required claims `iss`, `sub`, `jti`, `iat`. The backend forwards `client_id` to the AS `challenge_endpoint` [32].

The AS validates that `client_id` is registered, that the TLS SNI matches the AS `issuer`, and that the client is permitted to use the attestation-based flow.

(3) Attestation-challenge response

Actors: AS → App Backend → App Frontend.

The AS generates a **cryptographic attestation nonce** of ≥ 128 bits of entropy (FAPI 2.0 Security Profile §5.4 [23]), stored with a short TTL and bound to {`client_id`, issued-at, expiry}. This is the challenge value defined by the `challenge_endpoint` mechanism of `draft-ietf-oauth-attestation-based-client-auth` [32].

In the same response the AS additionally piggybacks an initial **DPoP-Nonce** via the `DPoP-Nonce` HTTP header (RFC 9449 §8 [14]). This pre-arms the frontend so that the first DPoP proof in step (8) carries a valid `nonce` claim without incurring a `use_dpop_nonce` retry. **Distinct nonces.** The attestation nonce and the DPoP-Nonce are independent objects with independent lifecycles: the attestation nonce is single-use and consumed in step (10), the DPoP- Nonce is rotated by the AS at its discretion across the session per RFC 9449 §8 [14] (see step (N)). The response body carries `nonce` and the AS `issuer` identifier (RFC 9207 [29]). The `DPoP-Nonce` travels in the HTTP response header. The App Backend forwards {`attestation_nonce`, `dpop_nonce`, `issuer`} to the App Frontend, which stores `issuer` for cross-checks later on.

(4) Client-instance key pair generation

Actor: App Frontend (device hardware).

The frontend generates a fresh EC P-256 key pair using the platform secure element (Android Keystore [7] with hardware-backed `StrongBox` [41], iOS Secure Enclave with [6] `kSecAttrTokenID = kSecAttrTokenIDSecureEnclave`). The private key is non-extractable. The public key is wrapped in a JWK (RFC 7517 [39]). The key pair is limited to this session.

(5)–(7) Device and app attestation

Actors: App Frontend ↔ Platform Attestation Service.

Step (5). The frontend computes the platform binding value `clientDataHash = SHA-256(attestation_nonce ‖ SHA-256(pk))`, combining the AS challenge with the instance public key. This value is passed into the platform attestation API:

- Android: as the `nonce` field of `IntegrityTokenRequest` (Play Integrity API [19]).
- iOS: as the `clientDataHash` parameter of `DCAppAttestService.attestKey(_:clientDataHash:)` (App Attest [20]).

Step (6). The platform attestation service validates the calling app (Android: signing certificate digest; iOS: App ID), checks device integrity signals (bootloader state, root flags, emulator detection), and produces a signed attestation statement that binds the device, the app identity, and the binding value from step (5).

Step (7). The service returns the attestation statement. The on-the-wire format is platform-specific — Android Play Integrity returns a JWE-wrapped JWS integrity verdict token, Apple App Attest returns a CBOR-encoded attestation object.

The frontend forwards the platform statement (with the instance public key) to the App Backend. The backend wraps it in a **Client Attestation JWT** with `typ=client-attestation+jwt` (`draft-ietf-oauth-attestation-based-client-auth` [32]), signed by the backend's attestation-provider key, with `cnf.jwk` bound to the instance public key, and returns the wrapped CAT to the frontend.

The frontend additionally constructs a **Client Attestation PoP JWT** (`draft-ietf-oauth-attestation-based-client-auth` [32]) signed by the instance private key and carrying {`iss`, `aud`, `iat`, `exp`, `jti`, `nonce`=`attestation_nonce`}.

(8) DPoP proof creation

Actor: App Frontend.

The frontend constructs its DPoP Proof JWT for holding the private key, per RFC 9449 §4.2 [14]. JOSE header: `typ=dpop+jwt`, `alg=ES256`, `jwk=` the instance public key. Payload:

- `jti` — $\geq$ 96 bits pseudorandom (RFC 9449 §4.2 [14]).
- `htm` — `"POST"`.
- `htu` — the AS endpoint URL targeted by the upcoming request (variant-dependent; see step (9)).
- `iat` — current Unix time.
- `nonce` — the DPoP-Nonce piggybacked on the challenge response in step (3).

No `ath` access-token hash is included, as none has been issued. The proof is signed with the instance private key in the secure element.

For ROPC: the credentials entered in step (1) are carried inside this same DPoP proof as an additional `credentials` claim (see step (11b-R)), with `htu=token_endpoint`.

(9)–(10) Authorization request and verification

Step (9). The App Frontend transmits to the App Backend the assembled: Client Attestation JWT and Client Attestation PoP from step (7), the DPoP proof from step (8), the requested `scope`, `resource`, and optional `authorization_details`. The backend then issues a variant-specific request to the AS. All three variants share a common envelope and differ only in the target endpoint and in the user-context material.

Common to all variants:

- Backend client authentication: `private_key_jwt` assertion with `client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer` (RFC 7523 [17]).
- Client Attestation transported via the `OAuth-Client-Attestation` and `OAuth-Client-Attestation-PoP` HTTP headers (`draft-ietf-oauth-attestation-based-client-auth` [32]).
- The `DPoP` HTTP header carrying the proof from step (8).
- `dpop_jkt` request parameter (RFC 9449 §10) equal to the JWK SHA-256 thumbprint (RFC 7638 [16]) of the instance public key.
- `scope`, `resource`, optional `authorization_details` [31].

CIBA variant. Posts `application/x-www-form-urlencoded` to `backchannel_authentication_endpoint` (CIBA Core 1.0 [22]) with specific values:

- One of `login_hint`, `id_token_hint`, or `login_hint_token`.
- `acr_values`, `binding_message`, `requested_expiry`.
- Signed authentication request with claims `iss`, `aud`, `iat`, `exp`, `nbf`, `jti`, signed using the registered `backchannel_authentication_request_signing_alg`.

DPoP and `dpop_jkt` on the CIBA backchannel endpoint are profile-specific extensions of RFC 9449 §10 [14].

FIDO2 variant. Posts to the AS `authorization_endpoint` with the common envelope plus a parameter `auth_method=fido2` indicating that an assertion exchange is requested. No user hint is required at this step — the AS resolves the user from the credential the authenticator selects in step (11b-F2). The AS responds in step (11a-F2) with a `PublicKeyCredentialRequestOptions` [33] structure delivered via the backend.

ROPC variant (first-party only). Posts to the AS `token_endpoint` with the common envelope plus:

- `grant_type=password` (RFC 6749) [1].

- Credentials are *not* sent in the form body. The DPoP proof from step (8) carries them inline as a payload claim `credentials = { username, password_hash }`, where `password_hash` is the Argon2id [4] digest computed on-device with AS-published parameters. The DPoP signature binds the credentials to the instance key.

Step (10). The AS performs, in any order:

1. TLS handshake termination with the server certificate matching the registered `issuer` (RFC 8446 §4.4.2 [8], RFC 9207 [29]).
2. Backend client authentication verification (`private_key_jwt`).
3. DPoP proof verification per RFC 9449 §4.3 [14] (signature, `htm`/`htu` match, `iat` freshness, `jti` uniqueness, `nonce` match against the current DPoP-Nonce).
4. `dpop_jkt` consistency: thumbprint of the DPoP `jwk` equals the `dpop_jkt` parameter (RFC 9449 §10 [14]).
5. `cnf.jwk` in the Client Attestation PoP JWT has the same thumbprint as the DPoP proof `jwk` header.
6. `resource` and `authorization_details` validation: each `resource` URI is in the client's registered set, each `authorization_details` entry `type` is in the client's permitted set (RFC 8707 [30] with `error=invalid_target`).
7. Variant-specific preconditions, for CIBA, the user-hint resolves to a known subject. For FIDO2, the requested `rpId` [33] matches the client's registered relying-party identifier. For ROPC, the client carries the `first_party` flag.
8. Client Attestation verification: signature chain [32] and platform attestation statement verification against Apple/Google root certificates. The AS recomputes `clientDataHash = SHA-256(attestation_nonce ‖ SHA-256(pk))` from the stored attestation nonce for this `client_id` and the instance key `pk` (from the DPoP `jwk` / PoP `cnf.jwk`), and verifies it equals the binding value in the platform statement.

On success, the AS records variant-specific state and proceeds to step (11):

- CIBA: `PendingAuth(auth_req_id, client_id, jkt, scope, resource, authz_details, issuer)`.
- FIDO2: `PendingFIDO2(client_id, jkt, scope, resource, authz_details, issuer, rpId)`; the FIDO challenge is generated and added when the assertion options are issued in step (11a-F2).
- ROPC: no pending state, proceeds to step (11c-R) verification and then direct token issuance at step (11d-R) (not the separate step (12)/(13) exchange).

Any failure yields an error responses (`invalid_dpop_proof`, `invalid_target`, `invalid_authorization_details`, `invalid_request`, `invalid_client`, `invalid_grant`).

(11) User authentication

The user-authentication step is variant-specific. Each variant authenticates the user and yields a variant-specific grant handle — FIDO2 an authorization `code`, CIBA an `auth_req_id`, ROPC none (tokens issued directly). The full per-variant procedures are given in Section 3.1.2.

(12)–(13) Token exchange

Step (12) For FIDO2 and CIBA, the App Backend POSTs to the AS `token_endpoint` (ROPC issues tokens directly at step (11d-R) and does not perform this step). The backend assembles the request and signs its own `private_key_jwt`, the DPoP proof is signed by the frontend in the secure element and returned to the backend:

- Variant-dependent grant:
  `grant_type=authorization_code` with `code` (FIDO2, issued in step (11d-F2)), or
  `grant_type=urn:openid:params:grant-type:ciba` with `auth_req_id` (CIBA).
- `client_id`.
- Backend client authentication via `private_key_jwt`.
- Client Attestation re-presented via `OAuth-Client-Attestation` and `OAuth-Client-Attestation-PoP` headers.
- `resource` and `authorization_details` (must be a subset of, or equal to, what was authorized in step (10), RFC 8707 [30], RFC 9396 [31]).
- The `DPoP` HTTP header carrying a *new* proof signed by the frontend in the secure element: same `jwk`, fresh `jti`, `htm=POST`, `htu=token_endpoint`, `iat=now`, `nonce=` most recent `DPoP-Nonce`. No `ath` (no access token yet).

Step (13) The AS performs:

Common to all variants:

1. Client authentication verification (RFC 7523 [17]).
2. DPoP proof verification (RFC 9449 §4.3 [14]).
3. `dpop_jkt` match against the key bound in step (10) (RFC 9449 §10 [14]).
4. `resource` / `authorization_details` subset check.
5. Each access-token `aud` corresponds to a registered RS canonical URI derived from the validated `resource` set (RFC 8707 [30], RFC 8725 [40]).

CIBA-specific checks: The AS validates the `auth_req_id`: it confirms the identifier was issued to this client (CIBA Core 1.0 [22]) and checks the authorization state. While the user has not yet approved it returns `authorization_pending`, excessively fast polling returns `slow_down`, terminal states are `expired_token` and

`access_denied`. The `auth_req_id` is polled repeatedly and is consumed only on successful issuance.

FIDO2-specific checks: The AS validates the authorization `code` lookup, single-use and TTL.

ROPC-specific checks: No grant handle. ROPC was already targeted at `token_endpoint` in step (9) and verified in step (11c-R), the AS issues tokens directly in that same response.

On success, the AS issues three tokens:

Access Token. JWT format per RFC 9068 [13], signed by the AS, which the Resource Server verifies against the AS JWKS at step (18). Header: `typ=at+jwt`. Claims: `iss`, `exp` (short), `aud` (from `resource`), `sub`, `client_id`, `iat`, `jti`, `scope`, optional `authorization_details`, and `cnf.jkt` (RFC 9449 [14]) equal to the SHA-256 JWK thumbprint of the instance public key.

Reference token. High-entropy ($\geq$ 128 bit, FAPI 2.0 [23]) random string. Stored server-side by the backend, tied to the access token.

Refresh token. Bound by the AS to the same `jkt` (RFC 9449 [14])and to the client instance per `draft-ietf-oauth-attestation-based-client-auth` [32]. Rotation required on every use (OAuth 2.1 [2], RFC 9700 [42]).

The response body follows RFC 9449:
`{ access_token, token_type:"DPoP", expires_in, refresh_token, scope, authorization_details (opt) }`. `token_type` must equal `DPoP`, a `Bearer` response is a misconfiguration and must be rejected by the backend.

(14) Reference-token delivery

Actor: App Backend $\rightarrow$ App Frontend. The backend stores `(reference_token` $\mapsto$ `access_token, refresh_token, jkt, issuer, exp, scope, authorization_details)` in its session store and returns {`reference_token`, `issuer`, `expires_in`} to the frontend. The real access token, refresh token, and DPoP-bound state remain in the backend.
The frontend must verify that the `issuer` value in the response equals the one learned in step (3). A mismatch aborts the session and triggers a revocation call.

(15)–(16) Resource request

Step (15) Before any RS call, the App Frontend prepares the resource-request parameters — `htm` = the method of the upcoming RS call (GET/POST), `htu` = the target RS URL — to be signed as a DPoP proof once `ath` is known. The frontend does not yet hold the access-token value, so it cannot compute `ath`, the DPoP proof is therefore constructed and signed at step (17).

Step (16) The frontend sends to the backend over its TLS channel: {`reference_token`, `request_method`, `request_url`, `request_body`}.

Step (17) The backend:

1. Looks up the reference token, retrieves the real access token and the bound `jkt`.
2. Computes `ath = BASE64URL(SHA-256(access_token))` and asks the frontend to sign a DPoP proof over the request method/URL with the `ath` claim included.
3. Verifies the returned proof per RFC 9449 §4.3 [14], including thumbprint match against the stored `jkt`.
4. Forwards the request to the RS with headers `Authorization: DPoP {access_token}` and `DPoP: {signed_proof}`.

(18) Resource server validation

The RS, aided by introspection via (step (I)), performs:

1. Access-token validation. For JWT access tokens: signature verification against the AS JWKS, `iss` = expected AS, `aud` contains this RS canonical URI (RFC 8725 [40]), `exp`, `scope`/`authorization_details` covers the requested operation, `cnf.jkt` present.
2. DPoP proof validation per RFC 9449 §4.3 [14], including the two protected-resource-specific checks: `ath` equals `BASE64URL(SHA-256(access_token))`, and the DPoP `jwk` thumbprint equals the access-token `cnf.jkt`.
3. Authorization check against `scope` and, when applicable, `authorization_details` matching (RFC 9396 [31]).
4. Replay / nonce check.
5. For introspection-based deployments: introspection response carries `token_type=DPoP` (RFC 9449 [14]).

(19) Resource delivery

On successful validation the RS serves the protected resource. The backend forwards the response body to the frontend. The response does not require its own DPoP signature, response integrity relies on TLS 1.3.

The AS validates the DPoP proof, confirms its thumbprint matches the `jkt` bound to the refresh token and issues a new access token, a rotated refresh token, both bound to the same `jkt`.

If the `scope` parameter is included in a refresh token request, it must be a subset of the originally granted scope.

Step (11) variants:

(11-CIBA) Client-Initiated Backchannel Authentication

Used when authentication must occur out-of-band (e.g., BankID).

(11a-CIBA) Backchannel acknowledgement. The AS responds to the request issued in step (9) with HTTP 200 carrying `auth_req_id`, `expires_in`, and `interval` (CIBA Core 1.0 [22]).

(11b-CIBA) Out-of-band user authentication. The AS contacts the user's authentication app and presents `binding_message` together with the merged `scope`/RAR summary. The user authenticates via BankID and grants or denies consent.

(11c-CIBA) Polling loop. The App Backend POSTs to `token_endpoint` with `grant_type=urn:openid:params:grant-type:ciba` and `auth_req_id`. While authentication is in progress the AS returns `error=authorization_pending`, excessively fast polling yields `slow_down` with a new minimum interval. Terminal errors are `expired_token` and `access_denied`. On success the flow proceeds to step (13). Because CIBA user authentication occurs out-of-band on a separate channel, the binding to the attested instance key is carried by the `auth_req_id`, which is bound to the client's attested key at issuance.

(11-FIDO2) WebAuthn / FIDO2

(11a-F2) Assertion options. The AS responds to the step-(9) request with a `PublicKeyCredentialRequestOptions` structure (W3C WebAuthn Level 3 [33]):

- `challenge` — fresh $\geq$ 16-byte random value.
- `rpId` — the AS registrable domain matched to iOS Associated Domains and Android Digital Asset Links for the app.
- `allowCredentials` — list of `credentialId`s previously registered for the user.
- `userVerification` set to `required`.
- `timeout`.

Delivered to the App Frontend via the backend.

(11b-F2) Native API invocation. The frontend invokes the platform passkey API (Android `CredentialManager.getCredential` with `GetPublicKeyCredentialOption` [7];
iOS `ASAuthorizationPlatformPublicKeyCredentialProvider` [6]).

The platform WebAuthn Client constructs `clientDataJSON` with

`type="webauthn.get"`, the challenge, and the origin (for a native app, the app-origin bound via Associated Domains / Digital Asset Links to the AS `rpId`), passes it via FIDO CTAP 2.2 [34] to the authenticator, which:

1. Looks up the credential private key scoped to `rpId`.
2. Performs user verification, sets the UV flag.
3. Increments `signCount`.
4. Returns `authenticatorData` (rpIdHash, flags, signCount) and a signature over `authenticatorData` ‖ `SHA-256(clientDataJSON)`.

(11c-F2) Assertion response. The frontend forwards the `AuthenticatorAssertionResponse` via the backend to the AS.

(11d-F2) AS verification. The AS performs the WebAuthn Level 3 verification algorithm:

1. `clientDataJSON.type` equals `"webauthn.get"`.
2. `challenge` matches the stored value and has not been previously consumed.
3. `origin` is in the set of origins permitted for the client's app-origin.
4. `rpIdHash` equals `SHA-256(rpId)`.
5. UP flag is set, UV flag is set when `userVerification=required`.
6. `signCount`, if non-zero, is greater than the stored value; otherwise the credential is flagged as potentially cloned.
7. Signature verifies over `authenticatorData` ‖ `SHA-256(clientDataJSON)` against the stored credential public key.

On success, the AS records the authenticated subject and issues an authorization `code` that is returned to the backend; the flow proceeds to step (12).

(11-ROPC) Resource-Owner Password Credentials (deprecated, first-party only)

Per OAuth 2.1 [2], ROPC is removed from the core specification. This profile retains it as a fallback bound to client's carrying the `first_party` flag (M0), using the OAuth 2.0 (RFC 6749) [1] construction wrapped in this profiles attestation, DPoP, and TLS layers. Because ROPC issues tokens in a single request, the step numbering does not reflect execution order: the credential prompt (11a-R) and proof construction occur before the step-(9) request, and token issuance (11d-R) replaces the separate (12)/(13) exchange.

(11a-R) Credential prompt. The App Frontend prompts the user for `username` and `password` inside the apps own UI.

(11b-R) DPoP-wrapped credentials. Rather than transmitting credentials as plain form parameters, the frontend embeds them inside the same step-(8) DPoP proof (already targeted at `token_endpoint`) as an additional payload claim `credentials = { username, password_hash }`, where `password_hash` is the Argon2id [4] digest computed on-device with AS-published parameters. The DPoP signature, combined with TLS 1.3 confidentiality, is the primary protection in transit.

(11c-R) AS verification.

1. Verifies the DPoP proof (RFC 9449 §4.3 [14]).
2. Confirms thumbprint match to the instance key established in step (10) via the Client Attestation `cnf.jwk`.
3. Verifies `password_hash` against the stored credential record for `username`, with a server-side keyed comparison (HMAC over a server pepper).
4. Performs a second-factor challenge (e.g., emailed OTP) before issuing tokens.

(11d-R) Token issuance. On success, because the ROPC request was already targeted at `token_endpoint` in step (9), the AS issues the token bundle of step (13) directly in the same response, with no intermediate grant handle (RFC 6749 [1]). Refusal returns `error=invalid_grant`.

Flowchart of Main flow

Figure 3.2 showcases the (1) login prompt, (2) attestation-challenge request and (3) attestation-challenge response:

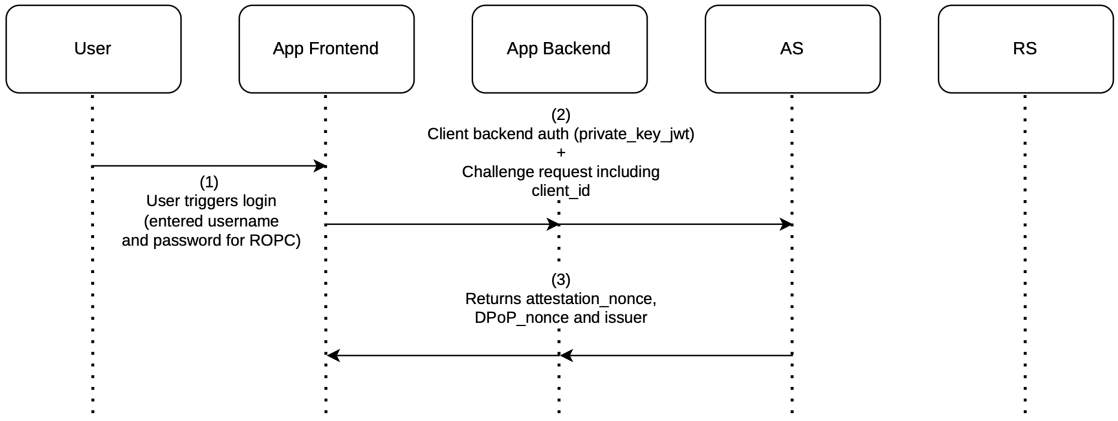

Figure 3.2: Step (1)-(3) of the main flow sequence.

Figure 3.3 showcases the (4) client-instance key pair generation and (5)-(7) device and app attestation:

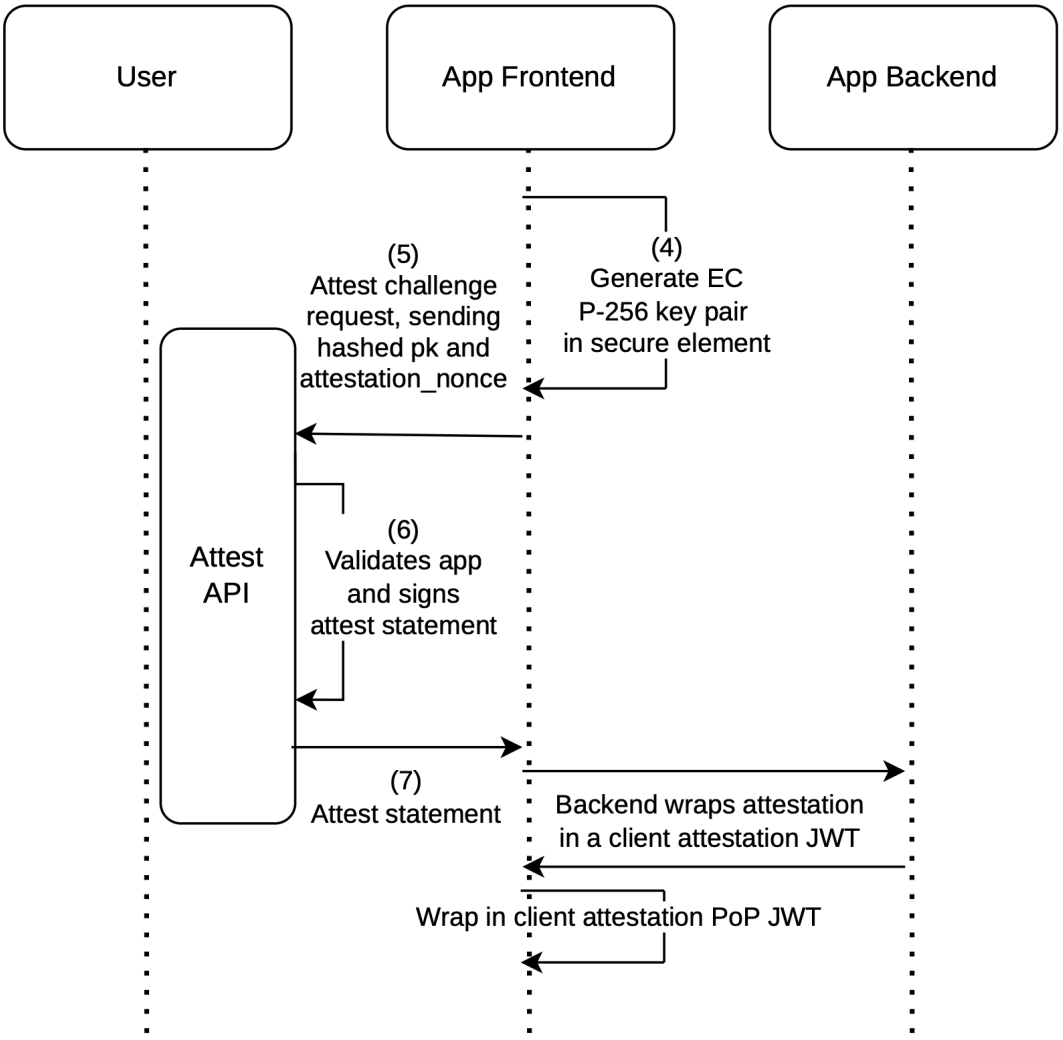

Figure 3.3: Step (4)-(7) of the main flow sequence.

Figure 3.4 showcases the (8) DPoP proof creation and (9)-(10) authorization request and verification:

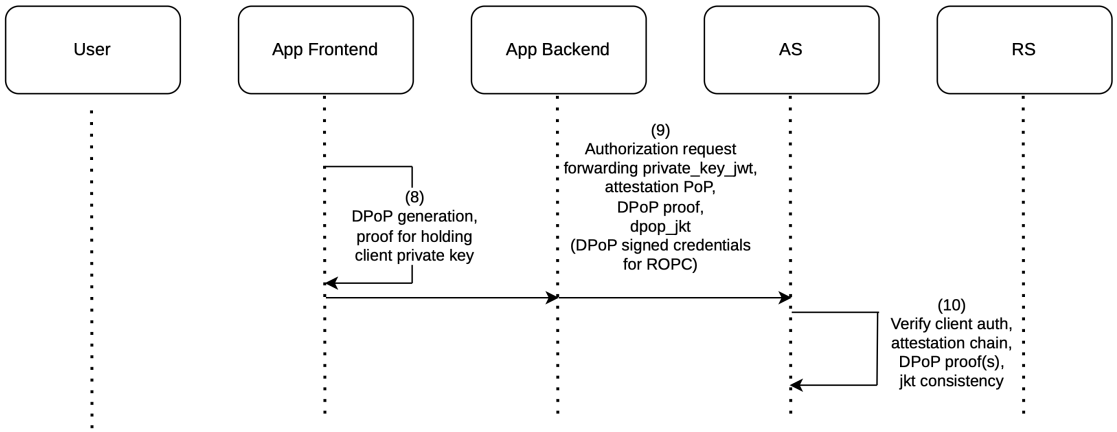

Figure 3.4: Step (8)-(10) of the main flow sequence.

Figure 3.5 showcases (11) user authentication, (12)-(13) token exchange, (14) reference token delivery and (15)-(16) resource request:

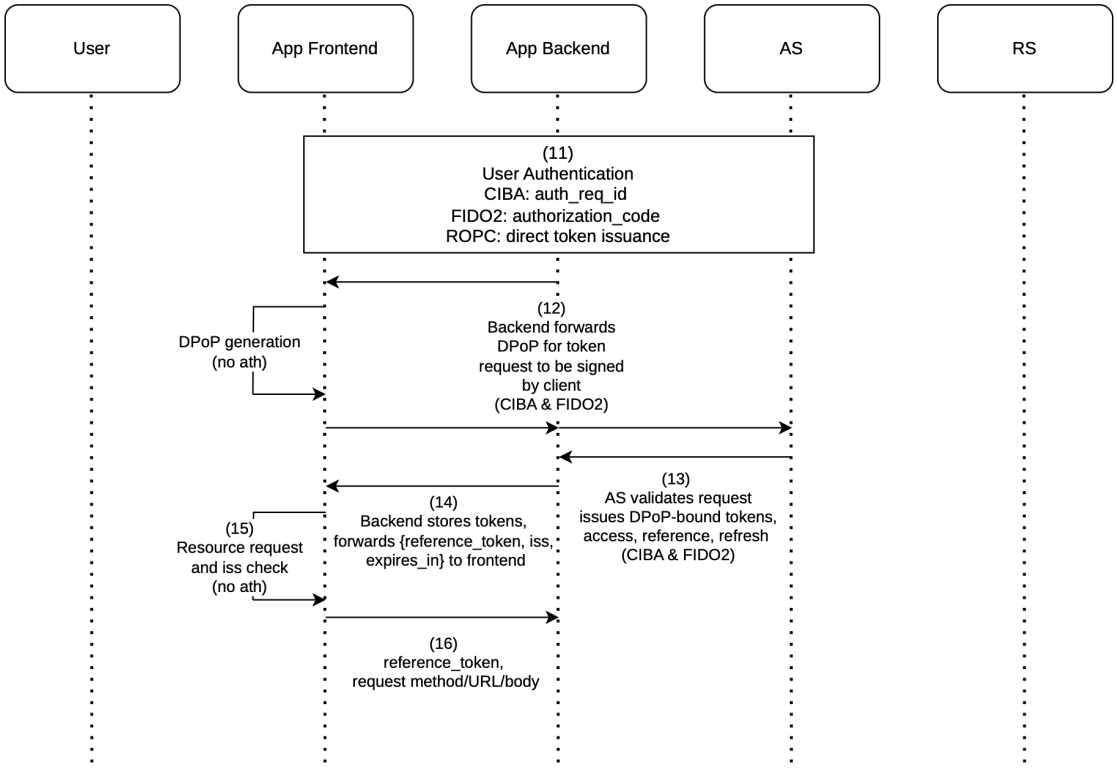

Figure 3.5: Step (11)-(16) of the main flow sequence.

Figure 3.6 showcases (17) resource request, (18) resource server validation and (19) resource delivery:

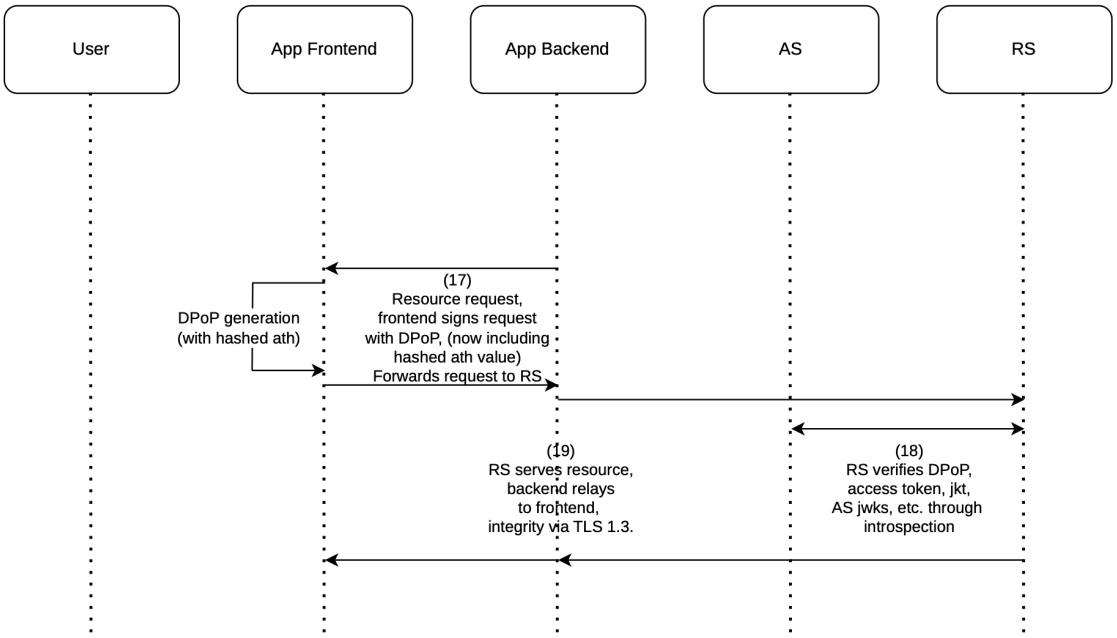

Figure 3.6: Step (17)-(19) of the main flow sequence.

3.1.3 Lifecycle operations

(R) Refresh-token grant

Actors: App Frontend → App Backend → AS.

Invoked when the access token expires or is about to expire. The frontend produces a fresh DPoP proof (same `jwk`, fresh `jti`, `htm=POST`, `htu=token_endpoint`, fresh `iat`, current `DPoP-Nonce`). The backend POSTs to `token_endpoint`:

- `grant_type=refresh_token`.
- `refresh_token` — the server-side-stored value.
- Backend client authentication via `private_key_jwt`.
- Client Attestation re-presented via the `OAuth-Client-Attestation` and `OAuth-Client-Attestation-PoP` headers.
  `draft-ietf-oauth-attestation-based-client-auth` [32] requires that refresh tokens issued under attestation be bound to the client instance, and that the same instance key be presented on refresh.
- `resource` — a subset of originally granted resources (RFC 8707 [30]).
- `scope` — if present, MUST be a subset of the originally granted scope (RFC 6749 [1]).
- `authorization_details` — if present, narrowed per RFC 9396 [31].
- The frontend DPoP proof in the `DPoP` header.

The AS validates the DPoP proof, confirms its thumbprint matches the `jkt` bound to the refresh token, validates the re-presented Client Attestation, and issues a new access token plus a rotated refresh token, both bound to the same `jkt` (OAuth 2.1 [2], RFC 9449 [14]). The previous refresh token is invalidated, reuse triggers the chain revocation defined in step (V).

(N) DPoP-Nonce rotation

The AS issues an initial DPoP-Nonce piggybacked on the attestation-challenge response in step (3) and may rotate it at any point during the session by returning a fresh `DPoP-Nonce` response header (RFC 9449 §8 [14]). On a stale or absent `nonce` claim, the server responds with HTTP 401, `error=use_dpop_nonce`, and a new `DPoP-Nonce` header. The App Frontend updates its stored nonce for that server and reissues the request with a fresh DPoP proof.

(I) Token introspection

Invoked by an RS that does not parse the access token locally. The RS POSTs to the AS `introspection_endpoint` (RFC 7662 [27]):

- `token` — the access-token value.
- Its own client authentication (RS introspection client's are separately registered at the AS).

The AS responds with:

- `active` — boolean.
- `scope`, `client_id`, `username` (ROPC), `sub`, `aud`, `iss`, `iat`, `exp`, `nbf`, `jti`.
- `token_type` — MUST equal `DPoP`.
- `cnf.jkt` — the thumbprint binding the token to its DPoP key. The RS uses this to complete the RFC 9449 §4.3 [14]checks locally.
- `authorization_details` (RFC 9396 [31]).

An inactive token returns `{active: false}`.

(V) Token revocation

Actors: App Backend → AS.

Triggered by:

- User logout in the app.
- Detection of suspicious activity.
- Refresh-token reuse detection.

The backend POSTs to the AS `revocation_endpoint` (RFC 7009 [28]):

- `token` — the refresh token.

- Client authentication via `private_key_jwt`.

If a previously rotated refresh token is presented again, the AS must detect reuse and revoke the entire authorization grant chain associated with that token.

3.2 Formalization of Underspecified Mechanisms

The original mAuth specification [3] describes the protocol at a level of abstraction appropriate for protocol design but leaves several mechanisms under specified. Formal verification in Tamarin requires these mechanisms to be made explicit. The following formalizations resolve ambiguities in the original specification without changing the protocol intended behavior.

1. **Nonce–ClientID binding at the AS.** The original specification states that the AS issues a nonce (Step 3) and later verifies it (Step 10) [3, Section 6.1], but does not specify how the AS remembers which nonce belongs to which client. The formalized model makes this binding explicit: the issued nonce is held together with the issuing ClientID and issuer in a single-use linear fact produced at Step 3 and consumed at Step 10. This both prevents nonce reuse and prevents a nonce issued for one client from being verified against another client's authorization request.

2. **Client key registration at the AS.** The original specification states that the AS verifies the attestation and DPoP at Step 10 [3, Section 6.1] but does not describe whether the AS stores the resulting client public key for later use. The formalized model makes this state explicit: upon successful Step 10 verification, the AS records the (ClientID, public key, issuer) triple as session-long persistent state. Subsequent authentication, token-exchange, and introspection rules consume this fact, ensuring that the same instance key is enforced throughout a session and across all three authentication variants.

3. **Authorization code binding.** The original pseudocode invokes `send_auth_code(auth_code, backend_address)` [3, Appendix B.3] without specifying what the code is bound to. The formalized model issues each authorization code as a fresh value bound to the (ClientID, public key, issuer) triple at the moment of issuance, so that any later redemption is forced to present the same triple. At the model level, this is the carrier for the `dpop_jkt` code binding, without the triple-binding, the modification has nothing to enforce.

4. **CIBA state management.** The original CIBA flow [3, Section 6.3, Figure 6.3] shows the AS issuing an `auth_req_id` but does not specify how the AS tracks which client, key, or issuer it belongs to. The formalized model introduces three persistent bindings produced at ACK issuance and consumed at authentication completion: the `auth_req_id` is bound to its ClientID, to that client's already-attested key, and to the issuer. These bindings ensure that the resulting authorization code inherits the correct triple and prevent a malicious party from completing authentication for one request against a different client's key.

5. **FIDO pending authentication state.** The original FIDO2 flow [3, Section 6.4, Figure 6.4] shows a challenge–response exchange but does not specify how the AS prevents unsolicited responses, nor how the FIDO challenge is correlated with the attestation phase that preceded it. The formalized model adds an explicit pending-authentication state produced at Step 10 and consumed during FIDO verification, parameterised by ClientID, the FIDO nonce, the issuer, and the relying-party identifier. This prevents the adversary from submitting responses for sessions that were never initiated and binds each FIDO exchange to a single attested client.

6. **ISS verification threading.** The original specification mentions issuer checking in its BCP analysis [3, Appendix A, Section 4.4] but the `iss` parameter does not appear in the flow diagrams or pseudocode. The formalized model threads the issuer identifier through the entire protocol: the AS instance is parameterised by a fresh issuer value, the value is recorded by the client at Step 4 as the *learned* issuer, retained through the session, and verified at Steps 15–16 as the *verified* issuer. The client refuses to proceed if the two diverge. This threading is the model-level realization of the RFC 9207 [29] mechanism added.

7. **Pre-flow setup as trusted bootstrap.** The original specification implicitly assumes that the client is registered, that the AS issuer URL is known, and that signing keys are distributed through metadata discovery and JWKS fetch, but does not place these operations in the flow. The formalized model represents these conditions as persistent setup facts established before any flow rule fires: the AS issuer term, the backend long-term key, the AA platform key, and the variant-specific credentials (FIDO credential pre-registration in the FIDO theory, user password storage in the ROPC theory). This makes the boundary explicit and isolates which assumptions are carried into the verification.

3.3 Protocol Modifications

The following changes alter the protocol behavior compared to the original mAuth specification. Unlike the formalizations above, which resolve ambiguities, these modifications introduce different protocol behavior that strengthens the security guarantees.

1. **Two-signature attestation with CAT wrapping.** The original mAuth carries a single attestation artefact: the platform attestation token signed by the attestation authority and returned by the device [3, Section 6.1, Step 7]. We add a second signature layer following the *draft-ietf-oauth-attestation-based-client-auth* [32] pattern: the client forwards the platform attestation to the backend, which wraps it in a Client Attestation Token (`typ=client-attestation+jwt`) signed by the backends long-term key and returns the wrapped token to the frontend. The AS therefore performs three independent verifications at Step 10, platform attestation against the AA root, CAT against the backend public key, and DPoP against the instance key,

rather than the single check implicit in the original.

2. **Explicit attestation-to-key binding.** The original protocol did not specify a concrete construction for binding the instance public key into the attestation request: the Android flow hashed all request parameters including the nonce, while the iOS flow bound the account ID, challenge, and key ID rather than the public key itself. We define a uniform binding value `SHA-256(nonce ‖ SHA-256(pk))` passed to the platform attestation API as the request data (Android: `nonce`, iOS: `clientDataHash`). This ensures the AS can verify, from the attestation statement alone, that the specific instance public key was committed to at attestation time and is bound to the server-issued nonce.

3. **Server-issued DPoP nonces (RFC 9449 §8).** The original protocols DPoP proofs were bound only to client-chosen `jti` and `iat` values. We require the AS and RS to issue `DPoP-Nonce` headers and reject proofs whose `nonce` claim does not match the most recent server-issued value. This removes the replay window that a client-only `jti` uniqueness check leaves open and aligns the protocol with the server-nonce requirement of RFC 9449 §8 [14].

4. **Request-level key binding via `dpop_jkt` (RFC 9449 §10).** The original mAuth attaches the DPoP proof to the authorization request via the HTTP `DPoP` header but does not include the `dpop_jkt` request parameter from RFC 9449 §10 [14]. The formalized model carries `dpop_jkt` as an explicit parameter equal to the JWK SHA-256 thumbprint (RFC 7638 [16]) of the instance public key, and the AS verifies that the thumbprint declared in `dpop_jkt` equals the public key in the proofs `jwk` header. This binds the authorization code itself to the instance key before any token exchange, providing the same one-client-per-grant guarantee that PKCE provides in redirect flows.

5. **Separate FIDO challenge nonce.** The original FIDO2 flow [3, Section 6.4, Figure 6.4] sends a challenge to the client at the start of FIDO authentication but does not specify whether this challenge is the same nonce issued at Step 3 or a fresh value. We require an independent fresh value at Step 10 (the FIDO challenge nonce). This prevents a cross-phase replay in which an adversary observes the attestation nonce on the client–backend channel and replays it into the FIDO flow.

6. **DPoP at the CIBA backchannel authentication endpoint.** The original CIBA integration [3, Section 6.3] does not include a DPoP proof at the backchannel authentication request itself, DPoP appears only at the token endpoint. We require a fresh DPoP proof at the backchannel authentication request, signed by the same instance key bound at Step 10. This aligns the CIBA variant with the per-request proof requirement of RFC 9449 [14] and removes the asymmetry in which only the CIBA authentication step would otherwise be unauthenticated by the instance key.

7. **Two-stage `ath` computation at the resource request.** The original mAuth does not specify how the access-token hash claim (`ath`) is computed at the resource-request DPoP proof, given that the frontend constructs the

proof but does not hold the access token (the backend does). We introduce an explicit two-stage procedure: the frontend produces a request (to become a DPoP proof) bound to the request method and URL but without `ath`, the backend looks up the access token, computes `ath = BASE64URL(SHA-256 (access_token))`, and asks the frontend to sign the proof with the `ath` value inserted before forwarding to the RS. This preserves the principle that only the instance private key signs the proof while still binding the proof to the access token, satisfying the protected-resource-specific checks of RFC 9449 §4.3 [14].

8. **Client-side password hashing and DPoP-binding in ROPC.** The original ROPC variant transmits the password to the AS as a plain form parameter, with TLS as the only protection. We require the password to be hashed on-device with Argon2id [4] using AS-published parameters and to be carried inside a DPoP proof as an additional payload claim rather than as a form field. The AS-stored credential is the same Argon2id digest, so the cleartext password never leaves the device. This cryptographically binds the credential to the proof and instance key, preventing an adversary from resubmitting intercepted credentials with a different DPoP proof.

9. **Audience-restricted access tokens via Resource Indicators.** The original mAuth invoked the principle of least privilege but did not bind tokens to a specific RS. We require the client to send a `resource` RFC 8707 [30] parameter on both the authorization and token requests, and the AS to mint access tokens whose `aud` claim is derived from the validated `resource` value. A token issued for RS_A is therefore cryptographically rejected at RS_B, eliminating the cross-RS replay vector that the original design only mitigated by deployment convention.

10. **Rich Authorization Requests.** We add `authorization_details` (RFC 9396 [31]) as a first-class request parameter, registered per-client at (M0) and validated at the token endpoint as a subset relation. This replaces coarse `scope` strings with structured, per-transaction authorization objects (e.g. a single payment of a specific amount to a specific recipient), reducing the scope of a leaked access token below what scope-only authorization can achieve.

11. **Refresh-token reuse detection with grant-chain revocation.** The original protocol recommended refresh-token rotation but did not specify the response to reuse. We require the AS to detect reuse of any rotated refresh token and revoke the entire authorization grant chain, not just the reused token. Combined with mandatory rotation on every use (OAuth 2.1 §4.3.3 [2]), this converts refresh-token theft from a persistent foothold into a single-use window that takes down the attackers session along with the legitimate user's.

4

Tamarin Model

The formal model is realised as three Tamarin theories, `mAuth_CIBA`, `mAuth_ROPC`, and `mAuth_FIDO`. The three files share an identical backbone (challenge, attestation, DPoP, token exchange, resource access, refresh) and differ only in the variant-specific user-authentication step at position (11) of the protocol flow described in Section 3.1.1. This section describes the structure of the model: the cryptographic primitives, the channel model, the way the rules trace the protocol, and the supporting event labels, restrictions, leak and reveal rules.

4.1 Built-ins and heuristics

The theories declare `builtins: signing` [37] which gives Tamarin the function symbols `sign`, `verify`, `pk`, and the appropriate equation for digital signatures with public-key recovery. This is the only cryptographic primitive needed: every signed object in the protocol (the AA-signed platform attestation, the backend-signed Client Attestation JWT, the client-signed DPoP proofs, and the FIDO assertion) is modelled as a `sign`-term. The heuristic annotation `heuristic: S` selects the *smart* ranking strategy as Tamarins default for goal selection during proof search.

4.2 Channel model

Two channels are used. The standard Tamarin `Out`/`In` pair represents the public, Dolev–Yao-controlled network and is used only by the leak and reveal rules. All protocol-level communication between the client, the backend, the authorization server, and the resource server uses a secure channel modelled by an explicit pair of channel rules:

```
rule ChanOut_S:  [ Out_S($A,$B,x) ] -[ChanOut_S]-> [ Sec($A,$B,x)
]
rule ChanIn_S:     [ Sec($A,$B,x) ] -[ChanIn_S]-> [ In_S($A,$B,x)
]
```

The intermediate `Sec(·,·,·)` fact is *linear* (no `!`-prefix), which corresponds to the variant of secure channel described in the Tamarin manual [37] with the additional property that messages cannot be replayed: each `Sec`-fact is consumed exactly once by `ChanIn_S`. The channel therefore provides confidentiality, authenticity, and fresh-

ness, modelling TLS 1.3 with mutual authentication between hosts. The Dolev–Yao adversary can neither read, modify, inject, nor replay messages on this channel.

4.3 Setup Rules

A small group of rules establishes the protocols long-term state. `Setup_AS_Identity` mints a fresh issuer identifier `!AS_Identity(~iss)`, `Setup_Resource_Server` mints a protected resource `!RS_Resource(~data)`, `Setup_Backend_Identity` generates the backends signing key pair, exposes the public half via `!PkBackend` and `!AS_Knows_Backend` (modelling the AS having the backends public key out-of-band), and labels the secret half with `Secret_BackendKey`. `Setup_AA_Enclave` generates the platform attestation authoritys signing key. In the CIBA and ROPC theories the AA key is generated on first attestation request, and bound to the specific client identifier (the persistent fact `!PkAA` carries the client identifier as an argument). In the FIDO theory the AA key is generated once globally, this difference is discussed in the modelling-abstractions section below. The ROPC theory adds `Setup_User_Credentials` to mint a fresh password and a matching server-side credential record, the FIDO theory adds `Setup_AS_rpId` and `Setup_FIDO_Credential` to mint the relying-party identifier and the user's FIDO key pair, scoped to that relying party.

4.4 Main flow

The 19-step main flow is captured by a sequence of rules whose names follow the naming convention `<Actor>_<Action>_<Direction>_Step<N>`. The flow proceeds linearly:

- Steps (2)–(3) are realised by four rules that walk an attestation challenge from the client through the backend to the AS and back. The AS records the issued nonce with a state fact `AS_Pending_Challenge_Nonce_Useable`, which gates all subsequent rules that consume the nonce.
- Steps (4)–(7) cover hardware key generation, attestation request, AA signature, and forwarding of the resulting platform attestation through the backend, where the backend wraps it into a Client Attestation JWT (CAT) and sends it back to the client. The persistent `!Client_Key` fact carries the client's secret and public key: `Client_Key_In_Hardware` is the action label that the I-series lemmas use to argue about hardware residence.
- Step (8) creates the first DPoP proof by signing fresh `DPoP_Params` with the client key, and step (9) transports the bundle (CAT, platform attestation, DPoP proof, public key, `dpop_jkt`) to the AS via the backend.
- Step (10) is the models central verification rule, `AS_VerifyJWTAndDPoPP_ToBackend_Step10`. Its action facts include the four `Eq(·,·)` equality constraints that demand successful signature verification of the platform attestation, of the DPoP proof, of the CAT, and of the `dpop_jkt`

parameter against the DPoP key. It commits the AS-side trust decision by emitting `AS_Accepted_Client_Key`, `AS_Verified_CAT`, `AS_Verified_PlatformAttest`, `AS_Verified_dpop_jkt`, and `DPoP_Verified`, and binds the client identifier to its public key in the persistent fact `!AS_Client_Tied_PublicKey`.

- Step (11) is the variant-specific user-authentication block. In the CIBA theory it is realised by a chain of rules that represent the CIBA acknowledgement, the user authentication initiated by an out-of-band authentication method, the polling behaviour, and the AS issuing the authorization code only after a successful poll. In the ROPC theory it is a single rule that consumes the user's password and a freshly created DPoP proof bound to that password. In the FIDO theory it is the WebAuthn challenge–response: the AS issues a fresh `FIDONonce`, the client routes it through the FIDO authenticator which signs over the relying-party identifier and the nonce, and the AS verifies the assertion against the registered FIDO credential. All three branches end with the AS issuing an authorization code, recorded by `AuthzCode_Issued` and `AS_Auth_Completed`.
- Steps (12)–(13) realise the authorization-code-for-tokens exchange. A second DPoP proof is created at the token endpoint (with context `'token_te'`), the AS verifies it, marks the code as redeemed, and mints three tokens: a constrained access token, an opaque reference token, and a refresh token. All three are persistent facts (`!ConstrainedAccess`, `!ConstrainedReference`, `!ConstrainedRefresh`) bound to the client's public key.
- Steps (14)–(19) cover delivery of the reference token to the client, construction of a third DPoP proof bound to the reference token, the resource request through the backend to the resource server, RS-driven introspection at the AS, and final delivery of the protected data back to the client.

4.5 Refresh sub-flow

After a successful resource access, the client may invoke the refresh flow via `Client_RequestRefresh`. The refresh rule produces yet another DPoP proof (context `'refresh'`), the backend forwards the stored refresh token together with the proof to the AS, and the AS verifies the proof and emits a `RefreshToken_Used` event. The model intentionally does not produce a replacement access token at this point, the lemma argument in the T- and E-series only needs to know that a refresh exchange requires a fresh DPoP proof bound to the same key.

4.6 Action labels

Each rule emits a small set of action labels that the lemmas use to constrain the trace. The convention is: fresh-issuance facts (`Token_Issued`, `AuthzCode_Issued`, `RefToken_Issued`, `RefreshToken_Issued`) name the fresh value, the client identi-

fier, and the public key. Verification facts (`DPoP_Verified`, `AS_Accepted_Client_Key`, `AA_Signed_Attestation`) record the value being verified together with the key it was bound to, consumption facts (`Token_Used_For_Access`, `RefreshToken_Used`, `AuthzCode_Redeemed`) record the use side of those values. Variant-specific labels (`UserAuth_Initiated` and `CIBA_Code_Issued` in CIBA, `ROPCAuth_Initiated` and `ROPC_Auth_Success` in ROPC, `FIDOAuth_Initiated`, `FIDO_Auth_Success`, `FIDO_Authenticator_Signed`, `FIDO_Credential_Registered`, and `AS_Issued_FIDONonce` in FIDO) carry the additional information the variant-specific lemmas need.

4.7 Restrictions

Restrictions condition the models traces in ways that match the real protocol but are not naturally enforced by the rule structure alone. `Equality` is the standard one that interprets every `Eq(`x`,`y`)` action as an equality constraint, so signature verification can be written inline as `Eq(verify(·),true)`. `unique_dpopp` forces every `DPoP_Used(`p`)` to occur at most once in a trace, encoding the `jti`-based replay protection of RFC 9449. `unique_attestation_per_nonce` forces the AA to sign at most one attestation per nonce. The CIBA theory adds `ciba_single_code_issuance`, which prevents the AS from issuing two codes for the same authentication-request identifier, the FIDO and ROPC theories do not need this because their flows already use a single fresh nonce per ceremony. `no_replay` folds together a number of single-issuance properties on `AS_Request_Accepted`-tagged events, ensuring that the AS does not double-process a challenge, an authorization request, a token-exchange request, a resource-access request, or a refresh request.

4.8 Leak and reveal rules

Compromise is modelled separately from the protocol logic. Three leak rules (`Leak_Access_Token`, `Leak_Reference_Token`, `Leak_Refresh_Token`) take a token from a persistent `!Constrained*` fact and emit it on the public network via `Out`, additionally tagging the trace with the corresponding `LeakToken`, `LeakRefToken`, or `LeakRefreshToken` event. Three reveal rules (`Reveal_AA_Key`, `Reveal_Client_Key`, `Reveal_Backend_Key`) do the same for long-term keys. The ROPC theory adds `Reveal_User_Password`, the FIDO theory adds `Reveal_FIDO_Key`. Each lemma in the S-, T-, I-, R-, and B-series gets a free-standing *escape clause* of the form "`(Ex k #r. Reveal*(k) @ #r)`" or "`(Ex t #l. Leak*(t) @ #l)`" at the right of the implication, which makes the property *conditionally true*: the property holds unless the corresponding compromise event was used. This is the standard Tamarin pattern for stating security against a partially compromised principal.

4.9 Source resolution and induction

The `typing` lemma is annotated `[sources]`, which makes Tamarin use it to resolve open chains during pre-computation. Without it, the theories do not generate sufficient sources for tokens and authorization codes, and the main lemmas time out. Several helper lemmas are tagged `[reuse]` so that Tamarin can cite them as free assumptions during the proof of the main lemmas: two of them (`accept_unique_pkC`, `client_iss_unique`, `access_implies_issued`) additionally use `[use_induction]` to switch the proof procedure to the inductive variant. The sanity lemmas use `hide_lemma` annotations to suppress `[reuse]` helpers during search: this is necessary because the helpers implications go in the wrong direction for an existence proof.

4.10 Practices Not Modeled

The formalization focuses on the core protocol logic of mAuth and deliberately omits a number of practices that the formalized mAuth uses. Most of these omissions concern operational concerns, transport-layer details, or implementation guidance, they are out of scope for a symbolic-protocol analysis or are subsumed by stronger abstractions in the model.

- **Browser- and redirect-based protections** mAuth has no browser redirect step (the App Frontend is a native app talking directly to its own App Backend), so threats such as CSRF, open redirectors, redirect-URI manipulation, and clickjacking are structurally inapplicable.
- **TLS configuration details** Cipher-suite selection, server-certificate validation, certificate pinning, and the TLS handshake itself are abstracted away: the model uses a single secure-channel primitive whose properties (confidentiality, authenticity, no replay) are taken as given. Real-world misconfigurations such as accepting weak ciphers, downgrade attacks, or trusting a corrupt CA are out of scope.
- **Server-side storage security** The session stores held by the App Backend (mapping reference tokens to access tokens and DPoP-bound state) and the credential databases held by the AS are modelled as in-memory persistent facts. Database encryption, access control and recovery from a server-side compromise are not represented.
- **Server-side and back-channel hardening** The AS, the backend, and the RS are trusted endpoints. Compromise of any of these would invalidate large parts of the model and is treated as out-of-scope rather than as an adversary capability.
- **Dynamic client registration (M0)** The pre-flow operation in which the App vendor registers `client_id`, `grant_types`, `jwks`, the `dpop_bound_access_tokens` flag, etc is not modelled. The model instead

starts from a state in which all three principals already share the necessary public-key material via the persistent setup facts.

- **Authorization-server metadata discovery (M1, M1a)** The `.well-known/oauth-authorization-server` document and the JWKS fetch are not represented. The AS issuer identifier and signing keys are simply known to the participants from setup.
- **Token lifetime and revocation (V)** The model does not enforce token expiration or revocation. Once issued, the persistent facts representing access, reference, and refresh tokens remain valid for the entire trace. The revocation flow described in Section 3.1.1 (V), including detection of refresh-token reuse and revocation of the grant chain, is therefore not part of the formal analysis.
- **DPoP-Nonce rotation (N)** The `DPoP-Nonce` server-issued challenge that real implementations rotate on every response is not modelled separately. The freshness it provides is instead delivered by the fresh `DPoP_Params` value that the client signs in every DPoP proof, plus the `unique_dpopp` restriction.
- **Token introspection (I)** Although the resource-server introspection step is present in the rules (`RS_ForwardIntrospection_ToAS`), the introspection response object with its `active`, `scope`, `aud`, `cnf.jkt`, etc., fields is collapsed into the AS verification rule. Differential behaviour for inactive tokens, and the introspection-client authentication scheme, are not represented.
- **CIBA polling timing** The `authorization_pending` response, the `slow_down` response, the `requested_expiry` parameter, and the `binding_message` that real CIBA flows surface to the user are not modelled. The polling pattern is reduced to a single pending-then-authorized transition triggered by the rule `AS_PollAnsAuthorized_ToBackend_CIBAStep5`.
- **FIDO platform-API details** The `clientDataJSON` structure, the `authenticatorData` byte layout, the `rpIdHash` computation, the UP/UV flag bits, and the `signCount` replay-detection counter described in Section 3.1.2 (11-FIDO2) are abstracted into a single `sign(<rpId, FIDONonce>, skFIDO)` term. The model therefore treats every FIDO assertion as fresh and authentic but does not detect signCount-based cloning.
- **Multi-factor authentication in ROPC** The optional secondary OTP factor mentioned in the (11c-R) verification step is not modelled. The ROPC variant proves that authentication requires the password and a DPoP proof bound to the attested instance key, but does not make any claim about a second factor.
- **Argon2id hashing of credentials** The on-device password-hash computation in the (11b-R) step is not represented. The password is modelled as a fresh symbolic value passed inside the DPoP-signed envelope, secrecy of the password is then a consequence of the secure channel and the DPoP signature, not of Argon2id slow-hash properties.
- **Scope, audience, and Rich Authorization Request handling** The model does not distinguish between scopes, between `resource` URIs, or between

`authorization_details` entries. Tokens are bound to a client public key but carry no further authorization claims. Subset checks at the token endpoint (RFC 8707 §2.2 [30], RFC 9396 §7 [31]) are correspondingly absent.

- **Cryptographic algorithm strength** Tamarins symbolic model treats cryptographic primitives as perfect: signatures cannot be forged, fresh values cannot be guessed, and equational theories are exact. Real-world concerns such as algorithm selection, key length, side channels, and entropy quality are outside the symbolic analysis.
- **Implementation-level attacks** Timing side channels, memory-safety bugs, integer overflows, and race conditions in the code that implements the protocol are not represented. The formal claim is about the *protocol*, not about any particular implementation of it.

4.11 Modeling Abstractions

The formal model simplifies certain aspects of the real world protocol in order to maintain tractability and focus the analysis on the protocol core mechanisms.

- **Hardware enclave.** The platform secure element is modelled by storing the client's secret key in a persistent fact `!Client_Key(ClientID, skC, pkC)`. Persistent facts are not transmitted on the public channel, so the Dolev–Yao adversary cannot extract them, compromise is reachable only via the `Reveal_Client_Key` rule. The `Client_Key_In_Hardware(pkC)` action label is emitted at key-generation time and is what the I-series lemmas reason about. The non-extractability and binding-to-attestation guarantees of Android Keystore / iOS Secure Enclave reduce, in the model, to the structural property that the secret never appears in any `Out`-fact except via an explicit reveal.
- **Secure channels in place of TLS.** TLS 1.3 with mutual authentication is collapsed to a single linear secure channel between every pair of communicating principals. The channel rules at the top of each theory implement confidentiality, authenticity, and freshness in a single primitive. The cost is that the model cannot reason about TLS-specific failure modes (downgrade, certificate bypass, session resumption issues). Where the protocol specifies `private_key_jwt` as the `token_endpoint_auth_method`, the model relies on the secure channel to provide the same authentication property.
- **DPoP-proof payload.** The DPoP proof JWT, which in real systems carries `jti`, `htm`, `htu`, `iat`, `nonce`, and (after token issuance) `ath`, is collapsed into a single fresh value `Fr(~DPoP_Params)` signed by the client key. The security-relevant properties of the payload are delegated to other parts of the model, freshness comes from the `Fr`-restriction, replay protection from the `unique_dpopp` restriction (the action `DPoP_Used` fires at most once per parameter), and target-binding from the second argument of the `sign`-term, which is set to a context tag (`'authz'`, `'token_te'`, `'resource'`, `'refresh'`, `'ropc'`) plus the value the proof is binding to the nonce, the access-token reference,

etc.

- **CIBA polling.** The repeated polling loop described in (11d-CIBA) is reduced to a two-state model, a pending-poll rule that fires while authentication has not completed and produces no output facts, and a single authorized-poll rule that fires once the user-authentication method has accepted. The `AS_PollAnsPending_ToBackend_CIBAStep5` rule produces no follow-on facts at all, modelling the "waiting" state in a way that is invisible to subsequent rules. This abstraction does not affect any of the security properties because the lemmas only constrain the ordering and uniqueness of the `poll_authorized` event, not the number of pending polls.

- **Refresh-token replacement.** The refresh rule verifies the DPoP binding and emits the `RefreshToken_Used` event but does not produce a new access token. The successor state `AS_RefreshCompleted` is not consumed by any subsequent rule. This is intentional, the sequence of rules following refresh would be identical to those following the original token issuance (steps 14–19), and the security properties of the refreshed session are therefore identical to those of the initial session. Including the full replacement-token issuance would double the proof effort without changing any conclusion.

- **Device information.** The `~Device_Info` fresh value that the client passes to the AA and that ends up inside the AA-signature payload is treated as an opaque structural placeholder. No rule inspects it, no lemma reasons about it, it is present so that the signed payload has the same shape as a real attestation JWT and so that the AA `AA_NonceUsed` label can be tied to a unique signed term.

- **FIDO ceremony shape.** The WebAuthn protocols `clientDataJSON`, `authenticatorData`, `rpIdHash`, and signature-counter machinery are reduced to a single signature `sign(<rpId, FIDONonce>, skFIDO)`. The `rpId` is itself a fresh symbolic value in the `!AS_rpId` setup fact, so it can stand for any valid relying-party identifier without distinguishing between the formats real implementations use. The `FIDO_Credential_Registered`, `FIDO_Authenticator_Signed`, and `AS_Verified_rpIdHash` action labels carry the rpId explicitly so the FIDO7–FIDO9 lemmas can reason about scoping.

- **Single resource per session.** Each theory has a single resource server with a single fresh data value `~data`. The resource lemmas (E2, E3) are statements about that single data value rather than about resources, this is sufficient because the model is parametric in fresh names and Tamarin will instantiate as many parallel resource sessions as needed during proof search.

- **Issuer-identity opacity.** The issuer identifier is a fresh value created in `Setup_AS_Identity`, it has no structure (no URL, no `.well-known` content) and is unforgeable by being generated by `Fr`. The ISS-series lemmas reason purely about the equality of two such fresh values across rule applications, which corresponds to the intent of RFC 9207 [29] (`authorization_response_iss_parameter`) without modelling the

URL-comparison logic.

4.12 Proof Scope

This section makes explicit who is trusted, what the adversary can do, and what falls outside the formal claim.

4.12.1 Trusted components

The following components are treated as honest by default. Each one has a corresponding compromise rule so that the lemmas can also make conditional claims about behaviour after a partial compromise.

- **Attestation authority (AA).** The AA faithfully signs attestations only when it has been asked to. Signature integrity is provided by the `signing` built-in. Compromise is modelled by the rule `Reveal_AA_Key`, which exposes the private key on the public channel and emits a `RevealAA` event, lemmas in the B- and I-series carry an explicit `RevealAA` that reflects the conditional nature of the attestation guarantees.
- **Hardware enclave.** The client's private key, once generated, lives in a persistent fact and is structurally unreachable on the public channel. Compromise is reachable only through the rule `Reveal_Client_Key`, which emits a `RevealClientKey` event used as an escape clause in S1 and across the T-, R-, K-, and I-series.
- **Authorization server (AS).** The AS executes its own rules honestly, it only emits `AS_Accepted_Client_Key` after the corresponding signature checks succeed, only emits `Token_Issued` after a successful token-exchange verification, and only emits `Token_Used_For_Access` after a successful introspection. There is no compromise rule for the AS, in the present model, an attacker cannot act as the AS.
- **Backend.** The backend is similarly honest. Its private signing key is held in `!Backend_Ltk` and revealed only through `Reveal_Backend_Key`, this gives the B6 lemma its escape clause. The backends role of forwarding messages on the secure channel is modelled by deterministic rules that read on `In_S` and write on `Out_S`.
- **Resource server.** The RS performs its introspection-and-deliver behaviour honestly. There is no compromise rule for the RS, tokens that the RS accepts are by construction the tokens that the AS has authorized.
- **FIDO authenticator.** The FIDO private key is held in `!FIDO_Ltk` and is non-extractable on the public channel. Compromise is modelled by `Reveal_FIDO_Key`, which emits a `RevealFIDO` event used by the FIDO3–FIDO8 lemmas as the escape clause.

4.12.2 Adversary Capabilities

The adversary is the standard Dolev–Yao network attacker augmented with a small set of compromise rules. Concretely, the attacker can do the following [37]:

- **Read and manipulate the public network.** Tamarins built-in `Out/In` channel is fully under the attackers control. The attacker can read every message sent via `Out`, derive new terms by applying the equational theory of the `signing` built-in (in particular, extract public keys from `pk(sk)` and verify signatures with `verify`), and inject any term into `In` that it can construct from its current knowledge. In the present model, only the leak and reveal rules write to `Out`, so the public channel carries compromised material only.
- **Trigger compromise oracles.** The attacker chooses, at any point in the trace, to fire any of the rules `Reveal_Client_Key`, `Reveal_AA_Key`, `Reveal_Backend_Key`, `Leak_Access_Token`, `Leak_Reference_Token`, or `Leak_Refresh_Token`: in the ROPC theory also `Reveal_User_Password`;:in the FIDO theory also `Reveal_FIDO_Key`. Each firing corresponds to a single compromise event of the named principal or token. Lemmas incorporate these as explicit disjuncts in their conclusions.
- **Run unbounded parallel sessions.** The attacker may cause arbitrarily many concurrent instances of every rule to fire, on arbitrarily many freshly chosen principals and identifiers. Tamarins multiset-rewriting semantics is inherently concurrent and unbounded, the lemmas are quantified over all traces, which include any interleaving of sessions.
- **Act as a legitimate participant.** The attacker may register as a client (the rules are quantified over fresh `ClientID`s), generate its own client key, obtain its own attestation, and run the protocol to completion to obtain its own tokens. The lemmas guarantee that any such tokens are bound to the attackers own key and cannot be used to access resources on behalf of a different client.
- **Combine compromised material.** Anything the attacker has on the public channel can be combined into new terms under the equational theory of the built-in. In particular, a leaked access token can be presented to the resource server in an introspection request.

The attacker explicitly *cannot* do the following, as a consequence of the channel model rather than of any restriction.

- **Read messages on the secure channel.** The `Sec(·,·,·)` fact is linear and confidential, there is no rule that copies its contents into `Out`, and the equational theory has no operator that extracts the contents of a `Sec`-fact. Confidentiality on the protocol channel is structural.
- **Modify messages on the secure channel.** Tampering would require constructing a different `Sec`-fact; only `ChanOut_S` produces `Sec`-facts, and only the legitimate sender can fire it for a given pair of agents. Authenticity on the protocol channel is structural.

- **Inject fabricated messages.** For the same reason, the attacker cannot place a `Sec`-fact onto the multiset, `ChanIn_S` can only consume facts that `ChanOut_S` has previously produced.
- **Replay messages on the secure channel.** The `Sec`-fact is linear (no `!`-prefix), so once it has been consumed by `ChanIn_S` on a given branch, it cannot be consumed again. This corresponds to the "secure channel" variant described in the Tamarin manual [37], with the modification of a linear `Sec` fact.

4.12.3 Scope Limitations

- **Server compromise.** Neither the AS, the backend, nor the RS is modelled as compromisable beyond the long-term signing-key reveals that the corresponding lemmas already handle (`Reveal_Backend_Key` for the backend, no reveal rule for the AS or the RS). A fully compromised AS would falsify almost all lemmas trivially and is therefore out of scope.
- **Secure-channel attacks.** All protocol communication between the client, backend, authorization server, and resource server uses the same secure-channel primitive. Network attacks on these channels are therefore not part of the model.
- **Token scope and audience.** Tokens are modelled as fresh values bound to a client public key, they do not carry `scope`, `aud`, or `authorization_details` fields. The privilege-restriction guidance from the OAuth BCP (RFC 9700 [42]) is therefore not formally evaluated, although the models tighter binding of tokens to client keys (T-series and K-series lemmas) is in some respects stronger than what scope restrictions alone would provide.
- **Cryptographic strength.** The model uses the symbolic abstraction of the `signing` built-in, questions of algorithm choice, key size, entropy, and side-channel resistance are out of scope.
- **User behaviour.** The user is modelled implicitly that completes the variant-specific authentication when invited to. Phishing resistance, social engineering, and credential reuse are not in scope.

4.13 Tamarin Lemmas

The security properties to be verified are built as lemmas. These lemmas serve as building blocks that specify the conditions that must hold for all protocol executions under the defined attacker and trust assumptions. Each lemma captures a specific security guarantee. Proving these lemmas establishes that the protocol satisfies its intended security objectives within the symbolic model. Snippets of code for certain lemmas are included below, the ones not included can be found in the appendix (B).

Lemmas are constructed from the oAuth RFC 9700[42]. Security considerations in DPoP RFC [14] is also utilized to construct lemmas.

4.13.1 Helper Lemmas

The helper lemmas are not security goals, they are auxiliary facts that Tamarin reuses while proving the main goals. They either guide source resolution (the `[sources]` annotation) or expose a structural invariant of the protocol that simplifies later proofs (the `[reuse]` annotation).

`typing`

The typing lemma closes Tamarins source resolver by stating that freshly generated values cannot reach the attacker out of thin air. If the attacker knows an access token, reference token, or refresh token, that knowledge must be explained by an explicit leak event:

```
  lemma typing [sources]:
" (All t cid pk #i #j.
      Token_Issued(t, cid, pk) @ #j & !KU(t) @ #i
      ==> (Ex #l. LeakToken(t, pk) @ #l & #l < #i))
& (All t cid pk #i #j.
      RefToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
      ==> (Ex #l. LeakRefToken(t, pk) @ #l & #l < #i))
& (All t cid pk #i #j.
      RefreshToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
      ==> (Ex #l. LeakRefreshToken(t, pk) @ #l & #l < #i))
& (All c cid pk iss #i #j.
      AuthzCode_Issued(c, cid, pk, iss) @ #j & !KU(c) @ #i
      ==> F)
"
```

`token_implies_code`

Whenever an access token has been issued, an authorization code for the same client and key was issued earlier:

```
  lemma token_implies_code [reuse]:
"All t cid pk #i. Token_Issued(t, cid, pk) @ #i
 ==> (Ex c iss #j. AuthzCode_Issued(c, cid, pk, iss) @ #j & #j < #i)"
```

The following `implies` lemmas use the same structure.

`code_implies_accept`

Every authorization code, in turn, was preceded by the AS accepting the client's public key.

access_implies_issued

Any token that is used to access a resource must have been issued beforehand.

refresh_used_implies_issued

Refresh-token issuance implies a prior issuance.

accept_unique_pkC

For a fixed client identifier, the AS accepts at most one public key.

client_iss_unique

A given client learns at most one issuer identity (and learns it exactly once).

L_unique_dpopp_proof

A theorem-form restatement of the `unique_dpopp` restriction: no DPoP proof is ever consumed twice.

L_unique_attestation_proof

Likewise the theorem form of `unique_attestation_per_nonce`: the AA signs at most one attestation per nonce, and that attestation binds a unique client public key.

4.13.2 Sanity Lemmas

The sanity lemmas (correctness checks) are existence statements. Their job is to demonstrate that the protocol model is not unsatisfiable, if every legitimate event were ruled out by some restriction, every all-traces lemma would hold vacuously and the results would be meaningless. Each of these lemmas exhibits a single honest trace in which the named event occurs, which together demonstrate that the protocol can run end-to-end as intended.

The first six lemmas:

- `SANITY_keygen`
- `SANITY_token_issued`
- `SANITY_dpop_verified`
- `SANITY_token_used`
- `SANITY_refresh_used`
- `SANITY_resource_received`)

Each witness a single key event along the protocol flow. Respectively, the client learning the issuers identity, the AS issuing an access token, a DPoP proof being verified, a token being used at the RS, a refresh token being redeemed, and the client finally receiving the protected resource.

`SANITY_full_chain` is the strongest sanity lemma: it exhibits a single trace in which the entire ordered sequence of events occurs. The exact sequence is variant-specific (CIBA inserts the user-auth and poll events, ROPC inserts the password-based authentication, FIDO inserts the FIDO credential ceremony) and matches the corresponding flow:

```
lemma SANITY_full_chain:
exists-trace
"Ex data ClientID nonce pkC code token auth_req_id iss
    #t1 #t2 #t3 #t4 #t5 #t6 #t7 #t8 #t9 #t10 #t11.
   AS_Issued_Nonce(nonce)                                  @ #t1
 & Client_Sent_Attestation_Request(ClientID, nonce, pkC) @ #t2
 & AA_Signed_Attestation(nonce, pkC)                       @ #t3
 & Backend_Issued_CAT(ClientID, pkC, nonce)              @ #t4
 & AS_Accepted_Client_Key(ClientID, pkC)                 @ #t5
 & UserAuth_Initiated(auth_req_id)                       @ #t6
 & AS_UserAuthAccepted(auth_req_id)                      @ #t7
 & AuthzCode_Issued(code, ClientID, pkC, iss)            @ #t7
 & CIBA_Code_Issued(auth_req_id)                         @ #t8
 & Token_Issued(token, ClientID, pkC)                    @ #t9
 & Token_Used_For_Access(token, ClientID, pkC)           @ #t10
 & Client_Recourse_Received(data, ClientID)              @ #t11
 & #t1 < #t2 & #t2 < #t3 & #t3 < #t4 & #t4 < #t5
 & #t5 < #t6 & #t6 < #t7 & #t7 < #t8 & #t8 < #t9
 & #t9 < #t10 & #t10 < #t11"
```

The above code showcases the full-chain lemma for the CIBA variant.

4.13.3 Secrecy (S)

The S-series lemmas state that long-lived secrets and freshly made tokens remain confidential as long as the corresponding compromise event is not invoked. Each lemma takes the same shape: the attacker either does not learn the secret, or there exists a matching reveal event that explicitly granted it.

`S1_client_key_secrecy.`

The client's private signing key is secret. The only way the attacker can learn it is via the explicit `RevealClientKey`.

```
lemma S1_client_key_secrecy:
all-traces
"All ClientID k #i. Secret_ClientKey(ClientID, k) @ #i
 ==> not (Ex #j. K(k) @ #j)
     | (Ex #r. RevealClientKey(ClientID, k) @ #r)"
```

The followins (S) lemmas use the same structure as above.

`S2_access_token_secrecy.`

Access tokens are secret unless explicitly leaked through the `Leak_Access_Token` rule.

`S3_reference_token_secrecy.`

The same property for the opaque reference token that the backend exchanges with the AS for introspection.

`S4_aa_key_secrecy.`

The attestation authoritys signing key is secret, unless leaked.

`S5_refresh_token_secrecy.`

Refresh tokens are secret unless explicitly leaked.

`S6_backend_key_secrecy.`

The backends signing key (used to make client-attestation JWTs) is secret unless explicitly revealed.

4.13.4 Authorization Correctness (Z)

The Z-series lemmas establish the integrity of the authorization-code issuance and token exchange.

`Z1_code_after_attestation.`

Every authorization code is preceded by a complete attestation chain for the client's public key: the AS issued a nonce, the AA signed an attestation over that nonce and the client key, and these events occurred in the correct order before the code was issued. The only escape clause is a compromised AA.

```
1  lemma Z1_code_after_attestation:
2  all-traces
3  "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
   iss) @ #i
4   ==> (Ex nonce #j #k.
5     AS_Issued_Nonce(nonce) @ #j
6     & AA_Signed_Attestation(nonce, pkC) @ #k & #j < #k & #k < #i)
7       | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
```

Z2_no_double_redeem.

An authorization code is redeemed at most once. Even if the attacker intercepts the code in transit, they cannot get it accepted at the token endpoint after the legitimate client has already used it.

Z3_no_auth_code_substitution.

If a code was issued for client C_1 with key pk_1 and is later redeemed claiming to be for client C_2 with key pk_2, then $C_1 = C_2$ and $pk_1 = pk_2$. The redeemer cannot swap in a different client identity or a different key.

Z4_code_unique_issuance.

The same authorization code is never issued twice.

4.13.5 Token Binding and Proof-of-Possession (T)

The T-series lemmas express the central DPoP guarantee: tokens are bound to a specific client key, and possession of the token alone is not sufficient to use it.

T1_stolen_token_unusable.

If the attacker leaks an access token and the token is later used at the RS, then the genuine client must have created and the AS must have verified a DPoP proof for the matching key in between. In other words, exfiltrating the token does not, by itself, give the attacker resource access, the attacker still needs to produce a DPoP proof, which requires the private key. The escape clause is a compromised client key.

```
lemma T1_stolen_token_unusable:
all-traces
"All token pkC ClientID #i #j.
   LeakToken(token, pkC) @ #i
   & Token_Used_For_Access(token, ClientID, pkC) @ #j
   ==> (Ex params ctx #k #l.
          Client_Created_DPoP(pkC, params,ctx) @ #k
          & DPoP_Verified(params, pkC) @ #l
          & #k < #l & #l < #j)
       | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
```

T2_token_access_requires_pop.

Every successful resource access is accompanied by a DPoP verification for the same key the AS originally accepted, and the AS-acceptance event precedes the DPoP verification.

T3_no_token_identity_crossing.

A token issued to client C_1 with key pk_1 cannot be accepted as belonging to a different client or key. Tokens stay tied to the (client, key) pair they were minted for.

T4_stolen_refresh_token_unusable.

A leaked refresh token can only be used if the legitimate client has also created a DPoP proof for the matching key, or if the client key itself is compromised. Possession of the refresh token alone is not enough.

T5_stolen_refresh_needs_dpop.

Extension of T4: the DPoP proof for the refresh-token exchange must have been both created by the client and verified by the AS, and the verification coincides with the refresh-token exchange. This rules out reuse of a DPoP proof minted for some other purpose.

4.13.6 Replay Resistance (R)

R1_dpop_requires_creation

Every DPoP verification at the AS is preceded by the client actually creating that DPoP proof. There is no way for the attacker to fabricate a DPoP proof from scratch, the only escape clause is a compromised client key. Combined with the `unique_dpopp` restriction (which prevents replay of the same proof), this establishes that DPoP proofs are both authentic and used once.

```
lemma R1_dpop_requires_creation:
all-traces
"All params pkC #i. DPoP_Verified(params, pkC) @ #i
 ==> (Ex ctx #j. Client_Created_DPoP(pkC, params, ctx) @ #j & #j < #i)
     | (Ex ClientID skC #r. RevealClientKey(ClientID, skC) @ #r)"
```

4.13.7 Attestation Binding (B)

The B-series lemmas link the AS-side trust decisions to events that must have happened at the AA and inside the platform.

B1_attestation_unforgeable

Every AA-signed attestation is preceded by the client requesting that attestation for the same nonce and the same key. Without compromising the AA signing key, the attacker cannot synthesise an attestation binding an arbitrary key to a nonce of their choosing.

```
lemma B1_attestation_unforgeable[reuse]:
all-traces
```

```
    "All nonce pkC #i. AA_Signed_Attestation(nonce, pkC) @ #i
     ==> (Ex ClientID #j. Client_Sent_Attestation_Request(ClientID,
    nonce, pkC) @ #j & #j < #i)
         | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
```

B2_as_requires_attestation

The AS only accepts a client public key after a matching attestation has been signed.

B3_token_implies_attestation

Token issuance implies a complete attestation chain in the correct order: a signed attestation, followed by the AS accepting the client key, followed by the token issuance.

B4_attestation_key_binding.

If the same nonce appears in two AA-signed attestations and both keys are accepted by the AS, then the two keys must be identical. Together with the `unique_attestation_per_nonce` restriction, this means a nonce is bound to at most one client key. The attacker cannot attach a fresh nonce to multiple keys to broaden their reach.

B5_attestation_nonce_single_use

Each attestation nonce is used at most once at the AA. This prevents attestation replay.

B6_cat_unforgeable

If the AS verifies a client-attestation token (CAT), the backend must have issued a CAT for the same client and key earlier, unless the backends signing key has been revealed. This forces the attacker through the backend, which is itself protected by lemma S6.

B7_dpop_jkt_matches_instance_key

If the AS accepts a DPoP-bound key thumbprint, the client must have sent an attestation request with that exact key. The DPoP-jkt parameter cannot be inflated to refer to a key the client never claimed.

4.13.8 Key Continuity (K)

The K-series lemmas state that the same client public key flows unchanged through the entire protocol, attestation, AS acceptance, token issuance, DPoP verification, and resource access.

K1_key_continuity_attestation_to_token

If the AS accepts pk_A for a client and later issues a token to the same client bound to pk_T, then $pk_A = pk_T$, unless the client key has been revealed.

```
lemma K1_key_continuity_attestation_to_token:
    all-traces
    "All ClientID pkC_accept token pkC_token #i #j.
       AS_Accepted_Client_Key(ClientID, pkC_accept) @ #i
       & Token_Issued(token, ClientID, pkC_token) @ #j
       & #i < #j
       ==> pkC_accept = pkC_token
           | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
```

K2_key_continuity_token_to_access

The key recorded at token issuance equals the key presented at resource access. The token does not rebind to a different key in between.

K3_dpop_key_matches_accepted

At the moment the RS verifies a DPoP proof for a token-bearing request, the key in the proof matches the key that the AS accepted for that client. There is no key drift between authorization and use.

4.13.9 Hardware-Bound Key Continuity (I)

The I-series lemmas establish that attested hardware-generated client keys remain continuously bound to authorization decisions, issued tokens, and subsequent resource access.

I1_token_requires_hardware_key.

Every issued token traces back to a hardware-placed key: the key appeared in a `Client_Key_In_Hardware` event before the AS accepted it, and that acceptance preceded the issuance. Compromise of either the client key or the AA key are the only escape clauses.

```
    lemma I1_token_requires_hardware_key [reuse]:
  all-traces
  "All token ClientID pkC #i.
    Token_Issued(token, ClientID, pkC) @ #i
    ==>
      (Ex #j #k.
         Client_Key_In_Hardware(pkC) @ #j
         & AS_Accepted_Client_Key(ClientID, pkC) @ #k
         & #j < #k & #k < #i)
      | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)
      | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
```

I2_access_requires_hardware_key.

The same property extended through to resource access: a token used at the RS was issued to a key that originated in hardware on the client device.

4.13.10 End-to-End (E)

The E-series lemmas span the entire protocol from authentication to resource delivery.

E1_token_requires_authentication

Token issuance is preceded by a successful AS authentication of the client. No silent token issuance is possible.

```
lemma E1_token_requires_authentication:
all-traces
"All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
 ==> (Ex #j. AS_Auth_Completed(ClientID) @ #j & #j < #i)"
```

E2_resource_matches_issued_token.

If the client receives data, then the RS sent that data tied to a specific token, and that token was issued in advance. The data the client gets is the data the RS released against a real token.

E3_resource_requires_fresh_dpop.

Resource delivery to the client is preceded by a freshly created DPoP proof, verified by the AS, for a key that the AS had previously accepted for that client.

E4_refresh_requires_issuance.

Refresh-token use implies a prior matching issuance event. This is the end-to-end version of `refresh_used_implies_issued`.

4.13.11 Issuer Mix-Up (ISS)

The ISS-series lemmas address mix-up attacks, where an attacker tries to confuse a client into accepting tokens from the wrong AS.

ISS1_consistency.

The issuer the client verifies on the response side equals an issuer it had previously learned. The client does not accept an issuer it has never seen.

```
lemma ISS1_consistency:
all-traces
"All ClientID iss #i.
```

```
4       Client_Verified_Iss(ClientID, iss) @ #i
5       ==> (Ex #j. Client_Learned_Iss(ClientID, iss) @ #j & #j < #i)"
```

ISS2_no_mix_up.

If a code was issued under issuer iss_1 and the resulting token is issued under iss_2, then $iss_1 = iss_2$. The pipeline does not splice issuers between code and token.

ISS3_token_origin.

Every token issued under a given issuer traces back to an authorization code issued under the same issuer.

ISS4_client_rejects_wrong_iss.

If the client first learned issuer iss_l and later verifies an issuer iss_v, then $iss_l = iss_v$. The client refuses to verify an issuer different from the one it originally bound to.

ISS5_refresh_preserves_iss.

A refresh-token redemption is consistent with the issuer of the original code: there must exist an authorization code for the same client, key, and issuer. Refresh does not change which AS the session belongs to.

4.13.12 Flow (F)

The F-series lemmas state that the AS event flow respects the intended ordering of phases.

F1_resource_access_requires_token_exchange.

A resource-access event at the AS is preceded by a token-exchange event. Resources cannot be accessed via a path that bypasses the token endpoint.

```
1     lemma F1_resource_access_requires_token_exchange:
2     all-traces
3     "All params #i.
4        AS_Request_Accepted(<'resource_access', params>) @ #i
5        ==> (Ex code #j.
6               AS_Request_Accepted(<'token_exchange', code>) @ #j
7               & #j < #i)"
```

F2_token_exchange_requires_authorization.

A token-exchange event is preceded by an authorization event. The token endpoint cannot be invoked without a prior authorization decision.

F3_authorization_requires_challenge.

An authorization event is preceded by a challenge event. The whole flow starts at the challenge endpoint, never elsewhere.

4.13.13 CIBA-specific Lemmas

The CIBA variant introduces a decoupled user-authentication step in which the AS issues an authentication request identifier (`auth_req_id`) and the client polls until the user has approved the request. The CIBA lemmas verify that this extra layer is sound.

CIBA1_authcode_after_user_auth.

Every authorization code in the CIBA flow is preceded by a user authentication for some `auth_req_id`, and the `auth_req_id` is bound to that exact code. There is no path to a code without a prior user decision.

```
lemma CIBA1_authcode_after_user_auth:
all-traces
"All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
iss) @ #i
 ==> (Ex auth_req_id #j.
        UserAuth_Initiated(auth_req_id) @ #j
        & CIBA_AuthEvent_Bound(auth_req_id, code) @ #i
        & #j < #i)"
```

CIBA2_complete_after_request.

The AS only marks authentication complete for a client after the client made a CIBA request for an `auth_req_id` bound to that client, and the user authentication for that `auth_req_id` took place. CIBA split between request and authentication does not let unsolicited authentications slip through.

CIBA3_auth_req_id_unique.

A given `auth_req_id` is accepted at most once. CIBA tokens of intent are one-shot.

CIBA4_auth_event_binding.

A given `auth_req_id` is bound to at most one authorization code. The two are in a one-to-one relationship.

CIBA5_code_single_issuance.

Each `auth_req_id` causes the AS to issue a code at most once. Re-polling does not produce a second code.

CIBA6_no_unsolicited_auth.

If a user authentication is accepted for an `auth_req_id`, some client previously made the CIBA request that produced that identifier. The AS will not honor an authentication for an `auth_req_id` that no client ever asked for.

CIBA7_poll_authorized_implies_user_auth.

The AS only returns a positive poll response after the user has actually authenticated.

CIBA8_code_requires_poll.

A CIBA code is only released after the client polled for it. The AS does not push codes preemptively.

CIBA9_auth_req_id_bound_to_client.

A given `auth_req_id` is associated with at most one client identity. Two different client's cannot both claim ownership of the same identifier.

4.13.14 FIDO-specific Lemmas

The FIDO-specific lemmas check that the WebAuthn-style assertion is correctly bound to the protocol session.

FIDO1_authcode_after_init.

Every authorization code in the FIDO flow is preceded by the client initiating FIDO authentication for some relying-party identifier (`rpId`). The code does not appear without a prior FIDO ceremony.

```
1    lemma FIDO1_authcode_after_init:
2    all-traces
3    "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
     iss) @ #i
4     ==> (Ex rpId #j. FIDOAuth_Initiated(ClientID, rpId) @ #j & #j < #i)"
```

FIDO2_complete_after_init.

Authentication completion at the AS implies that the client initiated FIDO authentication. As with the other variants, there is no completion without a matching initiation.

FIDO3_fido_key_secrecy.

The FIDO authenticators private key is confidential unless the authenticator is explicitly compromised.

FIDO4_credential_required_for_auth.

Every successful FIDO authentication implies the existence of a FIDO secret for the client, or an explicit authenticator compromise. There are no credential-less FIDO successes.

FIDO5_origin_binding.

A FIDO authentication for a relying-party identifier is preceded by a FIDO secret existing for the same client. The temporal ordering rules out post-hoc credential injection.

FIDO6_auth_binds_to_attested_client.

Successful FIDO authentication is preceded by the AS accepting a client key for that client.

FIDO7_credential_rpId_scoping.

If a FIDO credential was registered for relying party r_1 and the authenticator later signed for relying party r_2 using that same credential, then $r_1 = r_2$.

FIDO8_as_enforces_rpId_binding.

A successful FIDO authentication for a relying party implies that a FIDO credential was registered for that very relying party. The AS itself enforces the scoping, not just the authenticator.

FIDO9_fido_nonce_single_use.

A FIDO challenge nonce uniquely determines the client and relying party it was issued for, and is issued at most once. This prevents replay and cross-binding of FIDO challenges.

4.13.15 ROPC-specific Lemmas

The ROPC variant carries a password through the protocol. The ROPC-specific lemmas check that this password is handled safely and that the resulting authorization is bound to the same attested client that produced it.

ROPC1_authcode_after_init.

Every authorization code in the ROPC flow is preceded by the client initiating ROPC authentication. The flow has a clear starting point.

```
1  lemma ROPC1_authcode_after_init:
2  all-traces
3  "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
   iss) @ #i
4   ==> (Ex #j. ROPCAuth_Initiated(ClientID) @ #j & #j < #i)"
```

ROPC2_complete_after_init.

The AS only completes authentication after the client initiated ROPC. There is no path that completes authentication without the corresponding client-side initiation.

ROPC3_password_secrecy.

The user's password is confidential to the attacker unless the explicit reveal rule fires. The protocol does not leak the password.

ROPC4_auth_requires_credentials.

Successful ROPC authentication implies prior use of the user's password by the same client (or a password reveal). There is no authentication-success path that bypasses credentials.

ROPC5_ropc_requires_dpop.

ROPC authentication requires DPoP verification for a key that has been accepted for the client beforehand. Possession of the password alone is not enough, the request must also be DPoP-bound.

ROPC6_auth_binds_to_attested_client.

Every successful ROPC authentication is preceded by the AS accepting a client key for that client. Authentication only succeeds for client's that have already passed attestation.

ROPC7_password_not_leaked_by_protocol.

If the protocol used a password, the attacker does not learn it unless either the password or the client key was explicitly revealed. The DPoP envelope around the password binding is what makes this work.

ROPC8_dpop_params_bind_password_use.

ROPC authentication success implies that the credentials use and the DPoP verification both happened, with the credentials use coming first. Password use and DPoP proof are temporally interlocked.

5

Lemma Results

The results generated from the constructed Tamarin lemmas are listed below. The complete verification outputs can be found within the Github repository [43].

5.1 Verification Methodology

5.1.1 Tool, Heuristic, and Reproducibility

All three theories share the same attestation, DPoP, token-exchange, and resource-access. They differ only in the user-authentication step (Step 11). For each theory, the formal model, the attacker model, the trusted base of restrictions, and every lemma are presented together with the verification status. All lemmas listed below were discharged automatically by Tamarin under the heuristic `S`.

5.1.2 Verification Methodology

The three theories were verified with the Tamarin Prover. Each theory file declares `heuristic: S` and uses the automatic prover. No interactive proof guidance was required: every lemma was verified automatically once the helper lemmas (those tagged `[reuse]` or `[sources]`) were in place.

5.1.3 Attacker Model and Channel Abstraction

The verification is carried out under the standard symbolic Dolev–Yao adversary: the attacker controls the public network, may derive any term from messages they have observed (modulo the equational theory of the `signing` builtin), and may inject arbitrary messages onto public channels. The attacker cannot break cryptographic primitives modelled as perfect.

Beyond the public network, the model uses an explicit secure-channel abstraction realised by the linear fact `Sec`, produced exclusively by `ChanOut_S` and consumed exclusively by `ChanIn_S`. The attacker cannot read its content (no rule emits it via `Out`), cannot forge it (no other rule produces it), and cannot replay it (linearity enforces single consumption). Because the fact is linear rather than a public-key encryption term, the model deliberately abstracts TLS-protected legs of the protocol away from the TLS handshake itself: confidentiality, integrity, and single-delivery

are guaranteed; cryptographic details of TLS are not. Attacks at the TLS layer are by design out of scope.

All long-term keys, tokens, and credentials are shielded behind explicit *compromise rules*, compromise must be modelled by firing the corresponding `Reveal` or `Leak` rule.

5.1.4 Restrictions

Restrictions are global axioms imposed on every trace. They form part of the trusted base of the verification: their content is assumed, not proven. Following restrictions are used: `Equality` (for signature verification), `unique_dpopp` (each DPoP parameter set consumed at most once), `no_replay` (the AS deduplicates accepted requests by their identifier), `unique_attestation_per_nonce` (the attestation authority signs at most one attestation per challenge nonce), and `ciba_single_code_issuance` (CIBA only — each `auth_req_id` is bound to at most one issued authorization code).

5.2 Protocol Behavior Lemmas

The generated proofs are found on the Github repository [43].

5.2.1 Helper Lemmas

The `typing [sources]` lemma closes Tamarins source resolver. The `[reuse]` helpers (`token_implies_code`, `code_implies_accept`, `access_implies_issued`, `refresh_used_implies_issued`, `accept_unique_pkC`, `client_iss_unique` carry causal-order and functional-uniqueness facts used by the main lemmas. Two further helpers (`L_unique_dpopp_proof`, `L_unique_attestation_proof`) restate two of the restrictions as theorems. All helper lemmas were verified in all three theories.

5.2.2 Sanity Lemmas

The seven sanity lemmas (`SANITY_keygen`, `SANITY_token_issued`, `SANITY_dpop_verified`, `SANITY_token_used`, `SANITY_refresh_used`, `SANITY_resource_received`, `SANITY_full_chain`) are stated as `exists-trace`. Each exhibits an honest execution that reaches a particular state, ruling out vacuous satisfaction of the corresponding `all-traces` lemmas. The `SANITY_full_chain` lemma exhibits a single trace witnessing every key event of the protocol in causal order, from challenge issuance through resource delivery.

Status.

All seven sanity lemmas verified in all three theories.

5.2.3 Secrecy (S)

`S1_client_key_secrecy.`

The client's instance private key remains unknown to the attacker unless `Reveal_Client_Key` is fired.

`S2_access_token_secrecy.`

A constrained access token remains unknown to the attacker unless explicitly leaked.

`S3_reference_token_secrecy.`

A DPoP reference token remains unknown to the attacker unless explicitly leaked.

`S4_aa_key_secrecy.`

The AA private key remains unknown to the attacker unless `Reveal_AA_Key` is fired.

`S5_refresh_token_secrecy.`

A refresh token remains unknown to the attacker unless explicitly leaked.

`S6_backend_key_secrecy.`

The backend's long-term signing key remains unknown to the attacker unless `Reveal_Backend_Key` is fired.

Status.

All six lemmas verified in all three theories. Variant-specific secrecy lemmas (FIDO key, user password) are reported in the variant sections.

5.2.4 Authorization Correctness (Z)

`Z1_code_after_attestation.`

Every authorization-code issuance is preceded by a fresh challenge nonce and a corresponding attestation signed under that nonce, unless the AA key has been revealed.

`Z2_no_double_redeem.`

No authorization code is ever redeemed twice.

`Z3_no_auth_code_substitution.`

If a code is issued for a particular ClientID and key, any later redemption uses exactly the same ClientID and key.

`Z4_code_unique_issuance.`

An authorization code is issued at most once.

Status.

All four lemmas verified in all three theories.

5.2.5 Token Binding and Proof-of-Possession (T)

`T1_stolen_token_unusable.`

A leaked access token cannot be used for resource access without a client-created and AS-verified DPoP proof, unless the client key was revealed.

`T2_token_access_requires_pop.`

Every successful token-based access is preceded by a verified DPoP under the AS-accepted key.

`T3_no_token_identity_crossing.`

A token cannot be used under a different (ClientID, key) pair than it was issued for.

`T4_stolen_refresh_token_unusable.`

A leaked refresh token cannot be used without a prior client-created DPoP, or client-key revelation.

`T5_stolen_refresh_needs_dpop.`

Strengthens `T4`: the leaked refresh token cannot be exercised without a DPoP that is created *and* verified at the moment of use.

Status.

All five lemmas verified in all three theories.

5.2.6 Replay Resistance (R)

`R1_dpop_requires_creation.`

Every verified DPoP proof was created by a client beforehand, unless a client key was revealed.

Status.

Verified in all three theories.

5.2.7 Attestation Binding (B)

`B1_attestation_unforgeable.`

Every AA-signed attestation corresponds to a real client attestation request, unless the AA key was revealed.

`B2_as_requires_attestation.`

The AS only accepts a client public key after an AA-signed attestation for it has been observed, unless the AA key was revealed.

`B3_token_implies_attestation.`

Every issued token is preceded by a signed attestation followed by a client-key acceptance.

`B4_attestation_key_binding.`

If two AS-accepted client keys are attested under the same nonce, they are identical.

`B5_attestation_nonce_single_use.`

The AA never reuses a nonce.

`B6_cat_unforgeable.`

A CAT verified by the AS implies a prior CAT issuance by the backend, unless the backend key was revealed.

`B7_dpop_jkt_matches_instance_key.`

The `dpop_jkt` verified by the AS matches the key the client used in its attestation request, unless the client key was revealed.

Status.

All seven lemmas verified in all three theories.

5.2.8 Key Continuity (K)

`K1_key_continuity_attestation_to_token.`

The AS-accepted key matches the key in any later token under the same ClientID, unless that key was revealed.

`K2_key_continuity_token_to_access.`

A token issued under one key cannot be used under any other key.

K3_dpop_key_matches_accepted.

A DPoP-protected token-use against the AS uses the AS-accepted key.

Status.

All three lemmas verified in all three theories.

5.2.9 Client/Device Identity Binding (I)

I1_token_requires_hardware_key.

Every issued token traces back to a hardware-bound key generation followed by an AS acceptance of that key, unless the AA key or the client key was revealed.

I2_access_requires_hardware_key.

Every token-based resource access traces back to the same hardware-bound chain.

Status.

Both lemmas verified in all three theories.

5.2.10 End-to-End Properties (E)

E1_token_requires_authentication.

Every issued token is preceded by an authentication-completion event for the same ClientID.

E2_resource_matches_issued_token.

Every resource received by a client traces back to a server-side delivery under a previously issued token.

E3_resource_requires_fresh_dpop.

Every received resource is preceded by a fresh client-created DPoP, a verification of that DPoP, and an AS acceptance of the underlying client key, unless that key was revealed.

E4_refresh_requires_issuance.

Every use of a refresh token is preceded by a corresponding issuance event for the same refresh token, client key, and issuer.

Status.

All four lemmas verified in all three theories.

5.2.11 Issuer Mix-Up Prevention (ISS)

`ISS1_consistency`.

A client's verified issuer matches an issuer it had previously learned.

`ISS2_no_mix_up`.

If a code is issued under one issuer and later redeemed yielding a token under another issuer for the same ClientID/key, the issuers coincide.

`ISS3_token_origin`.

A token tagged as issued by some issuer is preceded by an authorization-code issuance for that ClientID under the same issuer.

`ISS4_client_rejects_wrong_iss`.

The issuer the client verifies in the token-response phase equals the issuer it learned at challenge time.

`ISS5_refresh_preserves_iss`.

Refresh-token use preserves the original issuer of the token.

Status.

All five lemmas verified in all three theories.

5.2.12 Flow Ordering (F)

`F1_resource_access_requires_token_exchange`,
`F2_token_exchange_requires_authorization`,
`F3_authorization_requires_challenge`.

Together these establish the AS temporal order: *challenge* → *authorization* → *token exchange* → *resource access*.

Status.

All three lemmas verified in all three theories.

5.2.13 CIBA: Variant-Specific Results

`CIBA1_authcode_after_user_auth`.

Every authorization code is preceded by a CIBA `UserAuth_Initiated` event whose `auth_req_id` is bound to that code via `CIBA_AuthEvent_Bound` at the same time point.

CIBA2_complete_after_request.

Every authentication-completion event is preceded by a `UserAuth_Initiated` event whose `auth_req_id` was already bound to the ClientID earlier.

CIBA3_auth_req_id_unique.

The AS accepts a user-authentication completion at most once per `auth_req_id`.

CIBA4_auth_event_binding.

Each `auth_req_id` is bound to at most one authorization code.

CIBA5_code_single_issuance.

Each `auth_req_id` produces at most one `CIBA_Code_Issued` event.

CIBA6_no_unsolicited_auth.

The AS only accepts a user-auth completion for an `auth_req_id` previously bound to a ClientID.

CIBA7_poll_authorized_implies_user_auth.

A `poll_authorized` response implies a prior user-auth completion.

CIBA8_code_requires_poll.

Every CIBA-issued authorization code is preceded by a poll request.

CIBA9_auth_req_id_bound_to_client.

An `auth_req_id` is bound to exactly one ClientID at the AS.

The CIBA `SANITY_full_chain` witnesses:
`AS_Issued_Nonce` →
`Client_Sent_Attestation_Request` →
`AA_Signed_Attestation` →
`Backend_Issued_CAT` →
`AS_Accepted_Client_Key` →
`UserAuth_Initiated` →
`AS_UserAuthAccepted` / `AuthzCode_Issued` →
`CIBA_Code_Issued` →
`Token_Issued` →
`Token_Used_For_Access` →
`Client_Resourse_Received`.

Status.

All CIBA-specific lemmas verified.

5.2.14 FIDO: Variant-Specific Results

The FIDO theory differs from CIBA and ROPC in three modelling deltas documented in the file header: the AA enclave is global rather than per-ClientID. ClientIDs are minted together with their FIDO credentials (so each has exactly one credential and one `rpId`); the `[sources]` typing lemma is strengthened with explicit `F`-cases for `AS_Issued_Nonce` and `AS_Issued_FIDONonce`.

`FIDO1_authcode_after_init.`

Every authorization code is preceded by a `FIDOAuth_Initiated` event for the same ClientID.

`FIDO2_complete_after_init.`

Every authentication-completion event is preceded by a `FIDOAuth_Initiated`.

`FIDO3_fido_key_secrecy.`

The FIDO authenticators private key remains unknown to the attacker unless `Reveal_FIDO_Key` is fired.

`FIDO4_credential_required_for_auth.`

Every successful FIDO authentication implies a registered FIDO key for that ClientID, unless the FIDO key was revealed.

`FIDO5_origin_binding.`

Every successful FIDO authentication is preceded by FIDO key registration.

`FIDO6_auth_binds_to_attested_client.`

Every successful FIDO authentication is preceded by an AS acceptance of some client key for the same ClientID.

`FIDO7_credential_rpId_scoping.`

The `rpId` used at signature time equals the `rpId` the credential was registered under, unless the FIDO key was revealed.

`FIDO8_as_enforces_rpId_binding.`

Every successful FIDO authentication for a given `rpId` traces back to a credential registration with the same `rpId`.

FIDO9_fido_nonce_single_use.

A FIDO challenge nonce is issued at most once and is bound to a unique (ClientID, rpId) pair.

The FIDO SANITY_full_chain witnesses:
AS_Issued_Nonce → ··· →
AS_Accepted_Client_Key →
FIDOAuth_Initiated →
FIDO_Auth_Success →
AuthzCode_Issued →
Token_Issued →
Token_Used_For_Access →
RS_Access_Granted.

Status.

All FIDO-specific lemmas verified.

5.2.15 ROPC: Variant-Specific Results

ROPC1_authcode_after_init.

Every authorization code is preceded by a ROPCAuth_Initiated event for the same ClientID.

ROPC2_complete_after_init.

Every authentication-completion event is preceded by a ROPCAuth_Initiated for the same ClientID.

ROPC3_password_secrecy.

The user's password remains unknown to the attacker unless Reveal_User_Password is fired.

ROPC4_auth_requires_credentials.

Every successful ROPC authentication is preceded by a credential-use event with a real password, unless the password was revealed.

ROPC5_ropc_requires_dpop.

Every successful ROPC authentication coincides with a verified DPoP under an AS-accepted key.

ROPC6_auth_binds_to_attested_client.

Every successful ROPC authentication is preceded by a client-key acceptance for the same ClientID.

`ROPC7_password_not_leaked_by_protocol.`

Use of a password in the protocol does not leak it: it stays unknown unless either the password or the client key is revealed.

`ROPC8_dpop_params_bind_password_use.`

Every successful ROPC authentication is preceded by a credential-use and binds a verified DPoP at the moment of success.

Status.

All ROPC-specific lemmas verified.

5.3 Summary

Table 5.1 covers the amount of all categorized lemmas, the last row being the amount that have been verified (all).

Category	CIBA	ROPC	FIDO
Sanity	7	7	7
Helpers (sources, reuse, unique)	9	9	9
Secrecy (S)	6	6	6
Authorization (Z)	4	4	4
Token binding (T)	5	5	5
Replay (R)	1	1	1
Attestation (B)	7	7	7
Key continuity (K)	3	3	3
Identity (I)	2	2	2
End-to-end (E)	4	4	4
Mix-up (ISS)	5	5	5
Flow (F)	3	3	3
Variant-specific	9	8	9
Total verified	**65**	**64**	**65**

Table 5.1: Lemma counts per theory.

Verification verdict.

All lemmas in each theory were discharged automatically by Tamarin under the heuristic `S`. The seven sanity lemmas (including `SANITY_full_chain`) establish that the `all-traces` guarantees apply to non-empty trace sets.

6

Alternative mAuth Variants

The baseline mAuth protocol relies on DPoP for sender-constrained tokens and device attestation for client authentication. These choices reflect a balance between deployability and security suited to the common case of consumer facing mobile applications. However, the protocol architecture could be altered for specific goals, the proof-of-possession and client authentication mechanisms can be substituted without altering the overall protocol flow. This chapter examines three alternative mechanisms, discusses the trade offs involved, and identifies deployment contexts where each alternative may be preferable.

6.1 mTLS

Mutual Transport Layer Security (mTLS), specified in RFC 8705 [9], extends the standard TLS handshake by requiring the client to present an X.509 certificate during connection establishment. The server verifies the client certificate against a trusted certificate authority or a pre-registered certificate, and the resulting TLS session is cryptographically bound to the client identity.

In an mTLS-based mAuth variant, the following substitutions would apply. DPoP proof-of-possession would be replaced by TLS certificate binding, instead of generating a fresh DPoP proof per request, the client establishes an mTLS connection and the server binds tokens to the client certificate thumbprint. The per-request freshness guarantee provided by unique DPoP parameters would be replaced by the session-level binding of the TLS connection. Device attestation could optionally be replaced or supplemented by certificate provisioning through a managed PKI, where the certificate issuance process itself serves as the attestation mechanism.

The primary advantage of mTLS is maturity. Certificate-based client authentication is well understood, widely supported in enterprise infrastructure, and has undergone extensive formal analysis. Financial-grade deployments frequently mandate mTLS because it integrates naturally with existing PKI infrastructure and hardware security modules. The FAPI 2.0 [23] Security Profile lists mTLS as one of two approved mechanisms for sender-constrained tokens. For environments where certificate lifecycle management is already established, mTLS avoids the need to implement DPoP proof generation and verification logic in both the client and server.

The disadvantages are significant for the mobile context that mAuth targets. Certificate provisioning on consumer device's is operationally complex, it requires either pre-installed certificates. Unlike DPoP, where the client generates a key pair per session, mTLS requires a persistent certificate that must be renewed, revoked, and distributed through a trusted channel.

An mTLS variant of mAuth would be most appropriate in enterprise and financial deployments where a managed device fleet allows centralized certificate provisioning, where existing PKI infrastructure is available, and where the operational cost of certificate lifecycle management is justified by regulatory requirements. Open banking ecosystems, which already mandate mTLS through FAPI compliance, are a natural fit. Consumer-facing applications on unmanaged device's, where certificate provisioning infrastructure is typically absent, would not benefit from this substitution.

6.2 HTTP Message Signatures

HTTP Message Signatures, specified in RFC 9421 [24], provide a mechanism for signing individual HTTP messages using asymmetric or symmetric cryptography. The sender selects a set of HTTP message components and produces a signature over these components. The signature, along with metadata identifying the covered components and the signing key, is transmitted in the `Signature` and `Signature-Input` HTTP headers.

In an mAuth variant based on HTTP Message Signatures, DPoP proof-of-possession would be replaced by per-request message signatures. Rather than signing a DPoP proof containing only the JTI, HTTP method, target URI, and timestamp, the client would sign the full set of relevant HTTP message components. This provides strictly stronger binding than DPoP, while DPoP signs only selected request metadata, HTTP Message Signatures can cover the request body, authorization headers, and any other components, preventing an attacker from modifying unsigned parts of the request. Device attestation would remain unchanged, as HTTP Message Signatures address only the per-request binding and do not provide client identity assurance.

The advantage of HTTP Message Signatures is the granularity of protection. DPoP does not sign the request body or most headers, which means that an attacker who obtains a valid DPoP proof could potentially modify the unsigned portions of the request. HTTP Message Signatures close this gap by allowing the client to cover all security-relevant components. The FAPI 2.0 Security Profile identifies this limitation of DPoP in its discussion of DPoP proof replay (Section 6.2 of [23]) and suggests message signing as a possible mitigation.

The disadvantages relate to complexity and ecosystem maturity. HTTP Message Signatures require both client and server to agree on which components are covered, to implement the canonicalization algorithm correctly, and to handle the interaction between signature verification and any intermediaries that may modify headers. The

specification is relatively recent (published in 2024), and library support is less mature than for JWTs and DPoP. Additionally, the canonicalization of HTTP message components introduces a new class of implementation errors.

A HTTP Message Signatures variant would be appropriate in high-security deployments where the integrity of the full HTTP request must be guaranteed, particularly where the threat model includes an attacker capable of reading and modifying requests at the resource server. Payment initiation APIs, where the request body contains transaction details such as amounts and destination accounts, are a use case where body signing provides meaningful additional protection beyond DPoP. For deployments where DPoP coverage is sufficient and implementation simplicity is preferred, the additional complexity of HTTP Message Signatures may not be justified.

6.3 Verifiable Credentials

Verifiable Credentials (VCs), specified by the W3C [44], are a data model for cryptographically signed claims about a subject. A credential is issued by an *issuer*, held by a *holder*, and presented to a *verifier*, who validates the credential using the issuers verification material — typically discovered through a Decentralized Identifier document or a trust registry rather than provided as a bare public key. In a VC-based mAuth variant, the platform attestation step would be replaced by a verifiable credential asserting properties of the client instance (binding of a public key to a device identity, application integrity, policy compliance). The authorization server would verify the credential against the resolved verification material. DPoP would remain unchanged, since it operates at a different layer.

The advantage is platform independence: mAuths current attestation step is tied to Googles Play Integrity and Apples App Attest, while a VC can be issued by any trusted party, allowing client's on other platforms to participate and enabling richer claims than the fixed verdicts those APIs return.

The main drawback is the lack of an initial trust foundation. The current design inherits issuer-key distribution and revocation from the mobile platforms; a VC-based design must replicate this with Decentralized Identifiers and a status mechanisms. The VC ecosystem is also still evolving and interoperability between implementations is uneven.

A VC-based variant would be appropriate in contexts where platform independence is a requirement, where the set of trusted client properties extends beyond what platform attestation APIs provide, or where decentralized identity infrastructure is already deployed. For consumer-facing deployments on standard Android and iOS device's, the maturity of platform attestation APIs make them a more practical choice than verifiable credentials at present.

7

Related Work

While the dominant approach to native-app authentication still routes the user through a system browser, a number of efforts share mAuths approach of keeping the entire flow inside the application.

7.1 HAAPI

Curitys Hypermedia Authentication API (HAAPI) [45] reframes authentication as a hypermedia state machine: instead of serving HTML and relying on browser redirects, the authorization server returns JSON representations of forms, links, and follow-up actions that the native client renders directly, enabling a fully native UI without leaving the application.

HAAPI and mAuth share the browserless objective and converge on the same building blocks: both bind tokens via DPoP, and both use established Android and iOS tools for hardware-backed client attestation. They differ in two respects. HAAPI is a Curity-specific protocol with a custom media type and a vendor-defined data model, whereas mAuth is a composition of standards-track specifications (OAuth 2.1, DPoP, CIBA, WebAuthn, attestation-based client authentication) targeting interoperable implementations. HAAPIs hypermedia model is also generic over the user-authentication method, with the authorization server dynamically driving the client through whichever method is configured, while mAuth fixes three concrete variants (CIBA, FIDO2, ROPC) and admits each to symbolic verification.

7.2 Microsoft Entra

Microsoft Entra Native Authentication [46] is a commercial implementation of a browserless authentication approach. The Microsoft Authentication Library (MSAL) SDK communicates directly with dedicated Entra endpoints using credential challenge types such as passwords, email OTPs, SMS OTPs, and passkeys. When Conditional Access requirements cannot be satisfied through native flows, the MSAL SDK may fall back to a system browser.

Compared with mAuth, the design emphasises native UX and policy integration rather than cryptographic client-instance continuity. Although Microsoft supports

proof-of-possession mechanisms in some contexts, the native authentication flow itself does not fundamentally bind issued tokens to a hardware-attested application instance. Consequently, the authorization server does not obtain protocol-level evidence that requests originate from a genuine, unmodified application running on a non-compromised device.

7.3 OAuth 2.0 App2App Browserless Flow

The IETF draft *OAuth 2.0 App2App Browserless Flow* [47] proposes a browserless OAuth interaction for native applications communicating across trust domains. Rather than relying on browser redirects, the draft introduces mechanisms for routing authorization requests between native applications using links and dedicated native authorization endpoints.

Like mAuth, the proposal aims to preserve a fully native user experience. However, its focus is on cross-domain request routing and native navigation rather than on cryptographic client-instance binding. The draft does not mandate hardware-backed attestation or proof-of-possession token semantics

7.4 OAuth 2.0 for First-Party Applications

The IETF working-group draft *OAuth 2.0 for First-Party Applications* (FiPA) [48] introduces an Authorization Challenge Endpoint that allows first-party native applications to conduct OAuth interactions without browser redirects, except in high-risk or exceptional situations. The draft defines the structure of a browserless OAuth flow, but leaves user authentication and client-attestation mechanisms to individual deployments. mAuth can be viewed as a concrete security profile layered on top of this model.

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

In [49], Pedram Hosseyni, Ralf Küsters, and Tim Würtele present an in-depth formal security analysis of the OpenID FAPI 2.0 protocol [23], conducted to accompany its official standardization process. Utilizing the Web Infrastructure Model (WIM) [50], the researchers developed a comprehensive formal model to evaluate FAPI 2.0 against core security goals such as authorization, authentication, and session integrity under a strong attacker model. The analysis successfully uncovered several new vulnerabilities, including attacker token injection and client impersonation attacks.

The security analysis in their work covered the parts of secure login relating to FAPI and not other parts which are present in mAuth such as ROPC or hardware attestation. Another deviation is that WIM models browser behavior, mAuths native app

approach does not utilize browser redirects as part of its login flow.

8

Discussion

8.1 Adopted Standards Sufficiency

A profile of OAuth is only as well-formed as the union of specifications it leans on. mAuth is not a single self-contained document, it is a composition of an authorization framework, a proof-of-possession mechanism, an attestation framework, two platform-specific attestation providers, several lifecycle endpoints, and normative security baselines. The question this section addresses is whether that composition is *sufficient*, in a precise sense: every wire-level message, endpoint, JWT, and cryptographic choice in the flow must trace to a normative anchor, so that nowhere is the implementer left to choose between security-relevant alternatives on their own authority. A profile that left such a choice open would not be a specification of one protocol but a family of protocols, only some of which are secure.

Pre-flow registration and discovery. The App vendor registers the composite client through Dynamic Client Registration [25] and the App Backend retrieves AS configuration through Authorization Server Metadata [26]. Together these specs fully constrain the static parameters that subsequent steps depend on: `client_id`, `grant_types`, `jwks`, `token_endpoint_auth_method`, `dpop_bound_access_tokens`, the registered `authorization_details_types`, and the AS endpoint URLs (8.4).

Backend client authentication. Every backend-initiated request to the AS authenticates with `private_key_jwt`, an authentication method named by OpenID Connect Core 1.0 [18] and constructed per the JWT bearer assertion profile [17]. The JWT BCP [40] additionally fixes the algorithm policy and explicit typing (8.1).

Attestation challenge and platform attestation. The attestation challenge endpoint comes from the OAuth attestation-based client authentication draft [32], and the platform attestation itself is produced by Google Play Integrity [19] on Android and Apple App Attest [20] on iOS. The instance key pair is generated and held in hardware (Android Keystore [7] / StrongBox [41], iOS Secure Enclave); these are not profile-level requirements that any RFC could specify in the abstract, but platform-vendor APIs that mAuth must invoke directly (8.3).

Authorization request and code binding. The authorization request itself is OAuth 2.1 authorization-code grant [2] (8.1), with the code bound to the instance

public key through the `dpop_jkt` parameter of RFC 9449 §10 [14]. The authorization response carries the issuer identifier of RFC 9207 [29], the two anchors that close the code-injection and mix-up gaps respectively (8.2).

User authentication. The three variants pull from three different specs. CIBA [22] provides the backchannel authentication endpoint, the `auth_req_id` machinery, and the poll-mode delivery rules. WebAuthn Level 3 [33] provides the challenge–response ceremony, the `rpId` scoping rules, and the verification algorithm at the AS. ROPC inherits its grant type from RFC 6749 §4.3 [1], hardened in this profile by hashing the password client-side with Argon2id [4] and embedding the result in a DPoP proof (8.1).

Token issuance and structure. The token endpoint follows OAuth 2.1 [2], the access token is a JWT per the JWT format [10], with the `cnf.jkt` confirmation claim of RFC 9449 §6.1 [14] binding it to the instance key. The JWK thumbprint construction itself is RFC 7638 [16] (8.1,8.2).

Audience and consent. Resource Indicators [30] give every issued token an `aud` value derived from the validated `resource` parameter, and Rich Authorization Requests [31] provide a structured `authorization_details` in place of opaque scope strings (8.4).

Resource access. The Resource Server validates the access token (locally or via introspection [27]) and the DPoP proof per RFC 9449 §4.3 [14], the point at which the token-to-key binding established at issuance is finally enforced (8.4,8.2).

Refresh and revocation. Refresh-token rotation comes from OAuth 2.1 §4.3.3 [2], the revocation endpoint comes from RFC 7009 [28], and the grant-chain flow on rotation reuse is mandated by both OAuth 2.1 and the OAuth 2.0 Security BCP [42] (8.4,8.5).

Transport. Every wire-level message uses TLS 1.3 [8], with the server certificate matched against the issuer host of RFC 9207 (8.5).

The table below (8.1) lists the main aspects covering the core protocol:

Mechanism	Specification	Role in mAuth
Core protocol		
Authorization framework	OAuth 2.1 [2]	Provides roles, endpoints, code-grant semantics, token-endpoint behavior
Backwards-compatible vocabulary	RFC 6749 [1]	Terminology, error codes, ROPC grant type
JWT format and claims	RFC 7519 [10]	Access-token structure
JWT best practices	RFC 8725 [40]	Algorithm policy, explicit typing, claim validation
JWK thumbprint computation	RFC 7638 [16]	JWK Hash Values
JWT bearer assertions	RFC 7523 [17]	JWT-based client authentication construction
`private_key_jwt` method, `id_token_hint`	OpenID Connect Core 1.0 [18]	Names the backend authentication method, defines hint parameters used by CIBA
Backchannel authentication	CIBA Core 1.0 [22]	Out-of-band auth flow, poll-mode delivery, binding messages
WebAuthn / FIDO2	WebAuthn Level 3 [33]	On-device passkey ceremony, `rpId` scoping, signCount-based clone detection

Table 8.1: Adopted Standards for core mAuth

The table below (8.2) lists protocol aspects relating to sender-constraint are listed along with the normative specifications connecting them:

Mechanism	Specification	Role in mAuth
Sender-constraint and proof-of-possession		
DPoP proof construction & verification	RFC 9449 [14]	Sender-constraint mechanism, full §4.3 verification checks
`dpop_jkt` authorization-request binding	RFC 9449 §10	Code-to-key binding at authorization endpoint
`cnf.jkt` access-token binding	RFC 9449 §6.1	Token-to-key binding at token endpoint
DPoP nonce & replay defense	RFC 9449 §8–9, §11.1	Server-issued nonces, `jti` freshness
Issuer identification	RFC 9207 [29]	Mix-up defense via `iss` threading

Table 8.2: Adopted Sender-constraint and proof-of-possession Standards

The table below (8.3) lists attestation mechanisms and their connecting specifications:

Mechanism *Attestation*	**Specification**	**Role in mAuth**
Client attestation framework	`draft-ietf-oauth-attestation-based-client-auth` [32]	Client Attestation JWT (`typ=client-attestation+jwt`), Client Attestation PoP, key continuity invariant
Hardware-backed key generation & storage	Android Keystore / StrongBox [7], iOS Secure Enclave [6]	Non-extractable instance private key
Android platform attestation	Google Play Integrity API [19]	Platform attestation provider (signing key, integrity verdict)
iOS platform attestation	Apple App Attest / DeviceCheck [20]	Platform attestation provider (signing key, attestation object)

Table 8.3: Adopted Attestation Standards

The table below (8.4) covers resource access and lifecycle operations and their connected normative sources:

Mechanism *Resource access & lifecycle*	**Specification**	**Role in mAuth**
Token introspection	RFC 7662 [27]	RS→AS validation for opaque tokens, DPoP metadata exposure
Token revocation	RFC 7009 [28]	Refresh-token revocation, grant-chain cascade
Resource indicators	RFC 8707 [30]	`resource` parameter, audience binding (against cross-RS replay)
Rich authorization requests	RFC 9396 [31]	`authorization_details` for fine-grained intent
Dynamic client registration	RFC 7591 [25]	Pre-flow client metadata establishment
AS metadata discovery	RFC 8414 [26]	Pre-flow endpoint and capability discovery

Table 8.4: Adopted Resource Access & Lifecycle Standards

The table below (8.5) lists the followed security baselines and their connecting normative sources:

Mechanism *Security baselines*	**Specification**	**Role in mAuth**
OAuth 2.0 Security BCP	RFC 9700 [42]	Threat enumeration and mitigation requirements
Financial-grade API profile	FAPI 2.0 Security Profile [23]	High-security profile: sender-constraint mandate, key-compromise granularity, ≥128-bit entropy
Transport security	TLS 1.3 (RFC 8446) [8]	Confidentiality, integrity, and authenticity for all hops

Table 8.5: Adopted Security Baseline Standards

Sufficiency Assessment

The set of specifications collected in the tables is sufficient to define mAuth. Every wire-level message, every endpoint, every JWT, and every cryptographic decision in the flow traces to a normative anchor in one of those tables, or to one of the profile-level refinements layered on top of them. There is no point in the protocol where the implementer is left to choose between security-relevant alternatives without guidance, which is precisely the property the specification-sufficiency argument set out to establish.

8.2 Tamarin Model Symbolic Representation of mAuth

The three Tamarin theories `mAuth_CIBA`, `mAuth_ROPC`, and `mAuth_FIDO` encode the formalized protocol. Every actor has a corresponding role, every protocol step has at least one rule producing the same outputs from the same inputs.

Structural correspondence

Each step of the formalized mAuth maps to a rule block whose section header mirrors the original numbering. The actors of the formalized design appear as role labels: `Client`, `Backend`, `AS`, `AA`, `RS API`, `UserAuthMethod`, and `FIDOAuthenticator`. Inter-actor messages use the `Out_S` / `In_S` / `Sec` pattern, abstracting the TLS hops into a single linear primitive whose properties (confidentiality, authenticity, no replay) are exactly those the formalized flow assumes. The `Out`/`In` channel is reserved for the leak and reveal rules.

The four AS-side bindings appear as explicit facts: the nonce–ClientID binding as `AS_Pending_Challenge_Nonce_Useable(ClientID, ~nonce, iss)`, produced at Step 3 and consumed at Step 10; the client-key registration as `!AS_Client_Tied_PublicKey(ClientID, pkC, iss)`, written by Step 10 and read by every later rule; the authorization-code binding as `AuthzCode(code, ClientID, pkC, iss)`, forcing every redemption to present the same triple, and the CIBA state-

management bindings as !ASOnly_AuthReq_ClientID, !ASOnly_AuthReq_ClientKey, and !ASOnly_AuthReq_Iss.

Phase-by-Phase Parallel

Pre-flow setup (M0, M0a, M1).

The AS metadata discovery, dynamic registration, and JWKs fetch operations are abstracted into trusted setup rules (Setup_AS_Identity, Setup_Backend_Identity, Setup_Resource_Server, Setup_AA_Enclave).

Attestation challenge (Steps 2–3).

The challenge request is realised by Client_AttestChallReq_ToBackend_Step2 and Backend_AttestChallReq_ToAS_Step2, the response by AS_IssueChallenge_ToBackend_Step3, which generates the nonce via Fr and produces the linear gating fact AS_Pending_Challenge_Nonce_Useable. The DPoP-Nonce is folded into the same freshness used at Step 8.

Hardware key generation (Step 4).

Client_GenerateKey_ToClient_Step4 draws a fresh ~skC, emits Client_Key_In_Hardware, and stores the key in the persistent fact !Client_Key.

Platform attestation (Steps 5–7).

The binding value SHA-256(nonce ‖ SHA-256(pk)) of the formalized flow is collapsed into the symbolic tuple <Device_Info, nonce, pkC>, signed in AA_SignAttestation_ToAA_Step6 as sign(<Device_Info, nonce, pkC>, aa_sk). Client_ForwardPlatformAttest_ToBackend_Step7b and Backend_WrapCAT_ToClient_Step7c produce the second of the two-signature artefact.

Authorization request and AS verification (Steps 8–10).

Client_CreateDPoPP_ToClient_Step8 produces the first DPoP proof, the context tag in the second argument plays the role of the htm/htu/nonce claims. The bundle reaches the AS through Client_SendAuthzReq_ToBackend_Step9 and Backend_SendAuthzReq_ToAS_Step9. AS_VerifyJWTAndDPoPP_ToBackend_Step10 performs four Eq(verify(...), true) checks corresponding exactly to the formalized verification: platform attestation against pkAA, DPoP against pkC, CAT against pkBackend, and dpop_jkt equality against pkC. On success it commits the trust decision via !AS_Client_Tied_PublicKey.

User authentication, CIBA variant (Step 11).

`AS_ACKSend_ToBackend_CIBAStep3` mints `~auth_req_id` and binds it to the client and key via the three persistent `!ASOnly_AuthReq_*` facts. A separate `UserAuthMethod` role carries the user's decision via `AuthMethod_UserAuth_ToAS_CIBAStep4`, after which `AS_UserAuthAccepted_ToAS_CIBAStep4` mints the authorization code and binds it to the same `auth_req_id`. The polling loop is reduced to two rules (pending and authorized); only the authorized rule emits `CIBA_Code_Issued`.

User authentication, FIDO2 variant (Step 11).

The challenge travels via `Backend_SendChallenge_ToClient_FIDOStep1` and `Client_StartAuth_ToFIDOAuthenticator_FIDOStep2` to the `FIDOAuthenticator` role, which performs `Eq(rpId, rpIdStored)` and returns `sign(<rpId, FIDONonce>, skFIDO)`. The `clientDataJSON`, `authenticatorData`, and `signCount` machinery of FIDO2 authentication are abstracted into this single signature.
At `AS_VerifyChallengeSendCode_ToBackend_FIDOStep5` the AS performs three equality checks — `rpId` matches both the stored and challenge values, and the signature verifies under `pkFIDO` — discharging the WebAuthn algorithm at the abstraction level the model targets.

User authentication, ROPC variant (Step 11).

`Client_ROPCAuth_ROPCStep2_3` produces `sign(<~DPoP_ROPC_Params, password>, skC)`, preserving the formalized hardening. The Argon2id layer is abstracted, secrecy of the password is then a consequence of the secure channel and the DPoP signature rather than of slow-hash properties.

Token exchange (Steps 12–13).

`Client_CreateDPoP_ForTokenExchange` produces a fresh proof `sign(<DPoP_Params_TE, 'token_endpoint'>, skC)`, and `AS_TokenAns_ToBackend_Step13` verifies it, marks the code redeemed, and mints the three tokens carried as persistent facts (`!ConstrainedAccess`, `!ConstrainedReference`, `!ConstrainedRefresh`) bound to `pkC`. The `cnf.jkt` confirmation claim is structural in this encoding, the key under which the token validates is part of the fact that names the token.

Reference-token delivery and resource access (Steps 14–19).

`Backend_SendReferenceToken_ToClient_Step14` stores the real tokens in `Backend_AccessTokenHeld` and `Backend_RefreshTokenHeld` and returns only the reference token, the direct analogue of the formalized reference-token construction. `Client_GenerateDPoP_Step15_SendDPoP_ToBackend_Step16` signs `<DPoP_Params2, reference_token>`, with the reference token playing the role of the `ath` claim. The two-stage `ath` computation is collapsed into this single sign-term because the model treats context-binding as primitive.

Refresh sub-flow (R).

`Client_RequestRefresh` produces
`sign(<DPoP_Params_Refresh, 'refresh'>, skC)`, and
`AS_VerifyRefresh_IssueToken` verifies the proof and emits `RefreshToken_Used`. The model intentionally does not produce a replacement access token, the rule sequence following refresh would be identical to the original token issuance and would double the proof effort without changing any conclusion.

Encoding of mAuth-Specific Hardenings

Two-signature attestation. The platform attestation `sign(<Device_Info, nonce, pkC>, aa_sk)` and the Client Attestation JWT `sign(<'client-attestation+jwt', ClientID, pkC, nonce, platformAttest>, backend_sk)` are verified independently at Step 10. This is what lets the model state separately that platform-level attestation is unforgeableand that the CAT is unforgeable.

Explicit `dpop_jkt`. The `dpop_jkt` parameter is encoded as a separate term in the authorization request and checked `Eq(dpop_jkt, pkC)` at the AS, exposing the dpop thumbprint check as an explicit equality.

Key continuity. The instance public key `pkC` appears identically in `cnf.jwk` (CAT), `dpop_jkt`, the DPoP proofs `jwk` header, and the access tokens `cnf.jkt`. Under perfect cryptography, reusing the same symbolic term across these positions is equivalent to the thumbprint equality checks.

Context-bound DPoP proofs. Proofs are signed over context-specific payloads: `<DPoP_Params, nonce>` for authorization, `<DPoP_Params, 'token_endpoint'>` for token exchange, `<DPoP_Params, 'refresh'>` for refresh, and `<DPoP_Params, reference_token>` for resource access. The context tags act as symbolic representations of `htm`/`htu`; proof uniqueness comes from the `unique_dpopp` restriction, the symbolic representation of `jti` freshness.

Coverage Matrix

The following tables cover all mAuth mechanisms with the Tamarin rule-names corresponding to them.

Below the setup rules are listed (8.6) with corresponding Tamarin rule names:

Step	**Formalized mechanisms**	**Tamarin representation**
Setup Rules		
M0, M0a, M1	Dynamic registration; AS metadata + JWKS discovery	`Setup_AS_Identity`, `Setup_Backend_Identity`, `Setup_Resource_Server`, `Setup_AA_Enclave`

Table 8.6: Formalized vs Tamarin Setup Rules

Below the attestation challenge steps are listed (8.7) with their corresponding tamarin rule-names:

Step	**Formalized mechanisms**	**Tamarin representation**
Attestation Challenge		
2	Attestation challenge request: Frontend → Backend → AS	`Client_AttestChallReq_ToBackend_Step2`, `Backend_AttestChallReq_ToAS_Step2`
3	Attestation challenge response with fresh nonce and `iss`	`AS_IssueChallenge_ToBackend_Step3` (`Fr(~nonce)`, `AS_Pending_Challenge_Nonce_Useable`), `Backend_IssueChallenge_ToClient_Step3`

Table 8.7: Formalized vs Tamarin Attestation Challenge

Below key generation steps along with attestation is listed (8.8) with their corresponding tamarin-rules:

Step	Formalized mechanisms	Tamarin representation
Key generation & Attestation		
4	Hardware-backed instance key generation in secure element	`Client_GenerateKey_ToClient_Step4` (`Fr(~skC)`, `Client_Key_In_Hardware`, `!Client_Key`)
5	Binding value passed to platform attestation API	`Client_RequestAttestation_ToAA_Step5` (tuple `<Device_Info, nonce, pkC>`)
6	Platform attestation produced by AA	`AA_SignAttestation_ToAA_Step6` (`sign(<Device_Info, nonce, pkC>, aa_sk)`)
7	Backend wraps platform attestation in CAT	`AA_SendAttestation_ToClient_Step7`, `Client_ForwardPlatformAttest_ToBackend_Step7b`, `Backend_WrapCAT_ToClient_Step7c`

Table 8.8: Formalized vs Tamarin Key Generation & Attestation

Below DPoP creation and authorization request steps are listed (8.9) with their corresponding tamarin-rules:

Step	Formalized mechanisms	Tamarin representation
DPoP Creation & Authorization Request		
8	DPoP proof creation, authorization context	`Client_CreateDPoPP_ToClient_Step8` (`sign(<DPoP_Params, nonce>, skC)`)
9	Authorization request: CAT + platform attestation + DPoP + `dpop_jkt`	`Client_SendAuthzReq_ToBackend_Step9`, `Backend_SendAuthzReq_ToAS_Step9`
10	AS verifies attestation chain, DPoP, and `dpop_jkt`; binds `pkC` to `ClientID`	`AS_VerifyJWTAndDPoPP_ToBackend_Step10` (four `Eq(verify(...), true)` actions; emits `!AS_Client_Tied_PublicKey`)

Table 8.9: Formalized vs Tamarin DPoP Creation & Authorization Request

Below the authentication steps are listed (8.10) with corresponding tamarin rule-names:

Step	**Formalized mechanisms**	**Tamarin representation**
Authentication		
11-CIBA	Backchannel ACK, out-of-band user auth, polling	`Backend_CIBAStart_ToClient _CIBAStep1`, `Client_CIBAStart_ToBackend _CIBAStep2`, `Backend_AuthReq_ToAS_CIBAStep2`, `AS_ACKSend_ToBackend_CIBAStep3`, `Backend_ACKSend_ToClient _CIBAStep3`, `Client_UserAuth_ToAuthMethod _PollStart_ToBackend_CIBAStep4`, `AuthMethod_UserAuth_ToAS _CIBAStep4`, `AS_UserAuthAccepted_ToAS _CIBAStep4`, `Backend_Poll_ToAS_Step5`, `AS_PollAnsPending/AnsAuthorized _ToBackend_CIBAStep5`
11-FIDO	WebAuthn challenge-response with `rpId` scoping	`Backend_SendChallenge_ToClient _FIDOStep1`, `Client_StartAuth _ToFIDOAuthenticator_FIDOStep2`, `FIDOAuthenticator_Authenticated (sign(<rpId, FIDONonce>, skFIDO))`, `Client_SendAuth_ToBackend _FIDOStep4`, `Backend_SendAuth_ToAS_FIDOStep4`, `AS_VerifyChallengeSendCode _ToBackend_FIDOStep5` (triple `Eq` on `rpId` and signature)
11-ROPC	DPoP-wrapped credentials	`Backend_ROPCStart_ToClient _ROPCStep1`, `Client_ROPCAuth_ROPCStep2_3 (sign(<DPoP_ROPC_Params, password>, skC))`, `Backend_ROPCAuth_ToAS_ROPCStep4`, `AS_VerifyROPC_SendCode _ROPCStep5_6`

Table 8.10: Formalized vs Tamarin Authentication

Below token exchange and resource access steps are listed (8.11) with corresponding tamarin rule-names:

Step	Formalized mechanisms	Tamarin representation
Token Exchange & Resource Access		
12	Token-endpoint DPoP creation and request	`Backend_RequestDPoP_ForTokenExchange`, `Client_CreateDPoP_ForTokenExchange (sign(<DPoP_Params_TE, 'token_endpoint'>, skC))`, `Backend_TokenRequest_ToAS_Step12`
13	AS issues access, reference, and refresh tokens, all bound to `pkC`	`AS_TokenAns_ToBackend_Step13` (`Fr` for each token, persistent `!ConstrainedAccess`, `!ConstrainedReference`, `!ConstrainedRefresh`)
14	Reference-token delivery to frontend; backend retains real tokens	`Backend_SendReferenceToken_ToClient_Step14` (`Backend_AccessTokenHeld`, `Backend_RefreshTokenHeld`)
15–16	Frontend creates resource-request DPoP and sends to backend	`Client_GenerateDPoP_Step15_SendDPoP_ToBackend_Step16 (sign(<DPoP_Params2, reference_token>, skC))`
17	Backend forwards request to RS	`Backend_ResourceRequest_ToRS_Step17`
18	RS introspects token at AS; AS verifies DPoP and token binding	`RS_ForwardIntrospection_ToAS`, `AS_VerifyDPoPPAndTokens_Step18`
19	RS delivers protected resource	`RS_SendResource_Step19`, `Client_Received_Data`

Table 8.11: Formalized vs Tamarin Token Exchange & Resource Access

Below the step for refresh token is listed (8.12) with its tamarin rule-names:

Step	Formalized mechanisms	Tamarin representation
Refresh Token		
R	Refresh-token grant with refresh-context DPoP	`Client_RequestRefresh (sign(<DPoP_Params_Refresh, 'refresh'>, skC))`, `Backend_RefreshRequest_ToAS`, `AS_VerifyRefresh_IssueToken`

Table 8.12: Formalized vs Tamarin Refresh Token

Conclusion

The Tamarin theories are an encoding of the formalized mAuth, not a separately constructed protocol that merely resemble it. Every step from the formalized mAuth traces to one or more rules, every actor of the formalized design has a role, every AS-side state binding is realised as a named fact, every mAuth-specific hardening the model undertakes to verify is encoded.

This correspondence is what makes the verification meaningful. The lemmas and the results are not statements about a parallel artefact whose relationship to the formalized mAuth would have to be argued separately, they are statements about the symbolic image of the formalized mAuth itself.

8.3 Lemma sufficiency

The lemmas are claimed to be sufficient in the sense that they collectively capture every security modeled property mAuth claims, against every attack class, under the stated abstractions and scope limitations. This section substantiates that claim by mapping each normative requirement of the relevant RFCs and profiles to the lemma or mechanism that establishes it.

Foundational Lemmas

Helper lemmas

The helper lemmas (`typing`, `token_implies_code`, `code_implies_accept`, `access_implies_issued`, `refresh_used_implies_issued`, `accept_unique_pkC`, `client_iss_unique`, `L_unique_dpopp_proof`, `L_unique_attestation_proof`) are not security goals. They close source resolution and expose structural invariants of the protocol that the main proofs cite as free assumptions. The two `L_*` lemmas are tautological in form — each restates a restriction belonging to the trusted base.

Sanity lemmas

The seven correctness checks (`SANITY_keygen`, `SANITY_token_issued`, `SANITY_dpop_verified`, `SANITY_token_used`, `SANITY_refresh_used`, `SANITY_resource_received`, `SANITY_full_chain`) demonstrate that the model is not vacuously satisfiable: no restriction silently rules out a legitimate trace. `SANITY_full_chain` is the strongest, exhibiting a single trace in which the entire ordered sequence of protocol events occurs, and is therefore by itself a witness that every other sanity property is reachable in the same model.

Coverage of Standardised Threat Models

RFC 9700 [42] and FAPI 2.0 are the two normative security baselines against which any OAuth profile is judged.

Below RFC 9700 adoptations are listed (8.13) with corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
RFC 9700 — OAuth 2.0 Security BCP	
§2.2.1 Sender-constrained access tokens	T1, T2, T3, K2, I1
§2.2.2 Sender-constrained refresh tokens	T4, T5, K1
§2.5 Client authentication (asymmetric)	B6, S6
§4.2.1 Leakage from OAuth client	T1, T3, T4, S2, S3, S5
§4.4 Mix-up attacks	ISS1, ISS2, ISS3, ISS4
§4.5 Authorization code injection	Z2, Z3, Z4, B7
§4.6 Access-token injection	T1, T3, K2
§4.9.2 Counterfeit / compromised RS	T1, T3
§4.10.1 Sender-constrained access tokens	T1, T2, T3, I1, I2
§4.14 Refresh-token protection	T4, T5, S5, E4
§4.15 Client impersonating RO	B1, B2, B3, B6, B7, I1

Table 8.13: RFC 9700 Security Properties and Connected Lemmas

Below FAPI 2.0 security standard adoptations are listed (8.14) with corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
FAPI 2.0 Security Profile	
§5.3.2 AS issues sender-constrained tokens only	T1, T2, T4
§5.3.3 Client's use sender-constrained access	T1, T2, I1, I2
§6.2 DPoP proof replay	R1
§6.3 Stolen access-token injection	T1, T3, K2
§6.7 Client impersonating Resource Owner	B1, B2, B3, B6, B7, I1

Table 8.14: FAPI 2.0 Security Properties and Connected Lemmas

Coverage of Mechanism-Specific Specifications

The remaining specifications adopted contribute individual mechanisms. The following tables map their normative requirements onto the lemmas that cover them.

Below RFC 9449 adoptations are listed (8.15) with corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
RFC 9449 (DPoP) — incorporated from flow	
§4.3 Verification checks	T2, R1
§6.1 `cnf.jkt` binding	T3, K2
§10 `dpop_jkt` thumbprint binding	B7
§11.1 Proof replay / context binding	R1

Table 8.15: RFC 9449 Security Properties and Connected Lemmas

Below RFC 9207 is listed 8.16 with its corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
RFC 9207 — Issuer Identification	
§2.4 Client-side issuer validation	ISS2, ISS3, ISS4, ISS5

Table 8.16: RFC 9207 Security Properties and Connected Lemmas

Below CIBA Core 1.0 is listed in Table 8.17 with its corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
CIBA Core 1.0 — incorporated from flow	
§7 Backchannel endpoint flow	CIBA1, CIBA2, CIBA3
§10 Poll-mode delivery	CIBA7, CIBA8
§11 Unsolicited authentication	CIBA6, CIBA9
§7 Auth-event binding to `auth_req_id`	CIBA1, CIBA4

Table 8.17: CIBA Core 1.0 Security Properties and Connected Lemmas

Below WebAuthn Level 3 is listed in Table 8.18 with its corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
WebAuthn Level 3 — incorporated from flow	
§7.2 Authentication ceremony	FIDO1, FIDO2, FIDO5, FIDO6
§7.2 `rpIdHash` verification	FIDO7, FIDO8
§7.2 Challenge freshness	FIDO9
§7.2 Credential key secrecy	FIDO3
§7.2 Credential required for authentication	FIDO4

Table 8.18: WebAuthn Security Properties and Connected Lemmas

Below `draft-ietf-oauth-attestation-based-client-auth` is listed in Table 8.19 with its corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
`draft-ietf-oauth-attestation-based-client-auth`	
Two-layer attestation (platform + CAT)	B1, B2, B3, B6
`cnf.jwk` binding to instance key	B7, K1, K2, K3
Key continuity across CAT / `dpop_jkt` / DPoP / `cnf.jkt`	K1, K2, K3, B4, B7
Attestation-authority key secrecy	S4
Attestation nonce single-use (anti-replay)	B5
§4.1 Token issuance requires completed authentication	E1

Table 8.19: Attestation-based client authentication Security Properties and Connected Lemmas

Below OAuth 2.1 is listed in Table 8.20 with its corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
OAuth 2.1 — core grant	
§4.1 Authorization code grant	Z1, Z2, Z3, Z4
§4.3 Refresh-token rotation	T5, E4, ISS5
§7.5.2 Single-use codes	Z2, Z4

Table 8.20: OAuth 2.1 Security Properties and Connected Lemmas

Below RFC 7662 is listed in Table 8.21 with its corresponding Tamarin lemmas:

Property	**Lemma(s) / mechanism**
RFC 7662 — Token Introspection	
RS → AS introspection	E2, E3

Table 8.21: Token-introspection Security Properties and Connected Lemmas

mAuth-Specific Properties

In addition to the standardised mechanisms above, mAuth introduces a small number of profile-specific hardenings. Each is verified by a dedicated lemma or combination thereof. Listed (8.22) below:

Property	Lemma(s) / mechanism
mAuth-specific hardenings	
Two-signature CAT wrapping	B6, S6
Reference-token / access-token split (back-end storage)	T1, T2, T3, K2
Explicit `dpop_jkt` check Hardware-backed instance key	B7 I1, I2
ROPC DPoP-wrapped credentials	ROPC1, ROPC4, ROPC5, ROPC6, ROPC7, ROPC8
ROPC password secrecy under DPoP envelope	ROPC3, ROPC7
FIDO `rpId` scoping as phishing defence	FIDO7, FIDO8
Separate FIDO challenge nonce	FIDO9
CIBA poll-mode single delivery	CIBA5, CIBA8
Client instance-key secrecy	S1, I1
ROPC completion requires explicit initiation	ROPC1, ROPC2
End-to-end ordering	F1, F2, F3

Table 8.22: mAuth-specific Security Properties and Connected Lemmas

Sufficiency Assessment

The mapping in shows that every applicable section of the normative security baselines either has at least one lemma establishing it symbolically or is out-of-scope. The variant-specific lemma series CIBA1–CIBA9, FIDO1–FIDO9, and ROPC1–ROPC8 close the surface that the common backbone lemmas (S, Z, T, R, B, K, I, E, ISS, F) do not address on their own. Within the scope this thesis claims, the lemma set covers every security property that is commonly known and relevant.

8.4 Results Discussion

The verification results can be summarised in one sentence: every lemma was verified by Tamarin under the smart heuristic, in each of the three theories `mAuth_CIBA`, `mAuth_FIDO`, and `mAuth_ROPC`.

The lemma results were expected. The lemmas were not constructed independently of the protocol, they were derived from the same specifications that governed the protocols design — and the protocol was, in turn, constructed by selecting mechanisms each of those documents recommends for each threat each of them enumerates. A run in which every lemma fails would have indicated that something had gone wrong in the encoding, a run in which most pass and a few fail would have shown a genuine gap in the design.

The gap that symbolic verification is designed to close is the gap between *a protocol that incorporates a mechanism the RFC says addresses threat* and *a protocol that actually addresses a threat when every interleaving of every rule, including the adversary, is enumerated.* Informal reasoning cannot in general distinguish those two. The Tamarin proofs do, because their quantification over all traces is exhaustive within the symbolic abstraction.

Three theories

The same lemmas hold across all three variant theories, with only the variant-specific lemmas differing, is itself a result. The common backbone — challenge, attestation, DPoP, token exchange, resource access, refresh — is identical in the three files, and the variant-specific Step 11 flows attach to it through the same structural points. The verification shows that the three ceremonies integrate cleanly with the backbone, that none of them weakens any backbone property, and that each discharges its own variant-specific lemmas without requiring help from the others. CIBAs polling does not break the single-issuance property of the authorization code, FIDOs `rpId` scoping does not break the issuer-binding properties of the access token, ROPCs DPoP-wrapped credentials do not break the secrecy properties of the client key.

The role of compromise events

Every security lemma is stated with an explicit disjunct admitting the relevant compromise event: `Reveal_Client_Key`, `Reveal_AA_Key`, `Reveal_Backend_Key`, the variant-specific reveals, or the matching `Leak_*` rule. The verified lemmas therefore do not state that the property holds unconditionally, but that it holds unless the named compromise has occurred. This is the precise characterisation the proofs deliver: T1 is meaningful because it identifies that the only way to use a stolen access token at the RS is to have also revealed the client key, B1 is meaningful because it identifies AA-key compromise as the only way to forge an attestation, S6 is meaningful because it identifies that the backends signing key is otherwise unknown to the symbolic adversary.

What the result does not say

The verification establishes that the symbolic encoding of the formalized mAuth satisfies the security properties the adopted standards require, within the scope of the symbolic model. It does not establish that a real implementation will hold: cryptographic strength, side-channel resistance, entropy quality, the correctness of the platform attestation APIs, the correctness of the TLS stack, and the operational hygiene of the AS. The App Backend, and the RS all sit below the symbolic abstraction and remain implementation concerns.

9

Conclusion

9.1 Conclusion

This thesis set out to answer the question: *What does mAuth require to be formalized, and once formalized against a set of normative standards, does it symbolically satisfy the security properties those standards require?*

A formalized mAuth was constructed as a profile of OAuth 2.1, with every protocol adaptation anchored to a published RFCs, FIDO2 / Webauthn recommendations, OpenID Foundation specifications. The formalized protocol was then encoded in Tamarin as three variant theories sharing an identical backbone, and a body of helper, sanity, and security lemmas drawn directly from the adopted standards.

The verification answers the research question within the scope the model claims. Every applicable section of the normative security baselines either has at least one lemma establishing it symbolically or is recorded as out-of-scope with a stated reason. Every adopted specification contributes lemmas verifying the property the document ascribes to it. The three variant ceremonies integrate cleanly with the common backbone without weakening any of its properties.

The thesis has therefore shown that, according to normative standards, mAuth satisfies the modeled security properties under the stated symbolic assumptions.

The work does not only contribute to a symbolically verified protocol, but also sets a blueprint for a real-world mAuth implementation.

9.2 Future Work

Several paths can be taken as a continuation on this work.

Usable implementation. The formalized protocol is specified at the level of detail required to write a conforming implementation. A reference codebase covering both the Authorization Server and the App Backend, together with client. Android and iOS exercising the platform attestation APIs, would make the profile usable as an actual deployment and would surface implementation-level concerns the symbolic

model abstracts away.

Performance evaluation. The protocol introduces multiple per-request signatures (DPoP at the authorization, token, refresh, and resource endpoints) and a two-signature attestation path at session establishment. A performance evaluation on representative mobile hardware, measuring latency,request overhead, and battery impact, would establish whether the security gains are delivered at a deployment cost meeting its value, and comparing the three variants against one another.

Implementation-level security testing. Symbolic verification establishes properties of the protocol, it does not establish properties of any particular implementation. A reference codebase as above would enable complementary analyses that operate below the symbolic layer: fuzzing of the JWT parsing and DPoP verification logic, side-channel analysis of the secure-element key operations, and attempts to bypass the platform attestation APIs through known emulator and rooted-device techniques.

Bibliography

[1] D. Hardt, *The OAuth 2.0 Authorization Framework*, RFC 6749, Oct. 2012. DOI: 10.17487/RFC6749. [Online]. Available: https://www.rfc-editor.org/info/rfc6749.

[2] D. Hardt, A. Parecki, and T. Lodderstedt, "The OAuth 2.1 Authorization Framework," Internet Engineering Task Force, Internet-Draft draft-ietf-oauth-v2-1-15, Mar. 2026, Work in Progress, 100 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/15/.

[3] A. T. Fredrik Hamrefors, "Mauth: Secure authorization and authentication protocol for native apps," Chalmers University of Technology, 2024.

[4] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*, RFC 9106, Sep. 2021. DOI: 10.17487/RFC9106. [Online]. Available: https://www.rfc-editor.org/info/rfc9106.

[5] V. Miller, "Elliptic curves and their use in cryptography," Nov. 1998.

[6] Apple Inc., *Protecting keys with the secure enclave*, https://developer.apple.com/documentation/security/protecting-keys-with-the-secure-enclave, Apple Developer Documentation, Security framework. Accessed: May 11, 2026.

[7] Android Developers, *Android keystore system*, https://developer.android.com/privacy-and-security/keystore. Accessed: May 11, 2026.

[8] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018. DOI: 10.17487/RFC8446. [Online]. Available: https://www.rfc-editor.org/info/rfc8446.

[9] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt, *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*, RFC 8705, Feb. 2020. DOI: 10.17487/RFC8705. [Online]. Available: https://www.rfc-editor.org/info/rfc8705.

[10] M. B. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT)*, RFC 7519, May 2015. DOI: 10.17487/RFC7519. [Online]. Available: https://www.rfc-editor.org/info/rfc7519.

[11] M. B. Jones, J. Bradley, and N. Sakimura, *JSON Web Signature (JWS)*, RFC 7515, May 2015. DOI: 10.17487/RFC7515. [Online]. Available: https://www.rfc-editor.org/info/rfc7515.

[12] M. B. Jones and J. Hildebrand, *JSON Web Encryption (JWE)*, RFC 7516, May 2015. DOI: 10.17487/RFC7516. [Online]. Available: https://www.rfc-editor.org/info/rfc7516.

[13] V. Bertocci, *JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens*, RFC 9068, Oct. 2021. DOI: 10.17487/RFC9068. [Online]. Available: https://www.rfc-editor.org/info/rfc9068.

[14] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. B. Jones, and D. Waite, *OAuth 2.0 Demonstrating Proof of Possession (DPoP)*, RFC 9449, Sep. 2023. DOI: 10.17487/RFC9449. [Online]. Available: https://www.rfc-editor.org/info/rfc9449.

[15] M. B. Jones, J. Bradley, and H. Tschofenig, *Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)*, RFC 7800, Apr. 2016. DOI: 10.17487/RFC7800. [Online]. Available: https://www.rfc-editor.org/info/rfc7800.

[16] M. B. Jones and N. Sakimura, *JSON Web Key (JWK) Thumbprint*, RFC 7638, Sep. 2015. DOI: 10.17487/RFC7638. [Online]. Available: https://www.rfc-editor.org/info/rfc7638.

[17] M. B. Jones, B. Campbell, and C. Mortimore, *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants*, RFC 7523, May 2015. DOI: 10.17487/RFC7523. [Online]. Available: https://www.rfc-editor.org/info/rfc7523.

[18] N. Sakimura, J. Bradley, M. B. Jones, B. de Medeiros, and C. Mortimore, *Openid connect core 1.0*, https://openid.net/specs/openid-connect-core-1_0.html, OpenID Foundation Specification, Dec. 2023.

[19] Google. "Play integrity api." Android Developers, Accessed: May 10, 2026. [Online]. Available: https://developer.android.com/google/play/integrity.

[20] Apple Inc. "Establishing your app's integrity." Apple Developer Documentation, Accessed: May 10, 2026. [Online]. Available: https://developer.apple.com/documentation/devicecheck/establishing-your-app-s-integrity.

[21] N. Sakimura, J. Bradley, and N. Agarwal, *Proof Key for Code Exchange by OAuth Public Clients*, RFC 7636, Sep. 2015. DOI: 10.17487/RFC7636. [Online]. Available: https://www.rfc-editor.org/info/rfc7636.

[22] G. F. Rodriguez, F. Walter, A. Nennker, D. Tonge, and B. Campbell, *Openid connect client-initiated backchannel authentication flow - core 1.0*, https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html, OpenID Foundation Specification, Sep. 2021.

[23] D. Fett, D. Tonge, and J. Heenan, *Fapi 2.0 security profile*, https://openid.net/specs/fapi-security-profile-2_0-final.html, OpenID Foundation Final Specification, Feb. 2025.

[24] A. Backman, J. Richer, and M. Sporny, *HTTP Message Signatures*, RFC 9421, Feb. 2024. DOI: 10.17487/RFC9421. [Online]. Available: https://www.rfc-editor.org/info/rfc9421.

[25] J. Richer, M. B. Jones, J. Bradley, M. Machulak, and P. Hunt, *OAuth 2.0 Dynamic Client Registration Protocol*, RFC 7591, Jul. 2015. DOI: 10.17487/RFC7591. [Online]. Available: https://www.rfc-editor.org/info/rfc7591.

[26] M. B. Jones, N. Sakimura, and J. Bradley, *OAuth 2.0 Authorization Server Metadata*, RFC 8414, Jun. 2018. DOI: 10.17487/RFC8414. [Online]. Available: https://www.rfc-editor.org/info/rfc8414.

[27] J. Richer, *OAuth 2.0 Token Introspection*, RFC 7662, Oct. 2015. DOI: 10.17487/RFC7662. [Online]. Available: https://www.rfc-editor.org/info/rfc7662.

[28] T. Lodderstedt, S. Dronia, and M. Scurtescu, *OAuth 2.0 Token Revocation*, RFC 7009, Aug. 2013. DOI: 10.17487/RFC7009. [Online]. Available: https://www.rfc-editor.org/info/rfc7009.

[29] K. M. zu Selhausen and D. Fett, *OAuth 2.0 Authorization Server Issuer Identification*, RFC 9207, Mar. 2022. DOI: 10.17487/RFC9207. [Online]. Available: https://www.rfc-editor.org/info/rfc9207.

[30] B. Campbell, J. Bradley, and H. Tschofenig, *Resource Indicators for OAuth 2.0*, RFC 8707, Feb. 2020. DOI: 10.17487/RFC8707. [Online]. Available: https://www.rfc-editor.org/info/rfc8707.

[31] T. Lodderstedt, J. Richer, and B. Campbell, *OAuth 2.0 Rich Authorization Requests*, RFC 9396, May 2023. DOI: 10.17487/RFC9396. [Online]. Available: https://www.rfc-editor.org/info/rfc9396.

[32] T. Looker, P. Bastian, and C. Bormann, "OAuth 2.0 Attestation-Based Client Authentication," Internet Engineering Task Force, Internet-Draft draft-ietf-oauth-attestation-based-client-auth-08, Mar. 2026, Work in Progress, 30 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-oauth-attestation-based-client-auth/08/.

[33] World Wide Web Consortium (W3C), *Web authentication: An api for accessing public key credentials level 3*, https://www.w3.org/TR/webauthn-3/, 2026.

[34] J. Bradley, M. B. Jones, A. Kumar, R. Lindemann, J. Verrept, and D. Waite, *Client to authenticator protocol (ctap)*, https://fidoalliance.org/specs/fido-v2.2-ps-20250714/fido-client-to-authenticator-protocol-v2.2-ps-20250714.html, FIDO Alliance Proposed Standard, Jul. 2025.

[35] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983. DOI: 10.1109/TIT.1983.1056650.

[36] D. Basin, C. Cremers, J. Dreier, and R. Sasse, *Tamarin: Symbolic verification (proving) and falsification (attack finding) for security protocols*, https://tamarin-prover.com/, 2012–present.

[37] Tamarin Team, *Tamarin prover manual*, https://tamarin-prover.com/manual/master/book/001_introduction.html, 2025.

[38] B. Blanchet, "The security protocol verifier proverif and its horn clause resolution algorithm," *Electronic Proceedings in Theoretical Computer Science*, vol. 373, pp. 14–22, Nov. 2022, ISSN: 2075-2180. DOI: 10.4204/eptcs.373.2. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.373.2.

[39] M. B. Jones, *JSON Web Key (JWK)*, RFC 7517, May 2015. DOI: 10.17487/RFC7517. [Online]. Available: https://www.rfc-editor.org/info/rfc7517.

[40] Y. Sheffer, D. Hardt, and M. B. Jones, *JSON Web Token Best Current Practices*, RFC 8725, Feb. 2020. DOI: 10.17487/RFC8725. [Online]. Available: https://www.rfc-editor.org/info/rfc8725.

[41] Android Open Source Project, *Hardware security best practices*, https://source.android.com/docs/security/best-practices/hardware. Accessed: May 11, 2026.

[42] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, *Best Current Practice for OAuth 2.0 Security*, RFC 9700, Jan. 2025. DOI: 10.17487/RFC9700. [Online]. Available: https://www.rfc-editor.org/info/rfc9700.

[43] S. Strömbäck Olofsson and J. Hermenius, *Mauth-tamarin-model-and-proofs*, https://github.com/L3B0W5K1/mAuth-Tamarin-Model-and-Proofs/, 2026.

[44] World Wide Web Consortium (W3C), *Verifiable credentials data model 2.0*, https://www.w3.org/TR/vc-data-model-2.0/. Accessed: May 11, 2026.

[45] Curity AB, *Hypermedia authentication api (haapi)*, https://curity.io/resources/haapi/. Accessed: May 11, 2026.

[46] Microsoft, *Microsoft entra*, Accessed: 2026-05-11, 2026. [Online]. Available: https://www.microsoft.com/en-us/security/business/microsoft-entra.

[47] Y. Zehavi, "OAuth 2.0 App2App Browser-less Flow," Internet Engineering Task Force, Internet-Draft draft-zehavi-oauth-app2app-browserless-07, Oct. 2025, Work in Progress, 22 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-zehavi-oauth-app2app-browserless/07/.

[48] A. Parecki, G. Fletcher, and P. Kasselman, "OAuth 2.0 for First-Party Applications," Internet Engineering Task Force, Internet-Draft draft-ietf-oauth-first-party-apps-03, Feb. 2026, Work in Progress, 41 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-oauth-first-party-apps/03/.

[49] P. Hosseyni, R. Küsters, and T. Würtele, "Formal security analysis of the openid fapi 2.0: Accompanying a standardization process," in *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, Jul. 2024, pp. 589–604. DOI: 10.1109/CSF61375.2024.00002.

[50] D. Fett, R. Kuesters, and G. Schmitz, *An expressive model for the web infrastructure: Definition and application to the browserid sso system*, 2014. arXiv: 1403.1866 [cs.CR]. [Online]. Available: https://arxiv.org/abs/1403.1866.

A

mAuth RFC

mAuth RFC

Jesper Hermenius, Simon Strömback Olofsson

June 22, 2026

This document is a work in progress. It does not represent a final design or an official commitment to implement. Content, scope, and technical details are subject to change.

Abstract

mAuth defines a modular OAuth-based protocol that combines authentication, attestation, and proof-of-possession mechanisms to strengthen client trust.

Contents

1 Introduction

In traditional oAuth flows the client is redirected using a browser which creates a hard to follow flow for a mobile native application where the user is not sure what is happening. MAuth aims to solve this issue and the following document describes the technical implementations to do so.

2 Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119]

3 Architecture

This chapter describes the different parts that the protocol is designed around/with.

3.1 Entities

For mAuth there are 4 basic entities to keep in mind when trying to understand the protocol. Implementation specific entities SHOULD be considered as part of one of these 4 entities and if that is not possible the protocol MAY NOT work as intended. The entities can be divided into the following:

- User/Resource owner
- Client(frontend and backend)
- Authorization server
- Resource server/API

3.2 Clients

Due to the nature of mobile devices and native apps not being able to keep client secrets the client backend MUST register as *confidential* with `token_endpoint_auth_method`. The frontend MUST identify itself through the use of attestation and DPoP as seen in [RFC 6749] and [The OAuth 2.1 Authorization Framework draft-ietf-oauth-v2-1-15].
The client MUST be issued a unique string client identifier by the authorization server. This identifier is not a secret and MAY be used in attestation requests.

3.3 refresh token grant

Refresh tokens issued to the composite mAuth client MUST be sender-constrained with DPoP (RFC 9449 §5). Refresh-token rotation (OAuth 2.1 §4.3.3) is RECOMMENDED, as the App Backend is a confidential client and can safely handle rotation. All tokens MUST be sender constrained and the sender MUST show proof of knowledge of a certain secret through the use of DPoP.[RFC9449]

3.4 Access-token scope and sender-constraining

All access tokens issued to mAuth clients MUST be sender-constrained.

3.4.1 Protocol endpoints

Token endpoint (`/token`), authorization endpoint (present for metadata discovery but *never* used through a browser on the consumption device), and the application of `application/x-www-form-urlencoded` bodies.

3.5 Generating DPoP proofs

All generated DPoP proofs MUST use the following in their JOSE header and payload [RFC 9449]:

`typ` Fixed value `"dpop+jwt"`. This explicit typing (RFC 8725 §3.11) prevents cross-JWT confusion.

`alg` An asymmetric JWS algorithm from the IANA JOSE registry, mAuth REQUIRES `ES256` as the baseline (matches the P-256 key generated on both Android KeyStore and iOS Secure Enclave). Values of `none` or any MAC algorithm MUST NOT be used.

`jwk` The public part of the client instance key pair, in JWK form. MUST NOT contain a private key.

`jti` $\geq$ 96 bits of entropy, unique per proof. Used by the AS/RS for replay detection.

`htm` The HTTP method of the target request.

`htu` The target URI, with query and fragment stripped.

`iat` Issuance timestamp.

`ath` Base64url-encoded SHA-256 hash of the associated access token.

`nonce` Most recent value of the server-issued nonce.

3.6 Validating DPoP Proofs

When validating any DPoP proof the reciever MUST perform all 12 checks as described in chapter 4.3 of [RFC9449]. The steps are listed here for convenience in a condensed manner. The steps MAY be performed in any order but the communication MUST not proceed unless all checks pass

- Exactly one DPoP http header field which MUST be a single well formed JWT
- All claims that are required MUST be present.
- The JOSE header parameter `typ` MUST take on the value `dpop+jwt`and `alg` MUST be supported by the application, NOT empty/none, and must be a valid algorithm for assymetric digital signature according to IANA.
- The jwk JOSE header containing the public key MUST verify the JWT signature
- The `htm` and `htu` claims MUST match their HTTP method and URI respectively.
- Any nonce value sent to the client by the server MUST match the nonce claim.

- The lifetime of the JWT MUST be within the allowed tolerance window.
- If an access token is present `ath` MUST match the hash of the access token and the public key from the access token MUST also match the public key from the DPoP proof.

3.7 Token revocation

The client can request to have its tokens revoked through a request to the authorization server. The AS SHOULD handle the revocation as quickly as possible. If a refresh token is revoked any tokens issued from it MUST also be revoked. If a access token is revoked the associated refresh token SHOULD also be revoked.

3.8 Token Introspection

The resource server SHOULD consult the authorization server for token introspection. The authorization server MUST return metadata about a token if properly requested.

3.9 Device attestation

The authorization server MUST be able to distinguish a client identity from a end user identity through the use of device attestation.

3.10 Resource indicators

In a request to the authorization server the client MUST include the resource parameter binding the access token to a specific resource including the intended resource path.

3.11 JWT

Every JWT in mAuth SHOULD follow the practices stated in [RFC 8725].

3.12 Authorization server issuer identification

Any authorization server response to a client MUST include the `iss` parameter to identify itself.

3.13 Trust Model

This section defines the trust assumptions and relationships between entities participating in mAuth.

MAuth involves the following entities: clients, authorization servers, resource servers, and resource owners. Each entity operates within a defined trust boundary and is responsible for maintaining the security of its own credentials and operations.

Clients trust the authorization server to correctly authenticate identities and to issue access tokens that accurately represent the granted authorization. Clients MUST validate responses from the authorization server and MUST ensure that they only interact with trusted authorization endpoints.

Resource servers trust the authorization server to issue valid and unforgeable access tokens. Resource servers MUST validate access tokens and MUST ensure that tokens are intended for their use before granting access to protected resources.

Authorization servers trust clients to protect their credentials and private keys. Confidential clients MUST securely store credentials, while public clients MUST use alternative mechanisms such as PKCE. The authorization server MUST NOT assume that public clients can maintain secrecy.

Communication between parties is assumed to occur over potentially untrusted networks. Implementations MUST use secure transport mechanisms, such as TLS, to provide confidentiality and integrity protection. Implementations MUST NOT rely on unprotected transport.

The trust model assumes that private cryptographic keys are not compromised. A compromise of such keys invalidates the guarantees provided by the system and MUST result in immediate revocation and replacement.

Adversaries are assumed to be capable of observing, intercepting, and modifying network traffic, and MAY attempt to impersonate clients or replay messages. Implementations MUST include protections against these threats as described in the Security Considerations section.

Only the clients backend communicates with the authorization server and never the client frontend.

3.14 Threat Model

This section defines the threat model for mAuth by describing the capabilities of potential adversaries and the types of attacks the protocol is expected to resist.

Adversaries are assumed to have full control over the network and MAY observe, intercept, modify, and replay messages.

Adversaries MAY attempt to impersonate legitimate clients or authorization servers. This includes obtaining client credentials, forging requests, or redirecting communication to malicious endpoints.

Adversaries MAY gain access to access tokens through leakage, interception, or compromise of client systems. Such adversaries MAY attempt to use stolen tokens to access protected resources.

Adversaries MAY attempt replay attacks by capturing valid protocol messages and retransmitting them at a later time. They MAY also attempt to reuse authorization requests or tokens outside their intended context.

Adversaries MAY control malicious clients that interact with the authorization server. Such clients MAY deviate from the protocol, request excessive privileges, or attempt to exploit implementation weaknesses.

Adversaries MAY exploit implementation flaws, including improper validation of inputs, insufficient randomness, or insecure storage of credentials and keys.

The threat model assumes that cryptographic primitives are secure and correctly implemented. However, adversaries MAY exploit weak key management practices, including poor key storage or failure to rotate compromised keys.

Physical attacks and full compromise of trusted execution environments are considered out of scope. However, partial compromise, such as leakage of application memory or logs, is considered within scope.

Implementations MUST consider these adversarial capabilities when designing and deploying mAuth systems.

4 Protocol Overview

4.1 Setup

There are some setup steps that need to be performed in order to make use of the protocol and they can be seen here. Some steps are needed regardless of which variant of the protocol is used and some are variant specific. All setup creating metadata follows [RFC 7591] meaning the metadata is stored at the authorization server.

4.1.1 General preparation

The clients MUST have a unique `client_id`, this will be different for every authorization server. A `grant_types` containing the refresh token. A `token_endpoint_auth_method` set to `private_key_jwt`. The `jwks` MUST be registered. The metadata MUST include `dpop_bound_access_tokens=true`. The AS MUST keep the canonical resource URLs that the client is permitted to target in the `Registered resource URIs`.

4.1.2 Variant specific preparations

CIBA variant

- `grant_types` additionally contains `urn:openid:params:grant-type:ciba`
- `backchannel_token_delivery_mode` MUST be set to `poll`.
- `backchannel_authentication_request_signing_alg` MUST be an asymmetric JWS algorithm.

FIDO2 variant

- `grant_types` additionally contains `authorization_code`.
- `response_types` MUST be `code` [RFC6749].

ROPC variant

- `grant_types` MUST additionally contain `password` [RFC6749].
- `first_party` flag MUST be set.

4.1.3 General authorization server metadata discovery

The discovery process for the metadata is done by the client backend through a GET request to `/.well-known/mauth-authorization-server`. This metadata MUST include.

- `issuer` MUST equal the expected issuer URL.
- `token_endpoint`, `introspection_endpoint` [RFC7662], `revocation_endpoint` [RFC7009], and `challenge_endpoint` `draft-ietf-oauth-attestation-based-client-auth` [ietf-oauth-attestation-based-client-auth-08] are HTTPS URLs.
- `dpop_signing_alg_values_supported` MUST contain `ES256` [RFC9449].
- `grant_types_supported` MUST contain the union of grants required by the variants this AS deployment offers.
- `authorization_details_types_supported` [RFC9396].
- `token_endpoint_auth_methods_supported`, `revocation_endpoint_auth_methods_supported`, and `introspection_endpoint_auth_methods_supported` MUST all contain `private_key_jwt`.
- `client_attestation_signing_alg_values_supported` [RFC draft-ietf-oauth-attestation-based-client-auth].
- `signed_metadata` [RFC8414].
- `authorization_response_iss_parameter_supported` MUST be `true` [RFC9207]

CIBA variant.

- `backchannel_authentication_endpoint` MUST be an HTTPS URL [oidc].
- `backchannel_token_delivery_modes_supported` MUST contain `poll`.

FIDO2 variant only.

- `authorization_endpoint` is a HTTPS URL — targeted by the FIDO2 variant in step 9 for WebAuthn [webauthn] challenge parameters.

ROPC variant only. The ROPC variant introduces no new metadata: it reuses `token_endpoint` from the common set, since RFC 6749 §4.3 [RFC6749] carries the password grant directly there.

4.2 High-Level Flow

Step 1

The user starts the authorisation/authentication process by pressing a login button. At this stage no user credentials have been submitted.

Step 2

The client frontend sends an attestation challenge to the backend which in turn sends it forward to the authorisation server.

If the client type is confidential, the client and authorization server establish a client authentication method suitable for the security requirements of the authorization server. The authorization server MAY accept any form of client authentication meeting its security requirements.

Confidential clients are typically issued (or establish) a set of client credentials used for authenticating with the authorization server (e.g., password, public/private key pair.

The authorization server MAY establish a client authentication method with public clients. However, the authorization server MUST NOT rely on public client authentication for the purpose of identifying the client.

The client MUST NOT use more than one authentication method in each request.

Step 3

The authorization server then responds with an authorization challenge. This is done in the form of a cryptographic nonce to prevent replay attacks. The client MUST adhere to the JWT best practices described in RFC8725 and the nonce that is generated MUST have a minimum of 128 bits of entropy so that an attacker cannot guess the value with any computational feasibility.

Credentials not intended for handling by end-users (e.g., access tokens, refresh tokens, authorization codes, etc.) shall be created with at least 128 bits of entropy such that an attacker correctly guessing the value is computationally infeasible. [RFC6749].

Step 4

The client then generates a new client instance key pair that is used for the DPoP process. This keypair MUST be securely stored on the device and is thus not possible if the device does not meet the security standards of the KeyStore system on android or the Secure Enclave on ios(https://developer.android.com/privacy-and-security/keystore
https://developer.apple.com/documentation/security/protecting-keys-with-the-secure-enclave).
The client generates a keypair using its respective system and converts the public key to a JWK following RFC 7517.

Step 5

The client makes a request to the attestation system of their device(KeyStore or Secure enclave). The request MUST contain the execution environment, operating system and the public key from the previous step.

Step 6

Depending on if you use android or ios the process will look slightly different. This is described below.

- Android
  The Android attestation service verifies the information that the client provides it and if it passes all checks it will return a signed certificate digest.
- IOS
  The Apple app attest and device check will check that the device and application is valid and then produce an attestation statement that it signs. The statement binds the application id, the device and the value provided by the previous step.

Step 7

The attestation service for the relevant operating system MUST return a attestation statement to the client. If the client is on an android device the expected return is a JWE-wrapped JWS integrity verdict token and if the client is on IOS it is a CBOR object encoding the attestation. The client SHOULD stop and abort any authentication if the returned object is not of the expected type. Details about the returned objects can be found here Android:

Step 8

The client frontend generates a unique DPoP Proof (DPoPP) JWT that proves that the client is in possession of the private key generated in step 4 The format of the different platforms attestation statements differ and as such both types MUST be wrapped with a client attestation JWT in accordance with [RFC8725] explicit typing using a JOSE header with `typ` set to dpop+jwt. The `alg` header MUST be set to ES256.

Step 9

The client makes an authorization code request to the authorization server via the backend server. In this request, it attaches the attestation JWT and adds the DPoPP in the HTTP header. The format of the request to the authorization server MUST be either CIBA, FIDO, or ROPC. The request MUST contain a `scope` and `resource` and MAY contain a `authorization_details`.

The attestation MUST follow the following list of things:

Client backend using authentication with `private_key_jwt`. This jwt MUST contain `client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer`. The header in the HTTP request MUST be in a token68 format and labeled OAuth-Client-Attestation. It also MUST use the DPoP HTTP header which has the proof generated in step 8, the dpop_jkt with its JWK SHA-256 digest of the public key.
From here the flow will differ depending on the configuration. for details regarding the chosen variation check the table below.

Variant	Information
CIBA Variant	Posts `application/x-www-form-urlencoded` to the `backchannel_authentication_endpoint`. Includes one of `login_hint`, `id_token_hint`, or `login_hint_token`. May include `acr_values`, `binding_message`, and `requested_expiry`. Uses a signed authentication request containing claims `iss`, `aud`, `iat`, `exp`, `nbf`, and `jti`, signed using the registered `backchannel_authentication_request_signing_alg`. Supports DPoP and `dpop_jkt` as profile-specific extensions of RFC 9449 §10.
FIDO2 Variant	Posts to the AS `authorization_endpoint` with the common envelope and parameter `auth_method=fido2`, indicating that an assertion exchange is requested. No user hint is required at this stage because the AS resolves the user from the credential selected by the authenticator in step 11. The AS responds with a `PublicKeyCredentialRequestOptions` structure delivered via the backend.
ROPC Variant	Posts to the AS `token_endpoint` with the common envelope plus `grant_type=password`. Credentials are not sent in the form body. Instead, the DPoP proof from step `(8)` carries inline credentials as `credentials = { username, password_hash }`, where `password_hash` is an Argon2id digest computed on-device using AS-published parameters. The DPoP signature binds the credentials to the instance key. Intended for first-party use only.

Table 1: Authentication Variant Comparison

Step 10

The authorization server verifies the JWT and DPoPP and, if approved, sends a message back to the client via the backend server that says that the client has been attested and needs to provide login information. If the variation is ROPC clients MUST have set the `first_party` flag in the setup and if they have not done so the session MUST be refused.

Step 11

Here, one of the three different flows can be used depending on user preference. At the end of this step, the app backend gets an authorization code that it can use for the token exchange. The different flows are described in detail in section 7. The choice is between CIBA, FIDO, and ROPC.

Variant	**Description**
CIBA	Out-of-band authentication flow. AS returns auth_req_id, expires_in, and interval. The AS triggers user authentication via an external app (e.g., BankID), presenting binding_message and scope/RAR summary through the use of WebAuthn. The user approves or denies consent. The client polls the token_endpoint using grant_type=urn:openid:params:grant-type:ciba and auth_req_id, receiving states such as authorization_pending, slow_down, expired_token, or access_denied. On success, the flow proceeds to token issuance. **(11-CIBA) Client-Initiated Backchannel Authentication** Used when authentication must occur out-of-band (e.g., BankID). **(11a-CIBA) Backchannel acknowledgement.** The AS responds to the request issued in step (9) with HTTP 200 carrying `auth_req_id`, `expires_in`, and `interval` [OpenID Connect Core 1.0]. **(11b-CIBA) Out-of-band user authentication.** The AS contacts the users authentication app and presents `binding_message` together with the merged `scope`/RAR summary. The user authenticates via BankID and grants or denies consent. **(11c-CIBA) Polling loop.** The App Backend POSTs to `token_endpoint` with `grant_type=urn:openid:params:grant-type:ciba` and `auth_req_id`. While authentication is in progress the AS returns `error=authorization_pending`, excessively fast polling yields `slow_down` with a new minimum interval. Terminal errors are `expired_token` and `access_denied`. On success the flow proceeds to step (13).

Table 2: Authentication Variant Overview

Variant	Description
F2 (FIDO2 / WebAuthn)	Passwordless authentication using passkeys. The AS returns PublicKeyCredentialRequestOptions including challenge, rpId, allowCredentials, and userVerification=required. The client invokes platform authenticator APIs (Android CredentialManager / iOS ASAuthorization). The authenticator performs user verification and signs the challenge using the credential private key, returning an assertion. The AS verifies origin, challenge, rpIdHash, flags, and signature. On success, an authorization code is issued. **(11-FIDO2) WebAuthn / FIDO2** **(11a-F2) Assertion options.** The AS responds to the step-(9) request with a `PublicKeyCredentialRequestOptions` structure (W3C WebAuthn Level 3 [webauthn]): • `challenge` — fresh $\geq$ 16-byte random value. • `rpId` — the AS registrable domain matched to iOS Associated Domains and Android Digital Asset Links for the app. • `allowCredentials` — list of `credentialId`s previously registered for the user. • `userVerification` set to `required`. • `timeout`. Delivered to the App Frontend via the backend.

Table 3: Authentication Variant Overview fido part1

Variant	Description)
F2 (FIDO2 / WebAuthn) (continuation	**(11b-F2) Native API invocation.** The frontend invokes the platform passkey API (Android `CredentialManager.getCredential` with `GetPublicKeyCredentialOption` [android keystore]; iOS `ASAuthorizationPlatformPublicKeyCredentialProvider` [ios secure enclave]). The platform WebAuthn Client constructs `clientDataJSON` with `type="webauthn.get"`, the challenge, and the origin (for a native app, the app-origin bound via Associated Domains / Digital Asset Links to the AS `rpId`), passes it via FIDO CTAP 2.2 [FIDO CTAP] to the authenticator, which: 1. Looks up the credential private key scoped to `rpId`. 2. Performs user verification, sets the UV flag. 3. Increments `signCount`. 4. Returns `authenticatorData` (rpIdHash, flags, signCount) and a signature over `authenticatorData` ‖ `SHA-256(clientDataJSON)`. **(11c-F2) Assertion response.** The frontend forwards the `AuthenticatorAssertionResponse` via the backend to the AS. **(11d-F2) AS verification.** The AS performs the WebAuthn Level 3 verification algorithm: 1. `clientDataJSON.type` equals `"webauthn.get"`. 2. `challenge` matches the stored value and has not been previously consumed. 3. `origin` is in the set of origins permitted for the clients app-origin. 4. `rpIdHash` equals `SHA-256(rpId)`. 5. UP flag is set, UV flag is set when `userVerification=required`. 6. `signCount`, if non-zero, is greater than the stored value; otherwise the credential is flagged as potentially cloned. 7. Signature verifies over `authenticatorData` ‖ `SHA-256(clientDataJSON)` against the stored credential public key. On success, the AS records the authenticated subject and issues an authorization `code` that is returned to the backend; the flow proceeds to step (12).

Table 4: Authentication Variant Overview

Variant	Description
ROPC	First-party password-based flow (deprecated). The user enters username and password in the client UI. Credentials are hashed on-device (password_hash, Argon2id) and embedded in a DPoP proof instead of being sent directly. The AS verifies DPoP binding, credential hash, and performs additional checks such as second-factor verification. On success, tokens are issued; on failure, invalid_grant is returned. **(11a-R) Credential prompt.** The App Frontend prompts the user for `username` and `password` inside the apps own UI. **(11b-R) DPoP-wrapped credentials.** Rather than transmitting credentials as plain form parameters, the frontend embeds them inside the DPoP proof of step (8) as an additional payload claim `credentials = { username, password_hash }`, where `password_hash` is the Argon2id [argon2id] digest computed on-device with AS-published parameters. The DPoP signature, combined with TLS 1.3 confidentiality, is the primary protection in transit. **(11c-R) AS verification.** 1. Verifies the DPoP proof [RFC9449]). 2. Confirms thumbprint match to the instance key established in step (10) via the Client Attestation `cnf.jwk`. 3. Verifies `password_hash` against the stored credential record for `username`, with a server-side keyed comparison (HMAC over a server pepper). 4. Performs a second-factor challenge (e.g., emailed OTP) before issuing tokens. **(11d-R) Token issuance.** On success, because the ROPC request was already targeted at `token_endpoint` in step (9), the AS issues the token bundle of step (13) directly in the same response. Refusal returns `error=invalid_grant` (RFC 6749 [RFC6749]).

Table 5: Authentication Variant Overview

Step 12

The backend server makes a token request by sending the authorization code to the authorization server. The authorization code is itself not a sender-constrained, it carries no cnf claim and is not presented with a proof-of-possession, but it is bound to the client instance key by the dpop_jkt parameter. The token request is carried over the backchannel between the backend server and the authorization server and must therefore present a fresh DPoP proof signed by the same instance key. The authorization server recomputes the thumbprint from the proof's public key and rejects the request unless it matches the dpop_jkt value recorded with the code. The authorization code is thus constrained to the client instance through dpop_jkt, even though the code itself is not a

DPoP-bound bearer artifact.

Step 13

The authorization server responds with DPoP-constrained access/ID- and reference tokens and MAY respond with a refresh token.

Step 14

The backend server sends the reference token to the client frontend, which the frontend can use to access the access/ID token or refresh token stored on the backend server. The client frontend MUST verify the received tokens to make sure they are from the correct issuer and if the verification fails it MUST abort the session and revoke the tokens.

Step 15

The client frontend generates a new unique DPoPP that proves that the client is still in possession of the private key generated in step 4

Step 16

The client frontend makes a resource request to the backend server with the DPoPP JWT to retrieve some data from the resource server using the reference token.

Step 17

The backend changes the reference token for the actual access/ID token and sends it along with the rest of the request to the API.

Step 18

The API sends the tokens to the authorization server along with the DPoPP to verify them.

Step 19

The resource server responds with the requested data that is forwarded from the backend server to the client frontend.

5 Interchangeable Authentication Methods

The protocol has a modular part in step 11 where one of three different flows needs to be selected to be used.

5.1 CIBA

CIBA (Client Initiated Backchannel Authentication) is a flow where the user authentication doesn't necessarily happen on the same device that is being authenticated for. A common example of this are authenticator apps when used while logging in on a computer.

5.2 FIDO2

FIDO2 (Fast Identity Online 2) Allows users to authenticate using biometrics or external hardware passkeys. Common examples of this are face scan, fingerprint sensor, or a product such as yubikey.

5.3 ROPC

ROPC (Resource Owner Password Credentials) is an authentication method where the user hands over its credentials(usually a username and password) to the application which then uses them to acquire authorization tokens for the resource server. This means there is no separate login for the user and trades security for convenience.

6 Security Considerations

MAuth supports multiple implementation options, and these introduce security considerations that MUST be addressed by any conforming implementation. The following sections describe relevant threats and required mitigations using normative language as defined in RFC 2119.

6.1 OAuth-Derived Threats

As mAuth is based on OAuth, it inherits several of its security considerations. Implementations MUST account for these inherited risks and apply appropriate mitigations.

6.1.1 Client Authentication

Clients MUST ensure that their credentials are stored and handled securely. Clients that cannot guarantee the confidentiality of their credentials, such as public clients, MUST use alternative mechanisms like Proof Key for Code Exchange (PKCE). Authorization servers MUST authenticate clients whenever possible. Since redirect-based flows are not used in mAuth, recommendations specific to such flows do not apply.

6.1.2 Client Impersonation

Authorization servers MUST protect against client impersonation. If a malicious actor obtains confidential client credentials, that actor MAY impersonate a legitimate client. To mitigate this risk, authorization servers MUST authenticate clients whenever possible and SHOULD avoid automatically processing repeated or suspicious authorization requests. Clients SHOULD ensure that credentials are protected against leakage and unauthorized access.

6.1.3 Access Tokens

Access tokens and their associated metadata MUST be kept confidential both in transit and at rest, and they MUST only be shared with the authorization server and the intended resource servers. Authorization servers MUST ensure that access tokens cannot be forged, modified, or guessed by unauthorized parties and MUST implement sufficient entropy and integrity protection mechanisms. Clients MUST request access tokens with the minimal required scope, and authorization servers MAY issue tokens with reduced scope compared to what was requested.

6.1.4 Resource Owner Password Credentials (ROPC)

The use of Resource Owner Password Credentials introduces significant risk and SHOULD be avoided unless absolutely necessary. When used, clients MUST NOT request broader scope than required and MUST protect user credentials from leakage, including through logs or improper storage. Since the resource owner effectively places full trust in the client, such clients MUST be treated as highly trusted components. Authorization servers SHOULD discourage or restrict the use of ROPC.

6.1.5 Request Confidentiality

Sensitive information, including tokens, credentials, state parameters, and scope values, MUST NOT be transmitted in plaintext and MUST be protected using secure transport mechanisms such as TLS. Implementations SHOULD minimize the exposure of metadata that could reveal information about system behavior or internal structure.

6.1.6 Clickjacking

Authorization endpoints are susceptible to clickjacking attacks, where a user is tricked into interacting with a hidden authorization interface. Authorization servers MUST mitigate this risk by preventing their authorization pages from being embedded in iframes where possible. They SHOULD employ frame protection mechanisms such as the `X-Frame-Options` header with values like `DENY` or `SAMEORIGIN`. Additional frame-busting techniques MAY be used for compatibility with older browsers, although these techniques are not fully reliable. Native clients SHOULD use external user agents, such as system browsers, rather than embedded web views when initiating authorization requests.

6.2 Token Theft

Access tokens MAY be stolen through a variety of attack vectors. Authorization servers MUST ensure that tokens are only accepted from the client to which they were issued and SHOULD bind tokens to client identity or contextual information when possible. Clients MUST protect tokens during storage and transmission and MUST NOT expose them to unauthorized components or third parties.

6.3 Replay Attacks

Replay attacks occur when valid requests are captured and reused by an attacker. Authorization servers MUST detect and reject replayed requests. To support this, requests SHOULD include timestamps and unique identifiers such as nonces, and authorization servers SHOULD enforce strict validity windows to limit the usefulness of intercepted messages.

6.4 Key Compromise

The security of mAuth depends on the confidentiality and integrity of cryptographic keys. Clients MUST securely generate and store private keys and SHOULD use hardware-backed storage mechanisms, such as secure enclaves or trusted platform modules, when available. Private keys MUST NOT be embedded in application code or distributed in a manner that allows unauthorized access.

Authorization servers MUST support key rotation, enforce key expiration policies, and provide

mechanisms for immediate revocation in the event of a suspected compromise. All parties MUST replace compromised keys without delay and SHOULD monitor for indications of key compromise.

Failure to adhere to these requirements MAY result in unauthorized access, impersonation, or compromise of protected resources.

7 Privacy Considerations

MAuth implementations process sensitive information related to users, clients, and authorization decisions. The following subsections outline key privacy considerations.

7.1 Data Minimization

Implementations should ensure that only the minimum amount of personal data necessary for a given operation is collected, transmitted, and stored. Access tokens and associated scopes should be limited to only what is required for the intended functionality.

Authorization servers should avoid including unnecessary user attributes in tokens or responses. Where possible, opaque identifiers should be used instead of directly identifying information.

7.2 User Identifiability

Persistent identifiers can allow tracking of users across services and over time. To mitigate this, systems should avoid reusing the same identifiers across different resource servers unless strictly necessary.

Where possible, pairwise or per-client identifiers should be used to prevent correlation of user activity between different services.

7.3 Token Content and Leakage

Access tokens and other credentials may contain sensitive information, either directly or indirectly. If tokens are exposed (e.g., through logs, browser history, or network interception), this may lead to privacy breaches.

Tokens SHOULD therefore be treated as confidential and should not contain personally identifiable information unless strictly necessary. Encryption or the use of opaque tokens is recommended to reduce the risk of information leakage.

7.4 Logging and Monitoring

Logging mechanisms MAY inadvertently capture sensitive data such as access tokens, user identifiers, or request parameters. Improper handling of logs can lead to large-scale privacy violations.

Implementations should sanitize logs to ensure that sensitive data is either removed or masked. Access to logs MUST be restricted and retention periods should be minimized in accordance with applicable data protection requirements.

7.5 Third-Party Dependencies

MAuth deployments may rely on third-party services such as attestation providers or identity services. These parties may process user-related data, introducing additional privacy risks.

Implementations should carefully evaluate the data shared with third parties and ensure that such sharing complies with relevant privacy regulations. Data processing agreements and clear data handling policies should be established.

7.6 User Consent and Transparency

Users SHOULD be informed about what data is collected, how it is used, and which parties it is shared with. Consent mechanisms SHOULD be clear and specific to the requested scope.

Authorization requests SHOULD clearly communicate the level of access being granted, and users SHOULD have the ability to review and revoke previously granted permissions.

7.7 Correlation and Linkability

Even when direct identifiers are not shared, patterns in requests or metadata (such as timestamps, IP addresses, or device characteristics) MAY allow user activity to be correlated.

To mitigate this, implementations SHOULD limit the exposure of metadata where possible and consider techniques such as anonymization or pseudonymization.

7.8 Data Retention

Storing user-related data for longer than necessary increases the risk of unauthorized access and privacy violations.

Implementations SHOULD define clear data retention policies and ensure that data is deleted or anonymized once it is no longer required for operational or legal purposes.

8 IANA Considerations

This document has no IANA actions.

9 References

9.1 Normative References

- RFC 9449 - OAuth 2.0 Demonstrating Proof of Possession (DPoP)
- RFC 7519 - JSON Web Token (JWT)
- RFC 9068 - JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens
- RFC 8725 - JSON Web Token Best Current Practices
- RFC 7638 - JSON Web Key (JWK) Thumbprint

- RFC 7523 - JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants
- CIBA Core 1.0 (OpenID)
- WebAuthn Level 3 (W3C)
- RFC 6749 - The OAuth 2.0 Authorization Framework
- OAuth 2.1 (draft-ietf-oauth-v2-1-15)
- RFC 9207 - OAuth 2.0 Authorization Server Issuer Identification
- draft-ietf-oauth-attestation-based-client-auth
- Google Play Integrity API
- Apple App Attest / DeviceCheck
- RFC 7009 - OAuth 2.0 Token Revocation
- RFC 8707 - Resource Indicators for OAuth 2.0
- RFC 9396 - OAuth 2.0 Rich Authorization Requests
- RFC 7591 - OAuth 2.0 Dynamic Client Registration Protocol
- RFC 8414 - OAuth 2.0 Authorization Server Metadata
- RFC 7662 - OAuth 2.0 Token Introspection

9.2 Informative References

- RFC 9700 - Best Current Practice for OAuth 2.0 Security
- FAPI 2.0 Security Profile

10 Deployment Guidance

This section provides guidance for implementation and deployment of the mAuth protocol.

Clients MUST generate the DPoP client instance key pair described in Step 4 using hardware backed cryptographic storage when available.

11 Extensibility Model

MAuth is designed to be extensible. This section defines the mechanisms by which new functionality can be introduced while maintaining interoperability between implementations.

Extension points include, but are not limited to, request parameters, response parameters, token types, and error codes. Future specifications MAY define additional parameters or values for these fields.

Unless otherwise specified, implementations MUST ignore unknown request or response parameters. Extensions that require mandatory processing MUST explicitly define error handling behavior.

To avoid collisions, extension parameters SHOULD use unique names. Specifications defining new extensions SHOULD clearly document their semantics and processing rules.

Authorization servers MAY support additional token types or grant mechanisms. Clients MUST NOT assume support for any extension unless it is explicitly advertised or documented by the authorization server.

Extensions MUST NOT weaken the security properties of mAuth. Any extension that affects authentication, authorization, or token handling MUST define its security considerations.

B

Tamarin mAuth Code

B.1 CIBA

```
    theory mAuth_CIBA
begin

builtins: signing
heuristic: S

rule ChanOut_S:
    [ Out_S($A, $B, x) ]
    --[ ChanOut_S($A, $B, x) ]->
    [ Sec($A, $B, x) ]

rule ChanIn_S:
    [ Sec($A, $B, x) ]
    --[ ChanIn_S($A, $B, x) ]->
    [ In_S($A, $B, x) ]

//
    ================================================================================
// SETUP RULES
//
    ================================================================================

rule Setup_AS_Identity[role="AS"]:
    [ Fr(~iss) ]
    -->
    [ !AS_Identity(~iss) ]

rule Setup_Resource_Server[role="RS"]:
    [ Fr(~data) ]
    -->
    [ !RS_Resource(~data) ]

rule Setup_Backend_Identity[role="Backend"]:
    [ Fr(~backend_sk) ]
    --[ Secret_BackendKey(~backend_sk) ]->
```

```
    [ !Backend_Ltk(~backend_sk),
      !PkBackend(pk(~backend_sk)),
      !AS_Knows_Backend(pk(~backend_sk)) ]

//
    ================================================================================
// ATTESTATION CHALLENGE (Steps 2-3)
//
    ================================================================================

rule Client_AttestChallReq_ToBackend_Step2[role="Client"]:
    [ Fr(~ClientID) ]
    -->
    [ Out_S($Client, $Backend, <'step2_c2b', ~ClientID>),
    Client_KeygenSlot(~ClientID) ]

rule Backend_AttestChallReq_ToAS_Step2[role="Backend"]:
    [ In_S($Client, $Backend, <'step2_c2b', ClientID>) ]
    -->
    [ Out_S($Backend, $AS, <'step2_b2as', ClientID>) ]

rule AS_IssueChallenge_ToBackend_Step3[role="AS"]:
    [ In_S($Backend, $AS, <'step2_b2as', ClientID>),
      Fr(~nonce),
      !AS_Identity(iss) ]
    --[ AS_Issued_Nonce(~nonce),
        AS_Request_Accepted(<'challenge', ~nonce>) ]->
    [ Out_S($AS, $Backend, <'step3_as2b', ClientID, ~nonce, iss>),
      AS_Pending_Challenge_Nonce_Useable(ClientID, ~nonce, iss) ]

rule Backend_IssueChallenge_ToClient_Step3[role="Backend"]:
    [ In_S($AS, $Backend, <'step3_as2b', ClientID, nonce, iss>) ]
    -->
    [ Out_S($Backend, $Client, <'step3_b2c', ClientID, nonce, iss>) ]

//
    ================================================================================
// KEY GENERATION & ATTESTATION (Steps 4-7)
//
    ================================================================================

rule Client_GenerateKey_ToClient_Step4[role="Client"]:
    [ In_S($Backend, $Client, <'step3_b2c', ClientID, nonce, iss>),
      Fr(~skC) ]
    --[ Secret_ClientKey(ClientID, ~skC),
        Client_Key_In_Hardware(pk(~skC)),
        Client_Learned_Iss(ClientID, iss) ]->
    [ Client_GenerateKey_ToClient_Step4(ClientID, nonce),
      !Client_Key(ClientID, ~skC, pk(~skC)),
```

```
79        !Client_Expected_Iss(ClientID, iss) ]
80
81  rule Client_RequestAttestation_ToAA_Step5[role="Client"]:
82      [ Client_GenerateKey_ToClient_Step4(ClientID, nonce),
83        !Client_Key(ClientID, skC, pkC),
84        Client_KeygenSlot(ClientID),
85        Fr(~Device_Info) ]
86      --[ Client_Sent_Attestation_Request(ClientID, nonce, pkC) ]->
87      [ Client_RequestAttestation_ToAA_Step5(<~Device_Info, nonce, pkC>),
88        Client_AttestRequestPending_ToClient_Step5To8(ClientID, nonce,
    ~Device_Info),
89        Client_ClientIDForKeygen_ToAA(ClientID) ]
90
91  rule Setup_AA_Enclave[role="AA"]:
92      [ Client_ClientIDForKeygen_ToAA(ClientID),
93        Fr(~aa_sk) ]
94      --[ Secret_AAKey(~aa_sk) ]->
95      [ !AA_Ltk(~aa_sk),
96        !PkAA(ClientID, pk(~aa_sk))]
97
98  rule AA_SignAttestation_ToAA_Step6[role="AA"]:
99      [ Client_RequestAttestation_ToAA_Step5(<Device_Info, nonce, pkC>),
100       !AA_Ltk(aa_sk) ]
101     --[ AA_Signed_Attestation(nonce, pkC),
102         AA_NonceUsed(nonce) ]->
103     [ AA_SignAttestation_ToAA_Step6(sign(<Device_Info, nonce, pkC>,
    aa_sk)) ]
104
105 rule AA_SendAttestation_ToClient_Step7[role="AA"]:
106     [ AA_SignAttestation_ToAA_Step6(attestJWT) ]
107     -->
108     [ AA_SendAttestation_ToClient_Step7(attestJWT) ]
109
110 rule Client_ForwardPlatformAttest_ToBackend_Step7b[role="Client"]:
111     [ AA_SendAttestation_ToClient_Step7(platformAttest),
112       Client_AttestRequestPending_ToClient_Step5To8(ClientID, nonce,
    Device_Info),
113       !Client_Key(ClientID, skC, pkC) ]
114     -->
115     [ Out_S($Client, $Backend, <'wrap_cat_c2b', ClientID, nonce, pkC,
    Device_Info, platformAttest>),
116       Client_AwaitingCAT_ToClient(ClientID, nonce, Device_Info,
    platformAttest, pkC) ]
117
118 rule Backend_WrapCAT_ToClient_Step7c[role="Backend"]:
119     [ In_S($Client, $Backend, <'wrap_cat_c2b', ClientID, nonce, pkC,
    Device_Info, platformAttest>),
120       !Backend_Ltk(backend_sk) ]
121     --[ Backend_Issued_CAT(ClientID, pkC, nonce) ]->
```

```
122     [ Out_S($Backend, $Client,
123          <'cat_response_b2c', ClientID,
124            sign(<'client-attestation+jwt', ClientID, pkC, nonce,
    platformAttest>, backend_sk),
125            platformAttest>) ]
126
127
128 //
    ================================================================================
129 // DPoP CREATION & AUTHORIZATION REQUEST (Steps 8-10)
130 //
    ================================================================================
131
132 rule Client_CreateDPoPP_ToClient_Step8[role="Client"]:
133     [ In_S($Backend, $Client, <'cat_response_b2c', ClientID,
    clientAttestJWT, platformAttest>),
134       Client_AwaitingCAT_ToClient(ClientID, nonce, Device_Info,
    platformAttest, pkC),
135       !Client_Key(ClientID, skC, pkC),
136       Fr(~DPoP_Params) ]
137     --[ Client_Created_DPoP(pkC, ~DPoP_Params,'authz') ]->
138     [ Client_DPoPAttestation_ToClient_Step8(ClientID, nonce,
    clientAttestJWT, platformAttest, Device_Info),
139       Client_CreateDPoPP_ToClient_Step8(
140           sign(<~DPoP_Params, nonce>, skC), pkC, ~DPoP_Params, nonce) ]
141
142 rule Client_SendAuthzReq_ToBackend_Step9[role="Client"]:
143     [ Client_DPoPAttestation_ToClient_Step8(ClientID, nonce,
    clientAttestJWT, platformAttest, Device_Info),
144       Client_CreateDPoPP_ToClient_Step8(dpopp, pkC, DPoP_Params, nonce) ]
145     -->
146     [ Out_S($Client, $Backend,
147         <'step9_c2b', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
148          dpopp, pkC, DPoP_Params, pkC>) ]
149
150 rule Backend_SendAuthzReq_ToAS_Step9[role="Backend"]:
151     [ In_S($Client, $Backend,
152         <'step9_c2b', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
153          dpopp, pkC, DPoP_Params, dpop_jkt>) ]
154     -->
155     [ Out_S($Backend, $AS,
156         <'step9_b2as', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
157          dpopp, pkC, DPoP_Params, dpop_jkt>) ]
158
159 rule AS_VerifyJWTAndDPoPP_ToBackend_Step10[role="AS"]:
```

```
160     [ In_S($Backend, $AS, <'step9_b2as', ClientID, nonce, Device_Info,
    platformAttest, clientAttestJWT, dpopp, pkC, DPoP_Params, dpop_jkt>),
161       AS_Pending_Challenge_Nonce_Useable(ClientID, nonce, iss),
162       !PkAA(ClientID, pkAA),
163       !AS_Knows_Backend(pkBackend),
164       !AS_Identity(iss) ]
165     --[ Eq(verify(platformAttest, <Device_Info, nonce, pkC>, pkAA), true),
166         Eq(verify(dpopp, <DPoP_Params,nonce>, pkC), true),
167         Eq(verify(clientAttestJWT, <'client-attestation+jwt', ClientID,
    pkC, nonce, platformAttest>, pkBackend), true),
168         Eq(dpop_jkt, pkC),
169         AS_Request_Accepted(<'authorization', nonce>),
170         AS_Accepted_Client_Key(ClientID, pkC),
171         AS_Verified_CAT(ClientID, pkC),
172         AS_Verified_PlatformAttest(ClientID, pkC),
173         AS_Verified_dpop_jkt(ClientID, pkC),
174         DPoP_Used(DPoP_Params),
175         DPoP_Verified(DPoP_Params, pkC) ]->
176     [ Out_S($AS, $Backend, <'ciba_start_as2b', ClientID, iss>),
177       !AS_Client_Tied_PublicKey(ClientID, pkC, iss) ]
178
179 //
    ================================================================================
180 // CIBA AUTHENTICATION (Step 11)
181 //
    ================================================================================
182
183 rule Backend_CIBAStart_ToClient_CIBAStep1[role="Backend"]:
184     [ In_S($AS, $Backend, <'ciba_start_as2b', ClientID, iss>) ]
185     -->
186     [ Out_S($Backend, $Client, <'ciba_start_b2c', ClientID, iss>) ]
187
188 rule Client_CIBAStart_ToBackend_CIBAStep2[role="Client"]:
189     [ In_S($Backend, $Client, <'ciba_start_b2c', ClientID, iss>),
190       !Client_Expected_Iss(ClientID, iss) ]
191     -->
192     [ Out_S($Client, $Backend, <'ciba_step2_c2b', ClientID>) ]
193
194 rule Backend_AuthReq_ToAS_CIBAStep2[role="Backend"]:
195     [ In_S($Client, $Backend, <'ciba_step2_c2b', ClientID>) ]
196     -->
197     [ Out_S($Backend, $AS, <'ciba_step2_b2as', ClientID>) ]
198
199 rule AS_ACKSend_ToBackend_CIBAStep3[role="AS"]:
200     [ In_S($Backend,$AS, <'ciba_step2_b2as', ClientID>),
201       !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
202       Fr(~auth_req_id),
203       !AS_Identity(iss) ]
204     --[ AS_Request_Accepted(<'ciba_ack', ~auth_req_id>),
```

```
205     ASOnly_AuthReq_ClientID_Action(~auth_req_id, ClientID) ]->
206     [ Out_S($AS, $Backend, <'ciba_ack_as2b', ~auth_req_id, iss>),
207       !AS_UserAuthPending_CIBA(~auth_req_id),
208       !ASOnly_AuthReq_ClientID(~auth_req_id, ClientID),
209       !ASOnly_AuthReq_ClientKey(~auth_req_id, pkC),
210       !ASOnly_AuthReq_Iss(~auth_req_id, iss) ]
211
212 rule Backend_ACKSend_ToClient_CIBAStep3[role="Backend"]:
213     [ In_S($AS,$Backend, <'ciba_ack_as2b', auth_req_id, iss>) ]
214     -->
215     [ Out_S($Backend, $Client, <'ciba_ack_b2c', auth_req_id, iss>) ]
216
217 rule
        Client_UserAuth_ToAuthMethod_PollStart_ToBackend_CIBAStep4[role="Client"]:
218     [ In_S($Backend,$Client, <'ciba_ack_b2c', auth_req_id, iss>),
219       !Client_Expected_Iss(ClientID, iss) ]
220     -->
221     [ Client_UserAuth_ToAuthMethod_CIBAStep4(auth_req_id),
222       Out_S($Client, $Backend, <'ciba_poll_c2b', auth_req_id, ClientID>) ]
223
224 rule AuthMethod_UserAuth_ToAS_CIBAStep4[role="UserAuthMethod"]:
225     [ Client_UserAuth_ToAuthMethod_CIBAStep4(auth_req_id) ]
226     --[ UserAuth_Initiated(auth_req_id) ]->
227     [ AuthMethod_UserAuth_ToAS_CIBAStep4(auth_req_id) ]
228
229 rule AS_UserAuthAccepted_ToAS_CIBAStep4[role="AS"]:
230     [ AuthMethod_UserAuth_ToAS_CIBAStep4(auth_req_id),
231       !AS_UserAuthPending_CIBA(auth_req_id),
232       !ASOnly_AuthReq_ClientID(auth_req_id, ClientID),
233       !ASOnly_AuthReq_ClientKey(auth_req_id, pkC),
234       !ASOnly_AuthReq_Iss(auth_req_id, iss),
235       Fr(~authorization_code) ]
236     --[ AS_UserAuthAccepted(auth_req_id),
237         CIBA_AuthEvent_Bound(auth_req_id, ~authorization_code),
238         AS_Request_Accepted(<'ciba_user_auth', auth_req_id>),
239         AS_Auth_Completed(ClientID),
240         AuthzCode_Issued(~authorization_code, ClientID, pkC, iss) ]->
241     [ AS_UserAuthAccepted_ToAS_CIBAStep4(~authorization_code,
        auth_req_id),
242       AuthzCode(~authorization_code, ClientID, pkC, iss) ]
243
244 rule Backend_Poll_ToAS_Step5[role="Backend"]:
245     [ In_S($Client, $Backend, <'ciba_poll_c2b', auth_req_id, ClientID>) ]
246     --[ CIBA_Poll_Sent(auth_req_id) ]->
247     [ Out_S($Backend, $AS, <'ciba_poll_b2as', auth_req_id>) ]
248
249 rule AS_PollAnsPending_ToBackend_CIBAStep5[role="AS"]:
250     [ In_S($Backend, $AS, <'ciba_poll_b2as', auth_req_id>),
251       !AS_UserAuthPending_CIBA(auth_req_id),
```

```
252       !ASOnly_AuthReq_ClientID(auth_req_id, ClientID) ]
253     --[ AS_Request_Accepted(<'poll_pending', auth_req_id>) ]->
254     []
255
256 rule AS_PollAnsAuthorized_ToBackend_CIBAStep5[role="AS"]:
257     [ In_S($Backend, $AS, <'ciba_poll_b2as', auth_req_id>),
258       !AS_UserAuthPending_CIBA(auth_req_id),
259       AS_UserAuthAccepted_ToAS_CIBAStep4(authorization_code, auth_req_id),
260       !ASOnly_AuthReq_ClientID(auth_req_id, ClientID) ]
261     --[ AS_Request_Accepted(<'poll_authorized', auth_req_id>),
262         CIBA_Code_Issued(auth_req_id) ]->
263     [ Out_S($AS, $Backend, <'auth_completed_as2b', ClientID,
    authorization_code>) ]
264
265 //
    ================================================================================
266 // TOKEN EXCHANGE & RESOURCE ACCESS (Steps 12-19)
267 //
    ================================================================================
268
269
270 rule Backend_RequestDPoP_ForTokenExchange[role="Backend"]:
271     [ In_S($AS, $Backend, <'auth_completed_as2b', ClientID,
    authorization_code>) ]
272     -->
273     [ Out_S($Backend, $Client, <'dpop_needed_for_token_b2c', ClientID>),
274       Backend_PendingTokenExchange(ClientID, authorization_code) ]
275
276 rule Client_CreateDPoP_ForTokenExchange[role="Client"]:
277     [ In_S($Backend, $Client, <'dpop_needed_for_token_b2c', ClientID>),
278       !Client_Key(ClientID, skC, pkC),
279       Fr(~DPoP_Params_TE) ]
280     --[ Client_Created_DPoP(pkC, ~DPoP_Params_TE,'token_te') ]->
281     [ Out_S($Client, $Backend, <'dpop_for_token_c2b',
282             sign(<~DPoP_Params_TE,'token_endpoint'>, skC), pkC,
    ~DPoP_Params_TE, ClientID>) ]
283
284 rule Backend_TokenRequest_ToAS_Step12[role="Backend"]:
285     [ In_S($Client, $Backend, <'dpop_for_token_c2b', dpopp_te, pkC,
    DPoP_Params_TE, ClientID>),
286       Backend_PendingTokenExchange(ClientID, authorization_code) ]
287     -->
288     [ Out_S($Backend, $AS, <'token_req_b2as', ClientID,
    authorization_code,
289             dpopp_te, pkC, DPoP_Params_TE>) ]
290
291 rule AS_TokenAns_ToBackend_Step13[role="AS"]:
292     [ In_S($Backend, $AS, <'token_req_b2as', ClientID, authorization_code,
293             dpopp_te, pkC, DPoP_Params_TE>),
```

```
294       AuthzCode(authorization_code, ClientID, pkC, iss),
295       !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
296       Fr(~constrained_access_token),
297       Fr(~DPoP_reference_token),
298       Fr(~refresh_token),
299       !AS_Identity(iss) ]
300     --[ Eq(verify(dpopp_te, <DPoP_Params_TE,'token_endpoint'>, pkC),
    true),
301         DPoP_Used(DPoP_Params_TE),
302         DPoP_Verified(DPoP_Params_TE, pkC),
303         AS_Request_Accepted(<'token_exchange', authorization_code>),
304         AuthzCode_Redeemed(authorization_code, ClientID, pkC),
305         Token_Issued(~constrained_access_token, ClientID, pkC),
306         RefToken_Issued(~DPoP_reference_token, ClientID, pkC),
307         RefreshToken_Issued(~refresh_token, ClientID, pkC),
308         Token_Issued_By_Iss(ClientID, iss) ]->
309     [ !ConstrainedAccess(~constrained_access_token, pkC),
310       !ConstrainedReference(~DPoP_reference_token, pkC, iss),
311       !ConstrainedRefresh(~refresh_token, pkC),
312       Out_S($AS, $Backend, <'token_resp_as2b', ClientID,
    ~DPoP_reference_token,
313              ~constrained_access_token, pkC, iss, ~refresh_token>) ]
314
315 rule Backend_SendReferenceToken_ToClient_Step14[role="Backend"]:
316     [ In_S($AS, $Backend, <'token_resp_as2b', ClientID,
    DPoP_reference_token,
317             constrained_access_token, pkC, iss, refresh_token>) ]
318     -->
319     [ Out_S($Backend, $Client, <'ref_token_b2c', ClientID,
    DPoP_reference_token, iss>),
320       Backend_AccessTokenHeld(DPoP_reference_token,
    constrained_access_token, pkC, ClientID),
321       Backend_RefreshTokenHeld(refresh_token, pkC, ClientID) ]
322
323 rule Client_GenerateDPoP_Step15_SendDPoP_ToBackend_Step16[role="Client"]:
324     [ In_S($Backend, $Client, <'ref_token_b2c', ClientID,
    DPoP_reference_token, iss>),
325       !Client_Key(ClientID, skC, pkC),
326       !Client_Expected_Iss(ClientID, iss),
327       Fr(~DPoP_Params2) ]
328     --[ Client_Verified_Iss(ClientID, iss),
329         Client_Created_DPoP(pkC, ~DPoP_Params2,'resource') ]->
330     [ Out_S($Client, $Backend, <'dpop_resource_c2b',
331             sign(<~DPoP_Params2, DPoP_reference_token>, skC),
332             pkC, ~DPoP_Params2, DPoP_reference_token>),
333       Client_CanRefresh(ClientID) ]
334
335 rule Backend_ResourceRequest_ToRS_Step17[role="Backend"]:
```

```
336     [ In_S($Client,$Backend, <'dpop_resource_c2b', dpopp, pkC,
    DPoP_Params2, DPoP_reference_token>),
337       Backend_AccessTokenHeld(DPoP_reference_token,
    constrained_access_token, pkC, ClientID) ]
338     -->
339     [ Out_S($Backend, $RS, <'rs_verify_b2rs', constrained_access_token,
    dpopp, pkC,
340             DPoP_Params2, DPoP_reference_token, ClientID>) ]
341
342 rule RS_ForwardIntrospection_ToAS[role="RS API"]:
343     [ In_S($Backend, $RS, <'rs_verify_b2rs', constrained_access_token,
    dpopp, pkC,
344            DPoP_Params2, DPoP_reference_token, ClientID>)]
345     -->
346     [ Out_S($RS, $AS, <'introspect', constrained_access_token, dpopp, pkC,
347             DPoP_Params2, DPoP_reference_token, ClientID>) ,
348             RS_Client_Tied_Access(ClientID,constrained_access_token)]
349
350 rule AS_VerifyDPoPPAndTokens_Step18[role="AS"]:
351     [ In_S($RS, $AS, <'introspect', constrained_access_token, dpopp, pkC,
352            DPoP_Params2, DPoP_reference_token, ClientID>),
353       !ConstrainedAccess(constrained_access_token, pkC),
354       !ConstrainedReference(DPoP_reference_token, pkC, iss),
355       !AS_Client_Tied_PublicKey(ClientID, pkC, iss) ,
356       !AS_Identity(iss) ]
357     --[ Eq(verify(dpopp, <DPoP_Params2, DPoP_reference_token>, pkC),
    true),
358         DPoP_Used(DPoP_Params2),
359         DPoP_Verified(DPoP_Params2, pkC),
360         AS_Request_Accepted(<'resource_access', DPoP_Params2>),
361         Token_Used_For_Access(constrained_access_token, ClientID, pkC) ]->
362     [ Out_S($AS, $RS, <'verified', ClientID, constrained_access_token>) ]
363
364 rule RS_SendResource_Step19[role="RS API"]:
365     [ In_S($AS, $RS, <'verified', ClientID, constrained_access_token>),
366       !RS_Resource(data),
367       RS_Client_Tied_Access(ClientID, constrained_access_token) ]
368     --[ RS_Sent_Resource(data, ClientID, constrained_access_token),
369         RS_Access_Granted(data, ClientID) ]->
370     [ Out_S($RS, $Client, <'resource_sent', data, ClientID>) ]
371
372 rule Client_Received_Data[role="Client"]:
373    [In_S($RS,$Client,<'resource_sent',data,ClientID>)]
374    --[Client_Recourse_Received(data,ClientID)]->
375    []
376
377
378 //
    ================================================================================
```

```
// REFRESH TOKEN EXCHANGE
//
    ================================================================================

rule Client_RequestRefresh[role="Client"]:
    [ Client_CanRefresh(ClientID),
      !Client_Key(ClientID, skC, pkC),
      Fr(~DPoP_Params_Refresh) ]
    --[ Client_Created_DPoP(pkC, ~DPoP_Params_Refresh,'refresh'),
        Client_Requested_Refresh(ClientID) ]->
    [ Out_S($Client, $Backend, <'refresh_req_c2b',
            sign(<~DPoP_Params_Refresh, 'refresh'>, skC),
            pkC, ~DPoP_Params_Refresh, ClientID>) ]

rule Backend_RefreshRequest_ToAS[role="Backend"]:
    [ In_S($Client, $Backend, <'refresh_req_c2b', dpopp, pkC,
    DPoP_Params_R, ClientID>),
      Backend_RefreshTokenHeld(refresh_token, pkC, ClientID) ]
    -->
    [ Out_S($Backend, $AS, <'refresh_req_b2as', refresh_token, dpopp,
    pkC, DPoP_Params_R, ClientID>) ]

rule AS_VerifyRefresh_IssueToken[role="AS"]:
    [ In_S($Backend, $AS, <'refresh_req_b2as', refresh_token, dpopp, pkC,
    DPoP_Params_R, ClientID>),
      !ConstrainedRefresh(refresh_token, pkC),
      !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
      !AS_Identity(iss)  ]
    --[ Eq(verify(dpopp, <DPoP_Params_R,'refresh'>, pkC), true),
        DPoP_Used(DPoP_Params_R),
        DPoP_Verified(DPoP_Params_R, pkC),
        AS_Request_Accepted(<'refresh', DPoP_Params_R>),
        RefreshToken_Used(refresh_token, ClientID, pkC) ]->
    [ AS_RefreshCompleted(ClientID) ]

//
    ================================================================================
// LEAK / ATTACKER EVENTS
//
    ================================================================================

rule Leak_Access_Token:
    [ !ConstrainedAccess(token, pkC) ]
    --[ LeakToken(token, pkC) ]->
    [ Out(token) ]

rule Leak_Reference_Token:
    [ !ConstrainedReference(ref_token, pkC, iss) ]
    --[ LeakRefToken(ref_token, pkC) ]->
```

```
422     [ Out(ref_token) ]
423
424 rule Leak_Refresh_Token:
425     [ !ConstrainedRefresh(refresh_token, pkC) ]
426     --[ LeakRefreshToken(refresh_token, pkC) ]->
427     [ Out(refresh_token) ]
428
429 rule Reveal_AA_Key:
430     [ !AA_Ltk(aa_sk) ]
431     --[ RevealAA(aa_sk) ]->
432     [ Out(aa_sk) ]
433
434 rule Reveal_Client_Key:
435     [ !Client_Key(ClientID, skC, pkC) ]
436     --[ RevealClientKey(ClientID, skC) ]->
437     [ Out(skC) ]
438
439 rule Reveal_Backend_Key:
440     [ !Backend_Ltk(backend_sk) ]
441     --[ RevealBackend(backend_sk) ]->
442     [ Out(backend_sk) ]
443
444 //
    ================================================================================
445 // RESTRICTIONS
446 //
    ================================================================================
447
448 restriction Equality:
449     "All x y #i. Eq(x,y) @ #i ==> x = y"
450
451 restriction unique_dpopp:
452     "All params #i #j.
453        DPoP_Used(params) @ #i & DPoP_Used(params) @ #j
454        ==> #i = #j"
455
456 restriction ciba_single_code_issuance:
457     "All id #i #j.
458        CIBA_Code_Issued(id) @ #i & CIBA_Code_Issued(id) @ #j
459        ==> #i = #j"
460
461 restriction no_replay:
462     "All id #i #j.
463        AS_Request_Accepted(id) @ #i & AS_Request_Accepted(id) @ #j
464        ==> #i = #j"
465
466 restriction unique_attestation_per_nonce:
467     "All nonce pkC1 pkC2 #i #j.
468        AA_Signed_Attestation(nonce, pkC1) @ #i
```

```
469         & AA_Signed_Attestation(nonce, pkC2) @ #j
470         ==> #i = #j"
471
472
473 lemma typing [sources]:
474   " (All t cid pk #i #j.
475         Token_Issued(t, cid, pk) @ #j & !KU(t) @ #i
476         ==> (Ex #l. LeakToken(t, pk) @ #l & #l < #i))
477   & (All t cid pk #i #j.
478         RefToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
479         ==> (Ex #l. LeakRefToken(t, pk) @ #l & #l < #i))
480   & (All t cid pk #i #j.
481         RefreshToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
482         ==> (Ex #l. LeakRefreshToken(t, pk) @ #l & #l < #i))
483   & (All c cid pk iss #i #j.
484         AuthzCode_Issued(c, cid, pk, iss) @ #j & !KU(c) @ #i
485         ==> F)
486   "
487
488 lemma token_implies_code [reuse]:
489   "All t cid pk #i. Token_Issued(t, cid, pk) @ #i
490    ==> (Ex c iss #j. AuthzCode_Issued(c, cid, pk, iss) @ #j & #j < #i)"
491
492 lemma code_implies_accept [reuse]:
493   "All c cid pk iss #i. AuthzCode_Issued(c, cid, pk, iss) @ #i
494    ==> (Ex #j. AS_Accepted_Client_Key(cid, pk) @ #j & #j < #i)"
495
496 lemma access_implies_issued [reuse, use_induction]:
497   "All t cid pk #i. Token_Used_For_Access(t, cid, pk) @ #i
498    ==> (Ex #j. Token_Issued(t, cid, pk) @ #j & #j < #i)"
499
500 lemma refresh_used_implies_issued [reuse]:
501   "All t cid pk #i. RefreshToken_Used(t, cid, pk) @ #i
502    ==> (Ex #j. RefreshToken_Issued(t, cid, pk) @ #j & #j < #i)"
503
504 lemma accept_unique_pkC [reuse, use_induction]:
505   "All cid pk1 pk2 #i #j.
506       AS_Accepted_Client_Key(cid, pk1) @ #i
507     & AS_Accepted_Client_Key(cid, pk2) @ #j
508     ==> pk1 = pk2"
509
510 lemma client_iss_unique [reuse, use_induction]:
511   "All cid iss1 iss2 #i #j.
512       Client_Learned_Iss(cid, iss1) @ #i
513     & Client_Learned_Iss(cid, iss2) @ #j
514     ==> iss1 = iss2 & #i = #j"
515
516 //
    ================================================================================
```

```
517 // UNIQUE PROOFS
518 //
        ==========================================================================
519
520
521 lemma L_unique_dpopp_proof:
522     all-traces
523     "All params #i #j.
524        DPoP_Used(params) @ #i & DPoP_Used(params) @ #j
525        ==> #i = #j"
526
527 lemma L_unique_attestation_proof:
528     all-traces
529     "All nonce pkC1 pkC2 #i #j.
530        AA_Signed_Attestation(nonce, pkC1) @ #i
531        & AA_Signed_Attestation(nonce, pkC2) @ #j
532        ==> #i = #j & pkC1 = pkC2"
533
534 //
        ==========================================================================
535 // SANITY
536 //
        ==========================================================================
537
538
539
540 lemma SANITY_keygen[hide_lemma=R2_dpop_requires_creation,
541    hide_lemma=B1_attestation_unforgeable,
542    hide_lemma=B2_as_requires_attestation,
543    hide_lemma=B3_token_implies_attestation,
544    hide_lemma=I1_token_requires_hardware_key,
545    hide_lemma=accept_unique_pkC,
546    hide_lemma=client_iss_unique,
547    hide_lemma=token_implies_code,
548    hide_lemma=code_implies_accept,
549    hide_lemma=access_implies_issued,
550    hide_lemma=refresh_used_implies_issued]: exists-trace
551   "Ex cid pk #t. Client_Learned_Iss(cid, pk) @ #t"
552
553 lemma SANITY_token_issued[hide_lemma=R2_dpop_requires_creation,
554    hide_lemma=B1_attestation_unforgeable,
555    hide_lemma=B2_as_requires_attestation,
556    hide_lemma=B3_token_implies_attestation,
557    hide_lemma=I1_token_requires_hardware_key,
558    hide_lemma=accept_unique_pkC,
559    hide_lemma=client_iss_unique,
560    hide_lemma=token_implies_code,
561    hide_lemma=code_implies_accept,
562    hide_lemma=access_implies_issued,
```

```
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex t cid pk #i. Token_Issued(t, cid, pk) @ #i"

lemma SANITY_dpop_verified[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
    hide_lemma=B3_token_implies_attestation,
    hide_lemma=I1_token_requires_hardware_key,
    hide_lemma=accept_unique_pkC,
    hide_lemma=client_iss_unique,
    hide_lemma=token_implies_code,
    hide_lemma=code_implies_accept,
    hide_lemma=access_implies_issued,
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex p pk #i. DPoP_Verified(p, pk) @ #i"

lemma SANITY_token_used[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
    hide_lemma=B3_token_implies_attestation,
    hide_lemma=I1_token_requires_hardware_key,
    hide_lemma=accept_unique_pkC,
    hide_lemma=client_iss_unique,
    hide_lemma=token_implies_code,
    hide_lemma=code_implies_accept,
    hide_lemma=access_implies_issued,
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex t cid pk #i. Token_Used_For_Access(t, cid, pk) @ #i"

lemma SANITY_refresh_used[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
    hide_lemma=B3_token_implies_attestation,
    hide_lemma=I1_token_requires_hardware_key,
    hide_lemma=accept_unique_pkC,
    hide_lemma=client_iss_unique,
    hide_lemma=token_implies_code,
    hide_lemma=code_implies_accept,
    hide_lemma=access_implies_issued,
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex t cid pk #i. RefreshToken_Used(t, cid, pk) @ #i"

lemma SANITY_resource_received[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
    hide_lemma=B3_token_implies_attestation,
    hide_lemma=I1_token_requires_hardware_key,
    hide_lemma=accept_unique_pkC,
    hide_lemma=client_iss_unique,
```

```
612     hide_lemma=token_implies_code,
613     hide_lemma=code_implies_accept,
614     hide_lemma=access_implies_issued,
615     hide_lemma=refresh_used_implies_issued]: exists-trace
616   "Ex data cid #t. Client_Recourse_Received(data, cid) @ #t"
617
618
619 lemma SANITY_full_chain[hide_lemma=R2_dpop_requires_creation,
620     hide_lemma=B1_attestation_unforgeable,
621     hide_lemma=B2_as_requires_attestation,
622     hide_lemma=B3_token_implies_attestation,
623     hide_lemma=I1_token_requires_hardware_key,
624     hide_lemma=accept_unique_pkC,
625     hide_lemma=client_iss_unique,
626     hide_lemma=token_implies_code,
627     hide_lemma=code_implies_accept,
628     hide_lemma=access_implies_issued,
629     hide_lemma=refresh_used_implies_issued]:
630      exists-trace
631      "Ex data ClientID nonce pkC code token auth_req_id iss
632          #t1 #t2 #t3 #t4 #t5 #t6 #t7 #t8 #t9 #t10 #t11.
633         AS_Issued_Nonce(nonce)                                  @ #t1
634       & Client_Sent_Attestation_Request(ClientID, nonce, pkC) @ #t2
635       & AA_Signed_Attestation(nonce, pkC)                       @ #t3
636       & Backend_Issued_CAT(ClientID, pkC, nonce)              @ #t4
637       & AS_Accepted_Client_Key(ClientID, pkC)                 @ #t5
638       & UserAuth_Initiated(auth_req_id)                       @ #t6
639       & AS_UserAuthAccepted(auth_req_id)                      @ #t7
640       & AuthzCode_Issued(code, ClientID, pkC, iss)            @ #t7
641       & CIBA_Code_Issued(auth_req_id)                         @ #t8
642       & Token_Issued(token, ClientID, pkC)                    @ #t9
643       & Token_Used_For_Access(token, ClientID, pkC)           @ #t10
644       & Client_Recourse_Received(data, ClientID)              @ #t11
645       & #t1 < #t2 & #t2 < #t3 & #t3 < #t4 & #t4 < #t5
646       & #t5 < #t6 & #t6 < #t7 & #t7 < #t8 & #t8 < #t9
647       & #t9 < #t10 & #t10 < #t11"
648
649 //
      ================================================================================
650 // S: SECRET KEY SECRECY
651 //
      ================================================================================
652
653 lemma S1_client_key_secrecy:
654     all-traces
655     "All ClientID k #i. Secret_ClientKey(ClientID, k) @ #i
656       ==> not (Ex #j. K(k) @ #j)
657           | (Ex #r. RevealClientKey(ClientID, k) @ #r)"
658
```

```
659 lemma S2_access_token_secrecy:
660     all-traces
661     "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
662      ==> not (Ex #j. K(token) @ #j)
663          | (Ex #l. LeakToken(token, pkC) @ #l)"
664
665 lemma S3_reference_token_secrecy:
666     all-traces
667     "All token ClientID pkC #i. RefToken_Issued(token, ClientID, pkC) @ #i
668      ==> not (Ex #j. K(token) @ #j)
669          | (Ex #l. LeakRefToken(token, pkC) @ #l)"
670
671 lemma S4_aa_key_secrecy:
672     all-traces
673     "All k #i. Secret_AAKey(k) @ #i
674      ==> not (Ex #j. K(k) @ #j)
675          | (Ex #r. RevealAA(k) @ #r)"
676
677 lemma S5_refresh_token_secrecy:
678     all-traces
679     "All token ClientID pkC #i. RefreshToken_Issued(token, ClientID, pkC)
    @ #i
680      ==> not (Ex #j. K(token) @ #j)
681          | (Ex #l. LeakRefreshToken(token, pkC) @ #l)"
682
683 lemma S6_backend_key_secrecy:
684     all-traces
685     "All k #i. Secret_BackendKey(k) @ #i
686      ==> not (Ex #j. K(k) @ #j)
687          | (Ex #r. RevealBackend(k) @ #r)"
688
689 //
    ================================================================================
690 // Z: AUTHORIZATION CORRECTNESS
691 //
    ================================================================================
692
693
694 lemma Z1_code_after_attestation:
695     all-traces
696     "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
    iss) @ #i
697      ==> (Ex nonce #j #k.
698        AS_Issued_Nonce(nonce) @ #j
699        & AA_Signed_Attestation(nonce, pkC) @ #k & #j < #k & #k < #i)
700          | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
701
702 lemma Z2_no_double_redeem:
703     all-traces
```

```
    "not (Ex code ClientID1 ClientID2 pkC1 pkC2 #i #j.
            AuthzCode_Redeemed(code, ClientID1, pkC1) @ #i
            & AuthzCode_Redeemed(code, ClientID2, pkC2) @ #j
            & not (#i = #j))"

lemma Z3_no_auth_code_substitution:
    all-traces
    "All code ClientID1 ClientID2 pkC1 pkC2 iss #i #j.
       AuthzCode_Issued(code, ClientID1, pkC1, iss) @ #i
       & AuthzCode_Redeemed(code, ClientID2, pkC2) @ #j
       ==> ClientID1 = ClientID2 & pkC1 = pkC2"


lemma Z4_code_unique_issuance:
    all-traces
    "All code ClientID1 ClientID2 pkC1 pkC2 iss1 iss2 #i #j.
       AuthzCode_Issued(code, ClientID1, pkC1, iss1) @ #i
       & AuthzCode_Issued(code, ClientID2, pkC2, iss2) @ #j
       ==> #i = #j & ClientID1 = ClientID2 & pkC1 = pkC2 & iss1 = iss2"

//
    ================================================================================
// T: TOKEN BINDING / PROOF-OF-POSSESSION
//
    ================================================================================

lemma T1_stolen_token_unusable:
    all-traces
    "All token pkC ClientID #i #j.
       LeakToken(token, pkC) @ #i
       & Token_Used_For_Access(token, ClientID, pkC) @ #j
       ==> (Ex params ctx #k #l.
              Client_Created_DPoP(pkC, params,ctx) @ #k
              & DPoP_Verified(params, pkC) @ #l
              & #k < #l & #l < #j)
           | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

lemma T2_token_access_requires_pop:
    all-traces
    "All token ClientID pkC #i. Token_Used_For_Access(token, ClientID,
    pkC) @ #i
     ==> (Ex params #j #k.
            DPoP_Verified(params, pkC) @ #j
            & AS_Accepted_Client_Key(ClientID, pkC) @ #k
            & (#k < #j | #k = #j)
            &  #j < #i)
         | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

lemma T3_no_token_identity_crossing:
```

```
    all-traces
    "All token ClientID1 ClientID2 pkC1 pkC2 #i #j.
       Token_Issued(token, ClientID1, pkC1) @ #i
       & Token_Used_For_Access(token, ClientID2, pkC2) @ #j
       ==> ClientID1 = ClientID2 & pkC1 = pkC2"

lemma T4_stolen_refresh_token_unusable:
    all-traces
    "All token pkC ClientID #i #j.
       LeakRefreshToken(token, pkC) @ #i
       & RefreshToken_Used(token, ClientID, pkC) @ #j
       ==> (Ex params ctx #k. Client_Created_DPoP(pkC, params,ctx) @ #k &
    #k < #j)
           | (Ex skC #k. RevealClientKey(ClientID, skC) @ #k)"

lemma T5_stolen_refresh_needs_dpop:
    all-traces
    "All token pkC ClientID #i #j.
       LeakRefreshToken(token, pkC) @ #i
       & RefreshToken_Used(token, ClientID, pkC) @ #j
       ==> (Ex params ctx #k #l.
              Client_Created_DPoP(pkC, params, ctx) @ #k
              & DPoP_Verified(params, pkC) @ #l
              & #k < #l & #l = #j)
           | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"


//
    ================================================================================
// R: REPLAY RESISTANCE AND FRESHNESS
//
    ================================================================================

lemma R1_dpop_requires_creation:
    all-traces
    "All params pkC #i. DPoP_Verified(params, pkC) @ #i
     ==> (Ex ctx #j. Client_Created_DPoP(pkC, params, ctx) @ #j & #j < #i)
         | (Ex ClientID skC #r. RevealClientKey(ClientID, skC) @ #r)"

//
    ================================================================================
// B: ATTESTATION BINDING
//
    ================================================================================

lemma B1_attestation_unforgeable[reuse]:
    all-traces
    "All nonce pkC #i. AA_Signed_Attestation(nonce, pkC) @ #i
```

```
793      ==> (Ex ClientID #j. Client_Sent_Attestation_Request(ClientID,
    nonce, pkC) @ #j & #j < #i)
794          | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
795
796 lemma B2_as_requires_attestation[reuse]:
797     all-traces
798     "All ClientID pkC #i. AS_Accepted_Client_Key(ClientID, pkC) @ #i
799      ==> (Ex nonce #j. AA_Signed_Attestation(nonce, pkC) @ #j & #j < #i)
800          | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
801
802 lemma B3_token_implies_attestation[reuse]:
803     all-traces
804     "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
805      ==> (Ex nonce #j #k.
806        AA_Signed_Attestation(nonce, pkC) @ #j
807        & AS_Accepted_Client_Key(ClientID, pkC) @ #k
808        & #j < #k & #k < #i)
809         | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
810
811
812 lemma B4_attestation_key_binding:
813     all-traces
814     "All ClientID1 ClientID2 pkC1 pkC2 nonce #i #j #k1 #k2.
815        AS_Accepted_Client_Key(ClientID1, pkC1) @ #i
816        & AS_Accepted_Client_Key(ClientID2, pkC2) @ #j
817        & AA_Signed_Attestation(nonce, pkC1) @ #k1
818        & AA_Signed_Attestation(nonce, pkC2) @ #k2
819        ==> pkC1 = pkC2"
820
821 lemma B5_attestation_nonce_single_use:
822     all-traces
823     "All nonce #i #j.
824        AA_NonceUsed(nonce) @ #i & AA_NonceUsed(nonce) @ #j
825        ==> #i = #j"
826
827
828 lemma B6_cat_unforgeable:
829     all-traces
830     "All ClientID pkC #i.
831        AS_Verified_CAT(ClientID, pkC) @ #i
832        ==> (Ex nonce #j. Backend_Issued_CAT(ClientID, pkC, nonce) @ #j &
    #j < #i)
833            | (Ex backend_sk #r. RevealBackend(backend_sk) @ #r)"
834
835 lemma B7_dpop_jkt_matches_instance_key:
836     all-traces
837     "All ClientID pkC #i.
838        AS_Verified_dpop_jkt(ClientID, pkC) @ #i
839        ==> (Ex nonce #j.
```

```
840                 Client_Sent_Attestation_Request(ClientID, nonce, pkC) @ #j
841                 & #j < #i)
842             | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
843
844 //
    ==========================================================================
845 // K: KEY CONTINUITY
846 //
    ==========================================================================
847
848 lemma K1_key_continuity_attestation_to_token:
849     all-traces
850     "All ClientID pkC_accept token pkC_token #i #j.
851        AS_Accepted_Client_Key(ClientID, pkC_accept) @ #i
852        & Token_Issued(token, ClientID, pkC_token) @ #j
853        & #i < #j
854        ==> pkC_accept = pkC_token
855            | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
856
857 lemma K2_key_continuity_token_to_access:
858     all-traces
859     "All token ClientID pkC1 pkC2 #i #j.
860        Token_Issued(token, ClientID, pkC1) @ #i
861        & Token_Used_For_Access(token, ClientID, pkC2) @ #j
862        ==> pkC1 = pkC2"
863
864
865 lemma K3_dpop_key_matches_accepted:
866     all-traces
867     "All params pkC_dpop ClientID pkC_accept token #i #j #k.
868        DPoP_Verified(params, pkC_dpop) @ #i
869        & AS_Accepted_Client_Key(ClientID, pkC_accept) @ #j
870        & Token_Used_For_Access(token, ClientID, pkC_dpop) @ #k
871        & #j < #i & #i = #k
872        ==> pkC_dpop = pkC_accept
873            | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
874
875
876 //
    ==========================================================================
877 //  I: HARDWARE BOUND KEY CONTINUITY
878 //
    ==========================================================================
879
880 lemma I1_token_requires_hardware_key [reuse]:
881   all-traces
882   "All token ClientID pkC #i.
883     Token_Issued(token, ClientID, pkC) @ #i
884     ==>
```

```
      (Ex #j #k.
         Client_Key_In_Hardware(pkC) @ #j
         & AS_Accepted_Client_Key(ClientID, pkC) @ #k
         & #j < #k & #k < #i)
      | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)
      | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"

lemma I2_access_requires_hardware_key:
    all-traces
    "All token ClientID pkC #i #j.
       Token_Used_For_Access( token, ClientID, pkC ) @ #i
       & Token_Issued( token, ClientID, pkC ) @ #j
       & #j < #i
       ==> (Ex #k #l.
              Client_Key_In_Hardware( pkC ) @ #k
              & AS_Accepted_Client_Key( ClientID, pkC ) @ #l
              & #k < #l & #l < #j)
           | (Ex skC #r. RevealClientKey( ClientID, skC ) @ #r)
           | (Ex aa_sk #r. RevealAA( aa_sk ) @ #r)"

//
    ================================================================================
// E: END-TO-END
//
    ================================================================================

lemma E1_token_requires_authentication:
    all-traces
    "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
     ==> (Ex #j. AS_Auth_Completed(ClientID) @ #j & #j < #i)"

lemma E2_resource_matches_issued_token:
    all-traces
    "All data ClientID #i.
       Client_Recourse_Received(data, ClientID) @ #i
       ==> (Ex token pkC #j #k.
              RS_Sent_Resource(data, ClientID, token) @ #j
              & Token_Issued(token, ClientID, pkC) @ #k
              & #k < #j & #j < #i)"


lemma
    E3_resource_requires_fresh_dpop[hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All data ClientID #i. Client_Recourse_Received(data, ClientID) @ #i
     ==> (Ex pkC ctx params #j #k.
```

```
            Client_Created_DPoP(pkC, params, ctx) @ #j
            & DPoP_Verified(params, pkC) @ #k
            & #j < #k & #k < #i
            & (Ex #l. AS_Accepted_Client_Key(ClientID, pkC) @ #l))
         | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

lemma E4_refresh_requires_issuance:
    all-traces
    "All token ClientID pkC #i.
       RefreshToken_Used(token, ClientID, pkC) @ #i
       ==> (Ex #j.
              RefreshToken_Issued(token, ClientID, pkC) @ #j
              & #j < #i)"


//
    ================================================================================
// ISS: MIX-UP PREVENTION
//
    ================================================================================

lemma ISS1_consistency[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All ClientID iss #i.
       Client_Verified_Iss(ClientID, iss) @ #i
       ==> (Ex #j. Client_Learned_Iss(ClientID, iss) @ #j & #j < #i)"


lemma ISS2_no_mix_up[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All code ClientID pkC iss1 iss2 #i #j.
       AuthzCode_Issued(code, ClientID, pkC, iss1) @ #i
       & AuthzCode_Redeemed(code, ClientID, pkC) @ #j
       & Token_Issued_By_Iss(ClientID, iss2) @ #j
       ==> iss1 = iss2"


lemma ISS3_token_origin[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
```

```
978     hide_lemma=I1_token_requires_hardware_key]:
979      all-traces
980      "All ClientID iss #i. Token_Issued_By_Iss(ClientID, iss) @ #i
981       ==> (Ex code pkC #j. AuthzCode_Issued(code, ClientID, pkC, iss) @ #j
        & #j < #i)"
982
983 lemma ISS4_client_rejects_wrong_iss[hide_lemma=R2_dpop_requires_creation,
984     hide_lemma=B1_attestation_unforgeable,
985     hide_lemma=B2_as_requires_attestation,
986     hide_lemma=B3_token_implies_attestation,
987     hide_lemma=I1_token_requires_hardware_key]:
988      all-traces
989      "All ClientID iss_learned iss_verified #i #j.
990         Client_Learned_Iss(ClientID, iss_learned) @ #i
991         & Client_Verified_Iss(ClientID, iss_verified) @ #j
992         & #i < #j
993         ==> iss_learned = iss_verified"
994
995 lemma ISS5_refresh_preserves_iss[hide_lemma=R2_dpop_requires_creation,
996     hide_lemma=B1_attestation_unforgeable,
997     hide_lemma=B2_as_requires_attestation,
998     hide_lemma=B3_token_implies_attestation,
999     hide_lemma=I1_token_requires_hardware_key]:
1000     all-traces
1001     "All ClientID iss_orig token pkC #i #j.
1002        Token_Issued_By_Iss(ClientID, iss_orig) @ #i
1003        & RefreshToken_Used(token, ClientID, pkC) @ #j
1004        & #i < #j
1005        ==> (Ex code #k.
1006               AuthzCode_Issued(code, ClientID, pkC, iss_orig) @ #k
1007               & #k < #j)"
1008
1009 //
     ==========================================================================
1010 // F: AS FLOW
1011 //
     ==========================================================================
1012 lemma F1_resource_access_requires_token_exchange:
1013     all-traces
1014     "All params #i.
1015        AS_Request_Accepted(<'resource_access', params>) @ #i
1016        ==> (Ex code #j.
1017               AS_Request_Accepted(<'token_exchange', code>) @ #j
1018               & #j < #i)"
1019
1020 lemma F2_token_exchange_requires_authorization:
1021     all-traces
1022     "All code #i.
1023        AS_Request_Accepted(<'token_exchange', code>) @ #i
```

```
       ==> (Ex nonce #j.
              AS_Request_Accepted(<'authorization', nonce>) @ #j
              & #j < #i)"

lemma F3_authorization_requires_challenge:
    all-traces
    "All nonce #i.
       AS_Request_Accepted(<'authorization', nonce>) @ #i
       ==> (Ex #j.
              AS_Request_Accepted(<'challenge', nonce>) @ #j
              & #j < #i)"

//
    ================================================================================
// CIBA-SPECIFIC
//
    ================================================================================

lemma CIBA1_authcode_after_user_auth:
    all-traces
    "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
    iss) @ #i
     ==> (Ex auth_req_id #j.
            UserAuth_Initiated(auth_req_id) @ #j
            & CIBA_AuthEvent_Bound(auth_req_id, code) @ #i
            & #j < #i)"

lemma CIBA2_complete_after_request:
    all-traces
    "All ClientID #i. AS_Auth_Completed(ClientID) @ #i
     ==> (Ex auth_req_id #j #k.
            UserAuth_Initiated(auth_req_id) @ #j
            & ASOnly_AuthReq_ClientID_Action(auth_req_id, ClientID) @ #k
            & #k < #j
            & #j < #i)"

lemma CIBA3_auth_req_id_unique:
    all-traces
    "All id #i #j.
       AS_UserAuthAccepted(id) @ #i & AS_UserAuthAccepted(id) @ #j
       ==> #i = #j"

lemma CIBA4_auth_event_binding:
    all-traces
    "All id code1 code2 #i #j.
       CIBA_AuthEvent_Bound(id, code1) @ #i
       & CIBA_AuthEvent_Bound(id, code2) @ #j
       ==> #i = #j"
```

```
lemma CIBA5_code_single_issuance:
    all-traces
    "All id #i #j.
       CIBA_Code_Issued(id) @ #i & CIBA_Code_Issued(id) @ #j
       ==> #i = #j"

lemma CIBA6_no_unsolicited_auth:
    all-traces
    "All auth_req_id #i.
       AS_UserAuthAccepted(auth_req_id) @ #i
       ==> (Ex ClientID #j.
              ASOnly_AuthReq_ClientID_Action(auth_req_id, ClientID) @ #j
              & #j < #i)"

lemma CIBA7_poll_authorized_implies_user_auth:
    all-traces
    "All auth_req_id #i.
       AS_Request_Accepted(<'poll_authorized', auth_req_id>) @ #i
       ==> (Ex #j. AS_UserAuthAccepted(auth_req_id) @ #j & #j < #i)"

lemma CIBA8_code_requires_poll[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All auth_req_id #i.
       CIBA_Code_Issued(auth_req_id) @ #i
       ==> (Ex #j. CIBA_Poll_Sent(auth_req_id) @ #j & #j < #i)"

lemma CIBA9_auth_req_id_bound_to_client:
    all-traces
    "All auth_req_id ClientID1 ClientID2 #i #j.
       ASOnly_AuthReq_ClientID_Action(auth_req_id, ClientID1) @ #i
       & ASOnly_AuthReq_ClientID_Action(auth_req_id, ClientID2) @ #j
       ==> ClientID1 = ClientID2 & #i = #j"

end
```

B.2 FIDO

```
theory mAuth_FIDO
begin

builtins: signing
heuristic: S

rule ChanOut_S:
```

```
    [ Out_S($A, $B, x) ]
    --[ ChanOut_S($A, $B, x) ]->
    [ Sec($A, $B, x) ]

rule ChanIn_S:
    [ Sec($A, $B, x) ]
    --[ ChanIn_S($A, $B, x) ]->
    [ In_S($A, $B, x) ]

//
    ================================================================================
// SETUP RULES (consolidated)
//
    ================================================================================

rule Setup_AS_Identity[role="AS"]:
    [ Fr(~iss) ]
    -->
    [ !AS_Identity(~iss) ]

rule Setup_Resource_Server[role="RS"]:
    [ Fr(~data) ]
    -->
    [ !RS_Resource(~data) ]

rule Setup_Backend_Identity[role="Backend"]:
    [ Fr(~backend_sk) ]
    --[ Secret_BackendKey(~backend_sk) ]->
    [ !Backend_Ltk(~backend_sk),
      !PkBackend(pk(~backend_sk)),
      !AS_Knows_Backend(pk(~backend_sk)) ]


rule Setup_AS_rpId[role="AS"]:
    [ Fr(~rpId) ]
    --[ AS_rpId_Registered(~rpId) ]->
    [ !AS_rpId(~rpId) ]

rule Setup_FIDO_Credential[role="FIDOAuthenticator"]:
    [ !AS_rpId(rpId),
      Fr(~ClientID),
      Fr(~FIDO_sk) ]
    --[ Secret_FIDOKey(~ClientID, ~FIDO_sk),
        FIDO_Credential_Registered(~ClientID, pk(~FIDO_sk), rpId) ]->
    [ !FIDO_Ltk(~ClientID, ~FIDO_sk, rpId),
      !AS_FIDO_Credential(~ClientID, pk(~FIDO_sk), rpId),
      ClientReady(~ClientID) ]

```

```
//
    ================================================================================
// ATTESTATION CHALLENGE (Steps 2-3)
//
    ================================================================================

rule Client_AttestChallReq_ToBackend_Step2[role="Client"]:
    [ ClientReady(ClientID) ]
    -->
    [ Out_S($Client, $Backend, <'step2_c2b', ClientID>),
      Client_KeygenSlot(ClientID) ]

rule Backend_AttestChallReq_ToAS_Step2[role="Backend"]:
    [ In_S($Client, $Backend, <'step2_c2b', ClientID>) ]
    -->
    [ Out_S($Backend, $AS, <'step2_b2as', ClientID>) ]

rule AS_IssueChallenge_ToBackend_Step3[role="AS"]:
    [ In_S($Backend, $AS, <'step2_b2as', ClientID>),
      Fr(~nonce),
      !AS_Identity(iss) ]
    --[ AS_Issued_Nonce(~nonce),
        AS_Request_Accepted(<'challenge', ~nonce>) ]->
    [ Out_S($AS, $Backend, <'step3_as2b', ClientID, ~nonce, iss>),
      AS_Pending_Challenge_Nonce_Useable(ClientID, ~nonce, iss) ]

rule Backend_IssueChallenge_ToClient_Step3[role="Backend"]:
    [ In_S($AS, $Backend, <'step3_as2b', ClientID, nonce, iss>) ]
    -->
    [ Out_S($Backend, $Client, <'step3_b2c', ClientID, nonce, iss>) ]

//
    ================================================================================
// KEY GENERATION & ATTESTATION (Steps 4-7)
//
    ================================================================================

rule Client_GenerateKey_ToClient_Step4[role="Client"]:
    [ In_S($Backend, $Client, <'step3_b2c', ClientID, nonce, iss>),
      Fr(~skC) ]
    --[ Secret_ClientKey(ClientID, ~skC),
        Client_Key_In_Hardware(pk(~skC)),
        Client_Learned_Iss(ClientID, iss) ]->
    [ Client_GenerateKey_ToClient_Step4(ClientID, nonce),
      !Client_Key(ClientID, ~skC, pk(~skC)),
      !Client_Expected_Iss(ClientID, iss) ]

rule Client_RequestAttestation_ToAA_Step5[role="Client"]:
    [ Client_GenerateKey_ToClient_Step4(ClientID, nonce),
```

```
99        !Client_Key(ClientID, skC, pkC),
100       Client_KeygenSlot(ClientID),
101       Fr(~Device_Info) ]
102     --[ Client_Sent_Attestation_Request(ClientID, nonce, pkC) ]->
103     [ Client_RequestAttestation_ToAA_Step5(<~Device_Info, nonce, pkC>),
104       Client_AttestRequestPending_ToClient_Step5To8(ClientID, nonce,
    ~Device_Info),
105       Client_ClientIDForKeygen_ToAA(ClientID) ]
106
107 rule Setup_AA_Enclave[role="AA"]:
108     [ Client_ClientIDForKeygen_ToAA(ClientID),
109       Fr(~aa_sk) ]
110     --[ Secret_AAKey(~aa_sk) ]->
111     [ !AA_Ltk(~aa_sk),
112       !PkAA(ClientID, pk(~aa_sk))]
113
114 rule AA_SignAttestation_ToAA_Step6[role="AA"]:
115     [ Client_RequestAttestation_ToAA_Step5(<Device_Info, nonce, pkC>),
116       !AA_Ltk(aa_sk) ]
117     --[ AA_Signed_Attestation(nonce, pkC),
118         AA_NonceUsed(nonce) ]->
119     [ AA_SignAttestation_ToAA_Step6(sign(<Device_Info, nonce, pkC>,
    aa_sk)) ]
120
121 rule AA_SendAttestation_ToClient_Step7[role="AA"]:
122     [ AA_SignAttestation_ToAA_Step6(attestJWT) ]
123     -->
124     [ AA_SendAttestation_ToClient_Step7(attestJWT) ]
125
126 rule Client_ForwardPlatformAttest_ToBackend_Step7b[role="Client"]:
127     [ AA_SendAttestation_ToClient_Step7(platformAttest),
128       Client_AttestRequestPending_ToClient_Step5To8(ClientID, nonce,
    Device_Info),
129       !Client_Key(ClientID, skC, pkC) ]
130     -->
131     [ Out_S($Client, $Backend, <'wrap_cat_c2b', ClientID, nonce, pkC,
    Device_Info, platformAttest>),
132       Client_AwaitingCAT_ToClient(ClientID, nonce, Device_Info,
    platformAttest, pkC) ]
133
134 rule Backend_WrapCAT_ToClient_Step7c[role="Backend"]:
135     [ In_S($Client, $Backend, <'wrap_cat_c2b', ClientID, nonce, pkC,
    Device_Info, platformAttest>),
136       !Backend_Ltk(backend_sk) ]
137     --[ Backend_Issued_CAT(ClientID, pkC, nonce) ]->
138     [ Out_S($Backend, $Client,
139         <'cat_response_b2c', ClientID,
140           sign(<'client-attestation+jwt', ClientID, pkC, nonce,
    platformAttest>, backend_sk),
```

```
141             platformAttest>) ]
142
143 //
    ================================================================================
144 // DPoP CREATION & AUTHORIZATION REQUEST (Steps 8-10)
145 //
    ================================================================================
146
147 rule Client_CreateDPoPP_ToClient_Step8[role="Client"]:
148     [ In_S($Backend, $Client, <'cat_response_b2c', ClientID,
    clientAttestJWT, platformAttest>),
149       Client_AwaitingCAT_ToClient(ClientID, nonce, Device_Info,
    platformAttest, pkC),
150       !Client_Key(ClientID, skC, pkC),
151       Fr(~DPoP_Params) ]
152     --[ Client_Created_DPoP(pkC, ~DPoP_Params,'authz') ]->
153     [ Client_DPoPAttestation_ToClient_Step8(ClientID, nonce,
    clientAttestJWT, platformAttest, Device_Info),
154       Client_CreateDPoPP_ToClient_Step8(
155           sign(<~DPoP_Params, nonce>, skC), pkC, ~DPoP_Params, nonce) ]
156
157 rule Client_SendAuthzReq_ToBackend_Step9[role="Client"]:
158     [ Client_DPoPAttestation_ToClient_Step8(ClientID, nonce,
    clientAttestJWT, platformAttest, Device_Info),
159       Client_CreateDPoPP_ToClient_Step8(dpopp, pkC, DPoP_Params, nonce) ]
160     -->
161     [ Out_S($Client, $Backend,
162         <'step9_c2b', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
163          dpopp, pkC, DPoP_Params, pkC>) ]
164
165 rule Backend_SendAuthzReq_ToAS_Step9[role="Backend"]:
166     [ In_S($Client, $Backend,
167         <'step9_c2b', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
168          dpopp, pkC, DPoP_Params, dpop_jkt>) ]
169     -->
170     [ Out_S($Backend, $AS,
171         <'step9_b2as', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
172          dpopp, pkC, DPoP_Params, dpop_jkt>) ]
173
174 rule AS_VerifyJWTAndDPoPP_ToBackend_Step10[role="AS"]:
175     [ In_S($Backend, $AS, <'step9_b2as', ClientID, nonce, Device_Info,
    platformAttest, clientAttestJWT, dpopp, pkC, DPoP_Params, dpop_jkt>),
176       AS_Pending_Challenge_Nonce_Useable(ClientID, nonce, iss),
177       !PkAA(ClientID,pkAA),
178       !AS_Knows_Backend(pkBackend),
179       !AS_Identity(iss),
```

```
180       !AS_FIDO_Credential(ClientID, pkFIDO, rpId),
181       Fr(~FIDONonce) ]
182     --[ Eq(verify(platformAttest, <Device_Info, nonce, pkC>, pkAA), true),
183         Eq(verify(dpopp, <DPoP_Params, nonce>, pkC), true),
184         Eq(verify(clientAttestJWT, <'client-attestation+jwt', ClientID,
    pkC, nonce, platformAttest>, pkBackend), true),
185         Eq(dpop_jkt, pkC),
186         AS_Request_Accepted(<'authorization', nonce>),
187         AS_Accepted_Client_Key(ClientID, pkC),
188         AS_Verified_CAT(ClientID, pkC),
189         AS_Verified_PlatformAttest(ClientID, pkC),
190         AS_Verified_dpop_jkt(ClientID, pkC),
191         DPoP_Used(DPoP_Params),
192         DPoP_Verified(DPoP_Params, pkC),
193         AS_Issued_FIDONonce(~FIDONonce, ClientID, rpId)]->
194     [ Out_S($AS, $Backend, <'fido_challenge_as2b', ClientID, ~FIDONonce,
    iss, rpId>),
195       AS_PendingFIDOAuth(ClientID, ~FIDONonce, iss, rpId),
196       !AS_Client_Tied_PublicKey(ClientID, pkC, iss) ]
197
198 //
    ================================================================================
199 // FIDO2 AUTHENTICATION (Step 11)
200 //
    ================================================================================
201
202 rule Backend_SendChallenge_ToClient_FIDOStep1[role="Backend"]:
203     [ In_S($AS, $Backend, <'fido_challenge_as2b', ClientID, FIDONonce,
    iss, rpId>) ]
204     -->
205     [ Out_S($Backend, $Client,<'fido_challenge_b2c', ClientID, FIDONonce,
    iss, rpId>) ]
206
207 rule Client_StartAuth_ToFIDOAuthenticator_FIDOStep2[role="Client"]:
208     [ In_S($Backend, $Client, <'fido_challenge_b2c', ClientID, FIDONonce,
    iss, rpId>),
209       !Client_Expected_Iss(ClientID, iss) ]
210     -->
211     [ Client_StartAuth_ToFIDOAuthenticator_FIDOStep2(ClientID, FIDONonce,
    rpId) ]
212
213 rule FIDOAuthenticator_Authenticated[role="FIDOAuthenticator"]:
214     [ Client_StartAuth_ToFIDOAuthenticator_FIDOStep2(ClientID, FIDONonce,
    rpId),
215       !FIDO_Ltk(ClientID, skFIDO, rpIdStored) ]
216     --[ FIDOAuth_Initiated(ClientID, rpId),
217         Eq(rpId, rpIdStored),
218         FIDO_Authenticator_Signed(ClientID, rpId, FIDONonce) ]->
219     [ FIDOAuthenticator_Authenticated(
```

```
220           sign(<rpId, FIDONonce>, skFIDO),
221           ClientID, FIDONonce, rpId) ]
222
223 rule Client_SendAuth_ToBackend_FIDOStep4[role="Client"]:
224     [ FIDOAuthenticator_Authenticated(SignedAuth, ClientID, FIDONonce,
    rpId) ]
225     -->
226     [ Out_S($Client, $Backend, <'fido_auth_c2b', SignedAuth, ClientID,
    FIDONonce, rpId>) ]
227
228 rule Backend_SendAuth_ToAS_FIDOStep4[role="Backend"]:
229     [ In_S($Client, $Backend, <'fido_auth_c2b', SignedAuth, ClientID,
    FIDONonce, rpId>) ]
230     -->
231     [ Out_S($Backend, $AS, <'fido_auth_b2c', SignedAuth, ClientID,
    FIDONonce, rpId>) ]
232
233 rule AS_VerifyChallengeSendCode_ToBackend_FIDOStep5[role="AS"]:
234     [ In_S($Backend, $AS,<'fido_auth_b2c', SignedAuth, ClientID,
    FIDONonce, rpId>),
235       !AS_FIDO_Credential(ClientID, pkFIDO, rpIdStored),
236       !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
237       Fr(~authorization_code),
238       AS_PendingFIDOAuth(ClientID, FIDONonce, iss, rpIdChallenge) ]
239     --[ Eq(rpId, rpIdStored),
240         Eq(rpId, rpIdChallenge),
241         Eq(verify(SignedAuth, <rpId, FIDONonce>, pkFIDO), true),
242         AS_Verified_rpIdHash(ClientID, rpId),
243         FIDO_Auth_Success(ClientID, rpId),
244         AS_Request_Accepted(<'fido_auth', FIDONonce>),
245         AS_Auth_Completed(ClientID),
246         AuthzCode_Issued(~authorization_code, ClientID, pkC, iss) ]->
247     [ Out_S($AS, $Backend, <'auth_completed_as2b', ClientID,
    ~authorization_code>),
248       AuthzCode(~authorization_code, ClientID, pkC, iss) ]
249
250 //
    ================================================================================
251 // TOKEN EXCHANGE & RESOURCE ACCESS (Steps 12-19)
252 //
    ================================================================================
253
254 rule Backend_RequestDPoP_ForTokenExchange[role="Backend"]:
255     [ In_S($AS, $Backend, <'auth_completed_as2b', ClientID,
    authorization_code>) ]
256     -->
257     [ Out_S($Backend, $Client, <'dpop_needed_for_token_b2c', ClientID>),
258       Backend_PendingTokenExchange(ClientID, authorization_code) ]
259
```

```
rule Client_CreateDPoP_ForTokenExchange[role="Client"]:
    [ In_S($Backend, $Client, <'dpop_needed_for_token_b2c', ClientID>),
      !Client_Key(ClientID, skC, pkC),
      Fr(~DPoP_Params_TE) ]
    --[ Client_Created_DPoP(pkC, ~DPoP_Params_TE,'token_te') ]->
    [ Out_S($Client, $Backend, <'dpop_for_token_c2b',
   sign(<~DPoP_Params_TE,'token_endpoint'>, skC), pkC, ~DPoP_Params_TE,
   ClientID>) ]

rule Backend_TokenRequest_ToAS_Step12[role="Backend"]:
    [ In_S($Client, $Backend, <'dpop_for_token_c2b', dpopp_te, pkC,
   DPoP_Params_TE, ClientID>),
      Backend_PendingTokenExchange(ClientID, authorization_code) ]
    -->
    [ Out_S($Backend, $AS, <'token_req_b2as', ClientID,
   authorization_code,
            dpopp_te, pkC, DPoP_Params_TE>) ]

rule AS_TokenAns_ToBackend_Step13[role="AS"]:
    [ In_S($Backend, $AS, <'token_req_b2as', ClientID, authorization_code,
            dpopp_te, pkC, DPoP_Params_TE>),
      AuthzCode(authorization_code, ClientID, pkC, iss),
      !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
      Fr(~constrained_access_token),
      Fr(~DPoP_reference_token),
      Fr(~refresh_token),
      !AS_Identity(iss) ]
    --[ Eq(verify(dpopp_te, <DPoP_Params_TE,'token_endpoint'>, pkC),
   true),
        DPoP_Used(DPoP_Params_TE),
        DPoP_Verified(DPoP_Params_TE, pkC),
        AS_Request_Accepted(<'token_exchange', authorization_code>),
        AuthzCode_Redeemed(authorization_code, ClientID, pkC),
        Token_Issued(~constrained_access_token, ClientID, pkC),
        RefToken_Issued(~DPoP_reference_token, ClientID, pkC),
        RefreshToken_Issued(~refresh_token, ClientID, pkC),
        Token_Issued_By_Iss(ClientID, iss) ]->
    [ !ConstrainedAccess(~constrained_access_token, pkC),
      !ConstrainedReference(~DPoP_reference_token, pkC, iss),
      !ConstrainedRefresh(~refresh_token, pkC),
      Out_S($AS, $Backend, <'token_resp_as2b', ClientID,
   ~DPoP_reference_token,
             ~constrained_access_token, pkC, iss, ~refresh_token>) ]

rule Backend_SendReferenceToken_ToClient_Step14[role="Backend"]:
    [ In_S($AS, $Backend, <'token_resp_as2b', ClientID,
   DPoP_reference_token,
            constrained_access_token, pkC, iss, refresh_token>) ]
    -->
```

```
302     [ Out_S($Backend, $Client, <'ref_token_b2c', ClientID,
    DPoP_reference_token, iss>),
303       Backend_AccessTokenHeld(DPoP_reference_token,
    constrained_access_token, pkC, ClientID),
304       Backend_RefreshTokenHeld(refresh_token, pkC, ClientID) ]
305
306 rule Client_GenerateDPoP_Step15_SendDPoP_ToBackend_Step16[role="Client"]:
307     [ In_S($Backend, $Client, <'ref_token_b2c', ClientID,
    DPoP_reference_token, iss>),
308       !Client_Key(ClientID, skC, pkC),
309       !Client_Expected_Iss(ClientID, iss),
310       Fr(~DPoP_Params2) ]
311     --[ Client_Verified_Iss(ClientID, iss),
312         Client_Created_DPoP(pkC, ~DPoP_Params2,'resource') ]->
313     [ Out_S($Client, $Backend, <'dpop_resource_c2b',
314             sign(<~DPoP_Params2, DPoP_reference_token>, skC),
315             pkC, ~DPoP_Params2, DPoP_reference_token>),
316       Client_CanRefresh(ClientID) ]
317
318 rule Backend_ResourceRequest_ToRS_Step17[role="Backend"]:
319     [ In_S($Client,$Backend, <'dpop_resource_c2b', dpopp, pkC,
    DPoP_Params2, DPoP_reference_token>),
320       Backend_AccessTokenHeld(DPoP_reference_token,
    constrained_access_token, pkC, ClientID) ]
321     -->
322     [ Out_S($Backend, $RS, <'rs_verify_b2rs', constrained_access_token,
    dpopp, pkC,
323             DPoP_Params2, DPoP_reference_token, ClientID>) ]
324
325 rule RS_ForwardIntrospection_ToAS[role="RS API"]:
326     [ In_S($Backend, $RS, <'rs_verify_b2rs', constrained_access_token,
    dpopp, pkC,
327            DPoP_Params2, DPoP_reference_token, ClientID>)]
328     -->
329     [ Out_S($RS, $AS, <'introspect', constrained_access_token, dpopp, pkC,
330             DPoP_Params2, DPoP_reference_token, ClientID>) ,
331             RS_Client_Tied_Access(ClientID,constrained_access_token)]
332
333 rule AS_VerifyDPoPPAndTokens_Step18[role="AS"]:
334     [ In_S($RS, $AS, <'introspect', constrained_access_token, dpopp, pkC,
335           DPoP_Params2, DPoP_reference_token, ClientID>),
336       !ConstrainedAccess(constrained_access_token, pkC),
337       !ConstrainedReference(DPoP_reference_token, pkC, iss),
338       !AS_Client_Tied_PublicKey(ClientID, pkC, iss) ,
339       !AS_Identity(iss) ]
340     --[ Eq(verify(dpopp, <DPoP_Params2, DPoP_reference_token>, pkC),
    true),
341         DPoP_Used(DPoP_Params2),
342         DPoP_Verified(DPoP_Params2, pkC),
```

```
343         AS_Request_Accepted(<'resource_access', DPoP_Params2>),
344         Token_Used_For_Access(constrained_access_token, ClientID, pkC) ]->
345     [ Out_S($AS, $RS, <'verified', ClientID, constrained_access_token>) ]
346
347 rule RS_SendResource_Step19[role="RS API"]:
348     [ In_S($AS, $RS, <'verified', ClientID, constrained_access_token>),
349       !RS_Resource(data),
350       RS_Client_Tied_Access(ClientID, constrained_access_token) ]
351     --[ RS_Sent_Resource(data, ClientID, constrained_access_token),
352         RS_Access_Granted(data, ClientID) ]->
353     [ Out_S($RS, $Client, <'resource_sent', data, ClientID>) ]
354
355 rule Client_Received_Data[role="Client"]:
356    [In_S($RS,$Client,<'resource_sent',data,ClientID>)]
357    --[Client_Recourse_Received(data,ClientID)]->
358    []
359
360 //
    ================================================================================
361 // REFRESH TOKEN EXCHANGE
362 //
    ================================================================================
363
364 rule Client_RequestRefresh[role="Client"]:
365     [ Client_CanRefresh(ClientID),
366       !Client_Key(ClientID, skC, pkC),
367       Fr(~DPoP_Params_Refresh) ]
368     --[ Client_Created_DPoP(pkC, ~DPoP_Params_Refresh,'refresh'),
369         Client_Requested_Refresh(ClientID) ]->
370     [ Out_S($Client, $Backend, <'refresh_req_c2b',
371             sign(<~DPoP_Params_Refresh, 'refresh'>, skC),
372             pkC, ~DPoP_Params_Refresh, ClientID>) ]
373
374 rule Backend_RefreshRequest_ToAS[role="Backend"]:
375     [ In_S($Client, $Backend, <'refresh_req_c2b', dpopp, pkC,
    DPoP_Params_R, ClientID>),
376       Backend_RefreshTokenHeld(refresh_token, pkC, ClientID) ]
377     -->
378     [ Out_S($Backend, $AS, <'refresh_req_b2as', refresh_token, dpopp,
    pkC, DPoP_Params_R, ClientID>) ]
379
380 rule AS_VerifyRefresh_IssueToken[role="AS"]:
381     [ In_S($Backend, $AS, <'refresh_req_b2as', refresh_token, dpopp, pkC,
    DPoP_Params_R, ClientID>),
382       !ConstrainedRefresh(refresh_token, pkC),
383       !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
384       !AS_Identity(iss)  ]
385     --[ Eq(verify(dpopp, <DPoP_Params_R,'refresh'>, pkC), true),
386         DPoP_Used(DPoP_Params_R),
```

```
387         DPoP_Verified(DPoP_Params_R, pkC),
388         AS_Request_Accepted(<'refresh', DPoP_Params_R>),
389         RefreshToken_Used(refresh_token, ClientID, pkC) ]->
390     [ AS_RefreshCompleted(ClientID) ]
391
392 //
    ==========================================================================
393 // LEAK / ATTACKER EVENTS
394 //
    ==========================================================================
395
396 rule Leak_Access_Token:
397     [ !ConstrainedAccess(token, pkC) ]
398     --[ LeakToken(token, pkC) ]->
399     [ Out(token) ]
400
401 rule Leak_Reference_Token:
402     [ !ConstrainedReference(ref_token, pkC, iss) ]
403     --[ LeakRefToken(ref_token, pkC) ]->
404     [ Out(ref_token) ]
405
406 rule Leak_Refresh_Token:
407     [ !ConstrainedRefresh(refresh_token, pkC) ]
408     --[ LeakRefreshToken(refresh_token, pkC) ]->
409     [ Out(refresh_token) ]
410
411 rule Reveal_AA_Key:
412     [ !AA_Ltk(aa_sk) ]
413     --[ RevealAA(aa_sk) ]->
414     [ Out(aa_sk) ]
415
416 rule Reveal_Client_Key:
417     [ !Client_Key(ClientID, skC, pkC) ]
418     --[ RevealClientKey(ClientID, skC) ]->
419     [ Out(skC) ]
420
421 rule Reveal_Backend_Key:
422     [ !Backend_Ltk(backend_sk) ]
423     --[ RevealBackend(backend_sk) ]->
424     [ Out(backend_sk) ]
425
426 rule Reveal_FIDO_Key:
427     [ !FIDO_Ltk(ClientID, skFIDO, rpId) ]
428     --[ RevealFIDO(ClientID) ]->
429     [ Out(skFIDO) ]
430
431 //
    ==========================================================================
432 // RESTRICTIONS
```

```
433 //
        ================================================================================
434
435 restriction Equality:
436     "All x y #i. Eq(x,y) @ #i ==> x = y"
437
438 restriction unique_dpopp:
439     "All params #i #j.
440        DPoP_Used(params) @ #i & DPoP_Used(params) @ #j
441        ==> #i = #j"
442
443 restriction no_replay:
444     "All id #i #j.
445        AS_Request_Accepted(id) @ #i & AS_Request_Accepted(id) @ #j
446        ==> #i = #j"
447
448 restriction unique_attestation_per_nonce:
449     "All nonce pkC1 pkC2 #i #j.
450        AA_Signed_Attestation(nonce, pkC1) @ #i
451        & AA_Signed_Attestation(nonce, pkC2) @ #j
452        ==> #i = #j"
453
454 //
        ================================================================================
455 // SOURCES
456 //
        ================================================================================
457
458 lemma typing [sources]:
459   " (All t cid pk #i #j.
460         Token_Issued(t, cid, pk) @ #j & !KU(t) @ #i
461         ==> (Ex #l. LeakToken(t, pk) @ #l & #l < #i))
462   & (All t cid pk #i #j.
463         RefToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
464         ==> (Ex #l. LeakRefToken(t, pk) @ #l & #l < #i))
465   & (All t cid pk #i #j.
466         RefreshToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
467         ==> (Ex #l. LeakRefreshToken(t, pk) @ #l & #l < #i))
468   & (All c cid pk iss #i #j.
469         AuthzCode_Issued(c, cid, pk, iss) @ #j & !KU(c) @ #i
470         ==> F)
471   & (All n #i #j.
472         AS_Issued_Nonce(n) @ #j & !KU(n) @ #i
473         ==> F)
474   & (All n cid rpId #i #j.
475         AS_Issued_FIDONonce(n, cid, rpId) @ #j & !KU(n) @ #i
476         ==> F)
477   "
478
```

```
479
480 lemma token_implies_code [reuse]:
481   "All t cid pk #i. Token_Issued(t, cid, pk) @ #i
482    ==> (Ex c iss #j. AuthzCode_Issued(c, cid, pk, iss) @ #j & #j < #i)"
483
484 lemma code_implies_accept [reuse]:
485   "All c cid pk iss #i. AuthzCode_Issued(c, cid, pk, iss) @ #i
486    ==> (Ex #j. AS_Accepted_Client_Key(cid, pk) @ #j & #j < #i)"
487
488 lemma access_implies_issued [reuse, use_induction]:
489   "All t cid pk #i. Token_Used_For_Access(t, cid, pk) @ #i
490    ==> (Ex #j. Token_Issued(t, cid, pk) @ #j & #j < #i)"
491
492 lemma refresh_used_implies_issued [reuse]:
493   "All t cid pk #i. RefreshToken_Used(t, cid, pk) @ #i
494    ==> (Ex #j. RefreshToken_Issued(t, cid, pk) @ #j & #j < #i)"
495
496 lemma accept_unique_pkC [reuse, use_induction]:
497   "All cid pk1 pk2 #i #j.
498       AS_Accepted_Client_Key(cid, pk1) @ #i
499     & AS_Accepted_Client_Key(cid, pk2) @ #j
500     ==> pk1 = pk2"
501
502 lemma client_iss_unique [reuse, use_induction]:
503   "All cid iss1 iss2 #i #j.
504       Client_Learned_Iss(cid, iss1) @ #i
505     & Client_Learned_Iss(cid, iss2) @ #j
506     ==> iss1 = iss2 & #i = #j"
507
508 //
    ================================================================================
509 // UNIQUE PROOFS
510 //
    ================================================================================
511
512 lemma L_unique_dpopp_proof:
513     all-traces
514     "All params #i #j.
515        DPoP_Used(params) @ #i & DPoP_Used(params) @ #j
516        ==> #i = #j"
517
518 lemma L_unique_attestation_proof:
519     all-traces
520     "All nonce pkC1 pkC2 #i #j.
521        AA_Signed_Attestation(nonce, pkC1) @ #i
522        & AA_Signed_Attestation(nonce, pkC2) @ #j
523        ==> #i = #j & pkC1 = pkC2"
524
```

```
//
    ================================================================================
// SANITY
//
    ================================================================================

lemma SANITY_keygen[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key,
   hide_lemma=accept_unique_pkC,
   hide_lemma=client_iss_unique,
   hide_lemma=token_implies_code,
   hide_lemma=code_implies_accept,
   hide_lemma=access_implies_issued,
   hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex cid pk #t. Client_Learned_Iss(cid, pk) @ #t"

lemma SANITY_token_issued[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key,
   hide_lemma=accept_unique_pkC,
   hide_lemma=client_iss_unique,
   hide_lemma=token_implies_code,
   hide_lemma=code_implies_accept,
   hide_lemma=access_implies_issued,
   hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex t cid pk #i. Token_Issued(t, cid, pk) @ #i"

lemma SANITY_dpop_verified[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key,
   hide_lemma=accept_unique_pkC,
   hide_lemma=client_iss_unique,
   hide_lemma=token_implies_code,
   hide_lemma=code_implies_accept,
   hide_lemma=access_implies_issued,
   hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex p pk #i. DPoP_Verified(p, pk) @ #i"

lemma SANITY_token_used[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
```

```
572     hide_lemma=I1_token_requires_hardware_key,
573     hide_lemma=accept_unique_pkC,
574     hide_lemma=client_iss_unique,
575     hide_lemma=token_implies_code,
576     hide_lemma=code_implies_accept,
577     hide_lemma=access_implies_issued,
578     hide_lemma=refresh_used_implies_issued]: exists-trace
579   "Ex t cid pk #i. Token_Used_For_Access(t, cid, pk) @ #i"
580
581 lemma SANITY_refresh_used[hide_lemma=R2_dpop_requires_creation,
582     hide_lemma=B1_attestation_unforgeable,
583     hide_lemma=B2_as_requires_attestation,
584     hide_lemma=B3_token_implies_attestation,
585     hide_lemma=I1_token_requires_hardware_key,
586     hide_lemma=accept_unique_pkC,
587     hide_lemma=client_iss_unique,
588     hide_lemma=token_implies_code,
589     hide_lemma=code_implies_accept,
590     hide_lemma=access_implies_issued,
591     hide_lemma=refresh_used_implies_issued]: exists-trace
592   "Ex t cid pk #i. RefreshToken_Used(t, cid, pk) @ #i"
593
594 lemma SANITY_resource_received[hide_lemma=R2_dpop_requires_creation,
595     hide_lemma=B1_attestation_unforgeable,
596     hide_lemma=B2_as_requires_attestation,
597     hide_lemma=B3_token_implies_attestation,
598     hide_lemma=I1_token_requires_hardware_key,
599     hide_lemma=accept_unique_pkC,
600     hide_lemma=client_iss_unique,
601     hide_lemma=token_implies_code,
602     hide_lemma=code_implies_accept,
603     hide_lemma=access_implies_issued,
604     hide_lemma=refresh_used_implies_issued]: exists-trace
605   "Ex data cid #t. Client_Recourse_Received(data, cid) @ #t"
606
607 lemma SANITY_full_chain[hide_lemma=R2_dpop_requires_creation,
608     hide_lemma=B1_attestation_unforgeable,
609     hide_lemma=B2_as_requires_attestation,
610     hide_lemma=B3_token_implies_attestation,
611     hide_lemma=I1_token_requires_hardware_key,
612     hide_lemma=accept_unique_pkC,
613     hide_lemma=client_iss_unique,
614     hide_lemma=token_implies_code,
615     hide_lemma=code_implies_accept,
616     hide_lemma=access_implies_issued,
617     hide_lemma=refresh_used_implies_issued]:
618      exists-trace
619      "Ex data ClientID nonce pkC code token iss rpId
620          #t1 #t2 #t3 #t4 #t5 #t6 #t7 #t8 #t9 #t10.
```

```
      AS_Issued_Nonce(nonce)                                    @ #t1
    & Client_Sent_Attestation_Request(ClientID, nonce, pkC) @ #t2
    & AA_Signed_Attestation(nonce, pkC)                         @ #t3
    & Backend_Issued_CAT(ClientID, pkC, nonce)                  @ #t4
    & AS_Accepted_Client_Key(ClientID, pkC)                     @ #t5
    & FIDOAuth_Initiated(ClientID, rpId)                        @ #t6
    & FIDO_Auth_Success(ClientID, rpId)                         @ #t7
    & AuthzCode_Issued(code, ClientID, pkC, iss)                @ #t7
    & Token_Issued(token, ClientID, pkC)                        @ #t8
    & Token_Used_For_Access(token, ClientID, pkC)               @ #t9
    & RS_Access_Granted(data, ClientID)                         @ #t10
    & #t1 < #t2 & #t2 < #t3 & #t3 < #t4 & #t4 < #t5
    & #t5 < #t6 & #t6 < #t7 & #t7 < #t8 & #t8 < #t9
    & #t9 < #t10"

//
    ================================================================================
// S: SECRET KEY SECRECY
//
    ================================================================================

lemma S1_client_key_secrecy:
    all-traces
    "All ClientID k #i. Secret_ClientKey(ClientID, k) @ #i
     ==> not (Ex #j. K(k) @ #j)
         | (Ex #r. RevealClientKey(ClientID, k) @ #r)"

lemma S2_access_token_secrecy:
    all-traces
    "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
     ==> not (Ex #j. K(token) @ #j)
         | (Ex #l. LeakToken(token, pkC) @ #l)"

lemma S3_reference_token_secrecy:
    all-traces
    "All token ClientID pkC #i. RefToken_Issued(token, ClientID, pkC) @ #i
     ==> not (Ex #j. K(token) @ #j)
         | (Ex #l. LeakRefToken(token, pkC) @ #l)"

lemma S4_aa_key_secrecy:
    all-traces
    "All k #i. Secret_AAKey(k) @ #i
     ==> not (Ex #j. K(k) @ #j)
         | (Ex #r. RevealAA(k) @ #r)"

lemma S5_refresh_token_secrecy:
    all-traces
    "All token ClientID pkC #i. RefreshToken_Issued(token, ClientID, pkC)
    @ #i
```

```
667      ==> not (Ex #j. K(token) @ #j)
668          | (Ex #l. LeakRefreshToken(token, pkC) @ #l)"
669
670  lemma S6_backend_key_secrecy:
671      all-traces
672      "All k #i. Secret_BackendKey(k) @ #i
673       ==> not (Ex #j. K(k) @ #j)
674           | (Ex #r. RevealBackend(k) @ #r)"
675
676  //
     ================================================================================
677  // Z: AUTHORIZATION CORRECTNESS
678  //
     ================================================================================
679
680  lemma Z1_code_after_attestation:
681      all-traces
682      "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
     iss) @ #i
683       ==> (Ex nonce #j #k.
684         AS_Issued_Nonce(nonce) @ #j
685         & AA_Signed_Attestation(nonce, pkC) @ #k & #j < #k & #k < #i)
686           | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
687
688  lemma Z2_no_double_redeem:
689      all-traces
690      "not (Ex code ClientID1 ClientID2 pkC1 pkC2 #i #j.
691              AuthzCode_Redeemed(code, ClientID1, pkC1) @ #i
692              & AuthzCode_Redeemed(code, ClientID2, pkC2) @ #j
693              & not (#i = #j))"
694
695  lemma Z3_no_auth_code_substitution:
696      all-traces
697      "All code ClientID1 ClientID2 pkC1 pkC2 iss #i #j.
698         AuthzCode_Issued(code, ClientID1, pkC1, iss) @ #i
699         & AuthzCode_Redeemed(code, ClientID2, pkC2) @ #j
700         ==> ClientID1 = ClientID2 & pkC1 = pkC2"
701
702
703  lemma Z4_code_unique_issuance:
704      all-traces
705      "All code ClientID1 ClientID2 pkC1 pkC2 iss1 iss2 #i #j.
706         AuthzCode_Issued(code, ClientID1, pkC1, iss1) @ #i
707         & AuthzCode_Issued(code, ClientID2, pkC2, iss2) @ #j
708         ==> #i = #j & ClientID1 = ClientID2 & pkC1 = pkC2 & iss1 = iss2"
709
710  //
     ================================================================================
711  // T: TOKEN BINDING / PROOF-OF-POSSESSION
```

```
712 //
    ==========================================================================
713
714 lemma T1_stolen_token_unusable:
715     all-traces
716     "All token pkC ClientID #i #j.
717        LeakToken(token, pkC) @ #i
718        & Token_Used_For_Access(token, ClientID, pkC) @ #j
719        ==> (Ex params ctx #k #l.
720               Client_Created_DPoP(pkC, params,ctx) @ #k
721               & DPoP_Verified(params, pkC) @ #l
722               & #k < #l & #l < #j)
723             | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
724
725 lemma T2_token_access_requires_pop:
726     all-traces
727     "All token ClientID pkC #i. Token_Used_For_Access(token, ClientID,
    pkC) @ #i
728      ==> (Ex params #j #k.
729             DPoP_Verified(params, pkC) @ #j
730             & AS_Accepted_Client_Key(ClientID, pkC) @ #k
731             & (#k < #j | #k = #j)
732             &  #j < #i)
733           | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
734
735 lemma T3_no_token_identity_crossing:
736     all-traces
737     "All token ClientID1 ClientID2 pkC1 pkC2 #i #j.
738        Token_Issued(token, ClientID1, pkC1) @ #i
739        & Token_Used_For_Access(token, ClientID2, pkC2) @ #j
740        ==> ClientID1 = ClientID2 & pkC1 = pkC2"
741
742 lemma T4_stolen_refresh_token_unusable:
743     all-traces
744     "All token pkC ClientID #i #j.
745        LeakRefreshToken(token, pkC) @ #i
746        & RefreshToken_Used(token, ClientID, pkC) @ #j
747        ==> (Ex params ctx #k. Client_Created_DPoP(pkC, params,ctx) @ #k &
    #k < #j)
748             | (Ex skC #k. RevealClientKey(ClientID, skC) @ #k)"
749
750 lemma T5_stolen_refresh_needs_dpop:
751     all-traces
752     "All token pkC ClientID #i #j.
753        LeakRefreshToken(token, pkC) @ #i
754        & RefreshToken_Used(token, ClientID, pkC) @ #j
755        ==> (Ex params ctx #k #l.
756               Client_Created_DPoP(pkC, params, ctx) @ #k
757               & DPoP_Verified(params, pkC) @ #l
```

```
758                & #k < #l & #l = #j)
759              | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
760
761 //
    ================================================================================
762 // R: REPLAY RESISTANCE AND FRESHNESS
763 //
    ================================================================================
764
765 lemma R1_dpop_requires_creation:
766     all-traces
767     "All params pkC #i. DPoP_Verified(params, pkC) @ #i
768      ==> (Ex ctx #j. Client_Created_DPoP(pkC, params, ctx) @ #j & #j < #i)
769          | (Ex ClientID skC #r. RevealClientKey(ClientID, skC) @ #r)"
770
771 //
    ================================================================================
772 // B: ATTESTATION BINDING
773 //
    ================================================================================
774
775 lemma B1_attestation_unforgeable[reuse]:
776     all-traces
777     "All nonce pkC #i. AA_Signed_Attestation(nonce, pkC) @ #i
778      ==> (Ex ClientID #j. Client_Sent_Attestation_Request(ClientID,
    nonce, pkC) @ #j & #j < #i)
779          | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
780
781 lemma B2_as_requires_attestation[reuse]:
782     all-traces
783     "All ClientID pkC #i. AS_Accepted_Client_Key(ClientID, pkC) @ #i
784      ==> (Ex nonce #j. AA_Signed_Attestation(nonce, pkC) @ #j & #j < #i)
785          | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
786
787 lemma B3_token_implies_attestation[reuse]:
788     all-traces
789     "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
790      ==> (Ex nonce #j #k.
791        AA_Signed_Attestation(nonce, pkC) @ #j
792        & AS_Accepted_Client_Key(ClientID, pkC) @ #k
793        & #j < #k & #k < #i)
794         | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
795
796 lemma B4_attestation_key_binding:
797     all-traces
798     "All ClientID1 ClientID2 pkC1 pkC2 nonce #i #j #k1 #k2.
799        AS_Accepted_Client_Key(ClientID1, pkC1) @ #i
800        & AS_Accepted_Client_Key(ClientID2, pkC2) @ #j
801        & AA_Signed_Attestation(nonce, pkC1) @ #k1
```

```
802         & AA_Signed_Attestation(nonce, pkC2) @ #k2
803         ==> pkC1 = pkC2"
804
805 lemma B5_attestation_nonce_single_use:
806     all-traces
807     "All nonce #i #j.
808         AA_NonceUsed(nonce) @ #i & AA_NonceUsed(nonce) @ #j
809         ==> #i = #j"
810
811 lemma B6_cat_unforgeable:
812     all-traces
813     "All ClientID pkC #i.
814         AS_Verified_CAT(ClientID, pkC) @ #i
815         ==> (Ex nonce #j. Backend_Issued_CAT(ClientID, pkC, nonce) @ #j &
    #j < #i)
816             | (Ex backend_sk #r. RevealBackend(backend_sk) @ #r)"
817
818 lemma B7_dpop_jkt_matches_instance_key:
819     all-traces
820     "All ClientID pkC #i.
821         AS_Verified_dpop_jkt(ClientID, pkC) @ #i
822         ==> (Ex nonce #j.
823                 Client_Sent_Attestation_Request(ClientID, nonce, pkC) @ #j
824                 & #j < #i)
825             | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
826
827 //
    ================================================================================
828 // K: KEY CONTINUITY
829 //
    ================================================================================
830
831 lemma K1_key_continuity_attestation_to_token:
832     all-traces
833     "All ClientID pkC_accept token pkC_token #i #j.
834         AS_Accepted_Client_Key(ClientID, pkC_accept) @ #i
835         & Token_Issued(token, ClientID, pkC_token) @ #j
836         & #i < #j
837         ==> pkC_accept = pkC_token
838             | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
839
840 lemma K2_key_continuity_token_to_access:
841     all-traces
842     "All token ClientID pkC1 pkC2 #i #j.
843         Token_Issued(token, ClientID, pkC1) @ #i
844         & Token_Used_For_Access(token, ClientID, pkC2) @ #j
845         ==> pkC1 = pkC2"
846
847 lemma K3_dpop_key_matches_accepted:
```

```
    all-traces
    "All params pkC_dpop ClientID pkC_accept token #i #j #k.
       DPoP_Verified(params, pkC_dpop) @ #i
       & AS_Accepted_Client_Key(ClientID, pkC_accept) @ #j
       & Token_Used_For_Access(token, ClientID, pkC_dpop) @ #k
       & #j < #i & #i = #k
       ==> pkC_dpop = pkC_accept
           | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

//
    ================================================================================
// I: HARDWARE BOUND KEY CONTINUITY
//
    ================================================================================

lemma I1_token_requires_hardware_key [reuse]:
  all-traces
  "All token ClientID pkC #i.
    Token_Issued(token, ClientID, pkC) @ #i
    ==>
      (Ex #j #k.
         Client_Key_In_Hardware(pkC) @ #j
         & AS_Accepted_Client_Key(ClientID, pkC) @ #k
         & #j < #k & #k < #i)
      | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)
      | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"


lemma I2_access_requires_hardware_key:
    all-traces
    "All token ClientID pkC #i #j.
       Token_Used_For_Access(token, ClientID, pkC) @ #i
       & Token_Issued(token, ClientID, pkC) @ #j
       & #j < #i
       ==> (Ex #k #l.
              Client_Key_In_Hardware(pkC) @ #k
              & AS_Accepted_Client_Key(ClientID, pkC) @ #l
              & #k < #l & #l < #j)
           | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)
           | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"

//
    ================================================================================
// E: END-TO-END
//
    ================================================================================

lemma E1_token_requires_authentication:
    all-traces
```

```
    "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
     ==> (Ex #j. AS_Auth_Completed(ClientID) @ #j & #j < #i)"

lemma E2_resource_matches_issued_token:
    all-traces
    "All data ClientID #i.
       Client_Recourse_Received(data, ClientID) @ #i
       ==> (Ex token pkC #j #k.
              RS_Sent_Resource(data, ClientID, token) @ #j
              & Token_Issued(token, ClientID, pkC) @ #k
              & #k < #j & #j < #i)"

lemma
    E3_resource_requires_fresh_dpop[hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All data ClientID #i. Client_Recourse_Received(data, ClientID) @ #i
     ==> (Ex pkC ctx params #j #k.
            Client_Created_DPoP(pkC, params, ctx) @ #j
            & DPoP_Verified(params, pkC) @ #k
            & #j < #k & #k < #i
            & (Ex #l. AS_Accepted_Client_Key(ClientID, pkC) @ #l))
         | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

lemma E4_refresh_requires_issuance:
    all-traces
    "All token ClientID pkC #i.
       RefreshToken_Used(token, ClientID, pkC) @ #i
       ==> (Ex #j.
              RefreshToken_Issued(token, ClientID, pkC) @ #j
              & #j < #i)"

//
    ================================================================================
// ISS: MIX-UP PREVENTION
//
    ================================================================================

lemma ISS1_consistency[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All ClientID iss #i.
       Client_Verified_Iss(ClientID, iss) @ #i
       ==> (Ex #j. Client_Learned_Iss(ClientID, iss) @ #j & #j < #i)"
```

```
939
940 lemma ISS2_no_mix_up[hide_lemma=R2_dpop_requires_creation,
941    hide_lemma=B1_attestation_unforgeable,
942    hide_lemma=B2_as_requires_attestation,
943    hide_lemma=B3_token_implies_attestation,
944    hide_lemma=I1_token_requires_hardware_key]:
945     all-traces
946     "All code ClientID pkC iss1 iss2 #i #j.
947        AuthzCode_Issued(code, ClientID, pkC, iss1) @ #i
948        & AuthzCode_Redeemed(code, ClientID, pkC) @ #j
949        & Token_Issued_By_Iss(ClientID, iss2) @ #j
950        ==> iss1 = iss2"
951
952 lemma ISS3_token_origin[hide_lemma=R2_dpop_requires_creation,
953    hide_lemma=B1_attestation_unforgeable,
954    hide_lemma=B2_as_requires_attestation,
955    hide_lemma=B3_token_implies_attestation,
956    hide_lemma=I1_token_requires_hardware_key]:
957     all-traces
958     "All ClientID iss #i. Token_Issued_By_Iss(ClientID, iss) @ #i
959      ==> (Ex code pkC #j. AuthzCode_Issued(code, ClientID, pkC, iss) @ #j
    & #j < #i)"
960
961 lemma ISS4_client_rejects_wrong_iss[hide_lemma=R2_dpop_requires_creation,
962    hide_lemma=B1_attestation_unforgeable,
963    hide_lemma=B2_as_requires_attestation,
964    hide_lemma=B3_token_implies_attestation,
965    hide_lemma=I1_token_requires_hardware_key]:
966     all-traces
967     "All ClientID iss_learned iss_verified #i #j.
968        Client_Learned_Iss(ClientID, iss_learned) @ #i
969        & Client_Verified_Iss(ClientID, iss_verified) @ #j
970        & #i < #j
971        ==> iss_learned = iss_verified"
972
973 lemma ISS5_refresh_preserves_iss[hide_lemma=R2_dpop_requires_creation,
974    hide_lemma=B1_attestation_unforgeable,
975    hide_lemma=B2_as_requires_attestation,
976    hide_lemma=B3_token_implies_attestation,
977    hide_lemma=I1_token_requires_hardware_key]:
978     all-traces
979     "All ClientID iss_orig token pkC #i #j.
980        Token_Issued_By_Iss(ClientID, iss_orig) @ #i
981        & RefreshToken_Used(token, ClientID, pkC) @ #j
982        & #i < #j
983        ==> (Ex code #k.
984               AuthzCode_Issued(code, ClientID, pkC, iss_orig) @ #k
985               & #k < #j)"
986
```

```
//
    ================================================================================
// F: AS FLOW
//
    ================================================================================

lemma F1_resource_access_requires_token_exchange:
    all-traces
    "All params #i.
       AS_Request_Accepted(<'resource_access', params>) @ #i
       ==> (Ex code #j.
              AS_Request_Accepted(<'token_exchange', code>) @ #j
              & #j < #i)"

lemma F2_token_exchange_requires_authorization:
    all-traces
    "All code #i.
       AS_Request_Accepted(<'token_exchange', code>) @ #i
       ==> (Ex nonce #j.
              AS_Request_Accepted(<'authorization', nonce>) @ #j
              & #j < #i)"

lemma F3_authorization_requires_challenge:
    all-traces
    "All nonce #i.
       AS_Request_Accepted(<'authorization', nonce>) @ #i
       ==> (Ex #j.
              AS_Request_Accepted(<'challenge', nonce>) @ #j
              & #j < #i)"

//
    ================================================================================
// FIDO-SPECIFIC
//
    ================================================================================

lemma FIDO1_authcode_after_init:
    all-traces
    "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
    iss) @ #i
     ==> (Ex rpId #j. FIDOAuth_Initiated(ClientID, rpId) @ #j & #j < #i)"

lemma FIDO2_complete_after_init:
    all-traces
    "All ClientID #i. AS_Auth_Completed(ClientID) @ #i
     ==> (Ex rpId #j. FIDOAuth_Initiated(ClientID, rpId) @ #j & #j < #i)"


lemma FIDO3_fido_key_secrecy:
```

```
    all-traces
    "All ClientID k #i. Secret_FIDOKey(ClientID, k) @ #i
     ==> not (Ex #j. K(k) @ #j)
         | (Ex #r. RevealFIDO(ClientID) @ #r)"

lemma FIDO4_credential_required_for_auth:
    all-traces
    "All ClientID rpId #i. FIDO_Auth_Success(ClientID, rpId) @ #i
     ==> (Ex skFIDO #j. Secret_FIDOKey(ClientID, skFIDO) @ #j)
         | (Ex #j. RevealFIDO(ClientID) @ #j)"

lemma FIDO5_origin_binding:
    all-traces
    "All ClientID rpId #i. FIDO_Auth_Success(ClientID, rpId) @ #i
     ==> (Ex skFIDO #j. Secret_FIDOKey(ClientID, skFIDO) @ #j & #j < #i)
         | (Ex #j. RevealFIDO(ClientID) @ #j)"

lemma FIDO6_auth_binds_to_attested_client:
    all-traces
    "All ClientID rpId #i. FIDO_Auth_Success(ClientID, rpId) @ #i
     ==> (Ex pkC #j. AS_Accepted_Client_Key(ClientID, pkC) @ #j & #j <
    #i)"

lemma FIDO7_credential_rpId_scoping:
    all-traces
    "All ClientID pkFIDO rpIdReg rpIdUse nonce #i #j.
       FIDO_Credential_Registered(ClientID, pkFIDO, rpIdReg) @ #i
       & FIDO_Authenticator_Signed(ClientID, rpIdUse, nonce) @ #j
       ==> rpIdReg = rpIdUse
           | (Ex #r. RevealFIDO(ClientID) @ #r)"

lemma FIDO8_as_enforces_rpId_binding:
    all-traces
    "All ClientID rpId #i.
       FIDO_Auth_Success(ClientID, rpId) @ #i
       ==> (Ex pkFIDO #j.
              FIDO_Credential_Registered(ClientID, pkFIDO, rpId) @ #j
              & #j < #i)
           | (Ex #r. RevealFIDO(ClientID) @ #r)"

lemma FIDO9_fido_nonce_single_use:
    all-traces
    "All nonce ClientID1 ClientID2 rpId1 rpId2 #i #j.
       AS_Issued_FIDONonce(nonce, ClientID1, rpId1) @ #i
       & AS_Issued_FIDONonce(nonce, ClientID2, rpId2) @ #j
       ==> #i = #j & ClientID1 = ClientID2 & rpId1 = rpId2"

end
```

B.3 ROPC

```
    theory mAuth_ROPC
begin

builtins: signing
heuristic: S


rule ChanOut_S:
    [ Out_S($A, $B, x) ]
    --[ ChanOut_S($A, $B, x) ]->
    [ Sec($A, $B, x) ]

rule ChanIn_S:
    [ Sec($A, $B, x) ]
    --[ ChanIn_S($A, $B, x) ]->
    [ In_S($A, $B, x) ]

//
    ================================================================================
// SETUP RULES
//
    ================================================================================


rule Setup_AS_Identity[role="AS"]:
    [ Fr(~iss) ]
    -->
    [ !AS_Identity(~iss) ]

rule Setup_Resource_Server[role="RS"]:
    [ Fr(~data) ]
    -->
    [ !RS_Resource(~data) ]

rule Setup_Backend_Identity[role="Backend"]:
    [ Fr(~backend_sk) ]
    --[ Secret_BackendKey(~backend_sk) ]->
    [ !Backend_Ltk(~backend_sk),
      !PkBackend(pk(~backend_sk)),
      !AS_Knows_Backend(pk(~backend_sk)) ]

rule Setup_User_Credentials:
    [ Fr(~password) ]
    --[ Secret_UserPassword(~password) ]->
    [ !UserPassword(~password),
      !AS_StoredCredentials(~password) ]

```

```
46
47  //
        ================================================================================
48  // ATTESTATION CHALLENGE (Steps 2-3)
49  //
        ================================================================================
50
51  rule Client_AttestChallReq_ToBackend_Step2[role="Client"]:
52      [ Fr(~ClientID) ]
53      -->
54      [ Out_S($Client, $Backend, <'step2_c2b', ~ClientID>),
55      Client_KeygenSlot(~ClientID) ]
56
57  rule Backend_AttestChallReq_ToAS_Step2[role="Backend"]:
58      [ In_S($Client, $Backend, <'step2_c2b', ClientID>) ]
59      -->
60      [ Out_S($Backend, $AS, <'step2_b2as', ClientID>) ]
61
62  rule AS_IssueChallenge_ToBackend_Step3[role="AS"]:
63      [ In_S($Backend, $AS, <'step2_b2as', ClientID>),
64        Fr(~nonce),
65        !AS_Identity(iss) ]
66      --[ AS_Issued_Nonce(~nonce),
67          AS_Request_Accepted(<'challenge', ~nonce>) ]->
68      [ Out_S($AS, $Backend, <'step3_as2b', ClientID, ~nonce, iss>),
69        AS_Pending_Challenge_Nonce_Useable(ClientID, ~nonce, iss) ]
70
71  rule Backend_IssueChallenge_ToClient_Step3[role="Backend"]:
72      [ In_S($AS, $Backend, <'step3_as2b', ClientID, nonce, iss>) ]
73      -->
74      [ Out_S($Backend, $Client, <'step3_b2c', ClientID, nonce, iss>) ]
75
76  //
        ================================================================================
77  // KEY GENERATION & ATTESTATION (Steps 4-7)
78  //
        ================================================================================
79
80  rule Client_GenerateKey_ToClient_Step4[role="Client"]:
81      [ In_S($Backend, $Client, <'step3_b2c', ClientID, nonce, iss>),
82        Fr(~skC) ]
83      --[ Secret_ClientKey(ClientID, ~skC),
84          Client_Key_In_Hardware(pk(~skC)),
85          Client_Learned_Iss(ClientID, iss) ]->
86      [ Client_GenerateKey_ToClient_Step4(ClientID, nonce),
87        !Client_Key(ClientID, ~skC, pk(~skC)),
88        !Client_Expected_Iss(ClientID, iss) ]
89
90  rule Client_RequestAttestation_ToAA_Step5[role="Client"]:
```

```
91      [ Client_GenerateKey_ToClient_Step4(ClientID, nonce),
92        !Client_Key(ClientID, skC, pkC),
93        Client_KeygenSlot(ClientID),
94        Fr(~Device_Info) ]
95      --[ Client_Sent_Attestation_Request(ClientID, nonce, pkC) ]->
96      [ Client_RequestAttestation_ToAA_Step5(<~Device_Info, nonce, pkC>),
97        Client_AttestRequestPending_ToClient_Step5To8(ClientID, nonce,
    ~Device_Info),
98        Client_ClientIDForKeygen_ToAA(ClientID) ]
99
100 rule Setup_AA_Enclave[role="AA"]:
101     [ Client_ClientIDForKeygen_ToAA(ClientID),
102       Fr(~aa_sk) ]
103     --[ Secret_AAKey(~aa_sk) ]->
104     [ !AA_Ltk(~aa_sk),
105       !PkAA(ClientID, pk(~aa_sk))]
106
107 rule AA_SignAttestation_ToAA_Step6[role="AA"]:
108     [ Client_RequestAttestation_ToAA_Step5(<Device_Info, nonce, pkC>),
109       !AA_Ltk(aa_sk) ]
110     --[ AA_Signed_Attestation(nonce, pkC),
111         AA_NonceUsed(nonce) ]->
112     [ AA_SignAttestation_ToAA_Step6(sign(<Device_Info, nonce, pkC>,
    aa_sk)) ]
113
114 rule AA_SendAttestation_ToClient_Step7[role="AA"]:
115     [ AA_SignAttestation_ToAA_Step6(attestJWT) ]
116     -->
117     [ AA_SendAttestation_ToClient_Step7(attestJWT) ]
118
119 rule Client_ForwardPlatformAttest_ToBackend_Step7b[role="Client"]:
120     [ AA_SendAttestation_ToClient_Step7(platformAttest),
121       Client_AttestRequestPending_ToClient_Step5To8(ClientID, nonce,
    Device_Info),
122       !Client_Key(ClientID, skC, pkC) ]
123     -->
124     [ Out_S($Client, $Backend, <'wrap_cat_c2b', ClientID, nonce, pkC,
    Device_Info, platformAttest>),
125       Client_AwaitingCAT_ToClient(ClientID, nonce, Device_Info,
    platformAttest, pkC) ]
126
127 rule Backend_WrapCAT_ToClient_Step7c[role="Backend"]:
128     [ In_S($Client, $Backend, <'wrap_cat_c2b', ClientID, nonce, pkC,
    Device_Info, platformAttest>),
129       !Backend_Ltk(backend_sk) ]
130     --[ Backend_Issued_CAT(ClientID, pkC, nonce) ]->
131     [ Out_S($Backend, $Client,
132         <'cat_response_b2c', ClientID,
```

```
133             sign(<'client-attestation+jwt', ClientID, pkC, nonce,
    platformAttest>, backend_sk),
134             platformAttest>) ]
135
136 //
    ================================================================================
137 // DPoP CREATION & AUTHORIZATION REQUEST (Steps 8-10)
138 //
    ================================================================================
139 rule Client_CreateDPoPP_ToClient_Step8[role="Client"]:
140     [ In_S($Backend, $Client, <'cat_response_b2c', ClientID,
    clientAttestJWT, platformAttest>),
141       Client_AwaitingCAT_ToClient(ClientID, nonce, Device_Info,
    platformAttest, pkC),
142       !Client_Key(ClientID, skC, pkC),
143       Fr(~DPoP_Params) ]
144     --[ Client_Created_DPoP(pkC, ~DPoP_Params,'authz') ]->
145     [ Client_DPoPAttestation_ToClient_Step8(ClientID, nonce,
    clientAttestJWT, platformAttest, Device_Info),
146       Client_CreateDPoPP_ToClient_Step8(
147           sign(<~DPoP_Params, nonce>, skC), pkC, ~DPoP_Params, nonce) ]
148
149 rule Client_SendAuthzReq_ToBackend_Step9[role="Client"]:
150     [ Client_DPoPAttestation_ToClient_Step8(ClientID, nonce,
    clientAttestJWT, platformAttest, Device_Info),
151       Client_CreateDPoPP_ToClient_Step8(dpopp, pkC, DPoP_Params, nonce) ]
152     -->
153     [ Out_S($Client, $Backend,
154         <'step9_c2b', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
155          dpopp, pkC, DPoP_Params, pkC>) ]
156
157 rule Backend_SendAuthzReq_ToAS_Step9[role="Backend"]:
158     [ In_S($Client, $Backend,
159         <'step9_c2b', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
160          dpopp, pkC, DPoP_Params, dpop_jkt>) ]
161     -->
162     [ Out_S($Backend, $AS,
163         <'step9_b2as', ClientID, nonce, Device_Info, platformAttest,
    clientAttestJWT,
164          dpopp, pkC, DPoP_Params, dpop_jkt>) ]
165
166 rule AS_VerifyJWTAndDPoPP_ToBackend_Step10[role="AS"]:
167     [ In_S($Backend, $AS, <'step9_b2as', ClientID, nonce, Device_Info,
    platformAttest, clientAttestJWT, dpopp, pkC, DPoP_Params, dpop_jkt>),
168       AS_Pending_Challenge_Nonce_Useable(ClientID, nonce, iss),
169       !PkAA(ClientID, pkAA),
170       !AS_Knows_Backend(pkBackend),
```

```
      !AS_Identity(iss) ]
    --[ Eq(verify(platformAttest, <Device_Info, nonce, pkC>, pkAA), true),
        Eq(verify(dpopp, <DPoP_Params,nonce>, pkC), true),
        Eq(verify(clientAttestJWT, <'client-attestation+jwt', ClientID,
    pkC, nonce, platformAttest>, pkBackend), true),
        Eq(dpop_jkt, pkC),
        AS_Request_Accepted(<'authorization', nonce>),
        AS_Accepted_Client_Key(ClientID, pkC),
        AS_Verified_CAT(ClientID, pkC),
        AS_Verified_PlatformAttest(ClientID, pkC),
        AS_Verified_dpop_jkt(ClientID, pkC),
        DPoP_Used(DPoP_Params),
        DPoP_Verified(DPoP_Params, pkC) ]->
    [ Out_S($AS, $Backend, <'ropc_start_as2b', ClientID, iss>),
      !AS_Client_Tied_PublicKey(ClientID, pkC, iss) ]

//
    ================================================================================
// INTEGRATED ROPC AUTHENTICATION (Step 11)
//
    ================================================================================

rule Backend_ROPCStart_ToClient_ROPCStep1[role="Backend"]:
    [ In_S($AS, $Backend,<'ropc_start_as2b', ClientID, iss>)]
    -->
    [ Out_S($Backend,$Client,<'ropc_start_b2c', ClientID, iss>) ]

rule Client_ROPCAuth_ROPCStep2_3[role="Client"]:
    [ In_S($Backend,$Client,<'ropc_start_b2c', ClientID, iss>),
      !Client_Expected_Iss(ClientID, iss),
      !Client_Key(ClientID, skC, pkC),
      !UserPassword(password),
      Fr(~DPoP_ROPC_Params) ]
    --[ Client_Created_DPoP(pkC, ~DPoP_ROPC_Params,'ropc'),
        ROPC_Credentials_Used(ClientID, password),
        ROPCAuth_Initiated(ClientID) ]->
    [ Out_S($Client,$Backend,<'ropc_auth_c2b',
                sign(<~DPoP_ROPC_Params, password>, skC),
                pkC, ~DPoP_ROPC_Params, password, ClientID>) ]

rule Backend_ROPCAuth_ToAS_ROPCStep4[role="Backend"]:
    [ In_S($Client,$Backend,<'ropc_auth_c2b', dpopp, pkC, DPoP_Params,
    password, ClientID>) ]
    -->
    [ Out_S($Backend,$AS,<'ropc_auth_b2as', dpopp, pkC, DPoP_Params,
    password, ClientID>) ]

rule AS_VerifyROPC_SendCode_ROPCStep5_6[role="AS"]:
```

```
214     [ In_S($Backend,$AS,<'ropc_auth_b2as', dpopp, pkC, DPoP_Params,
    password, ClientID>),
215       !AS_StoredCredentials(password),
216       !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
217       Fr(~authorization_code) ]
218     --[ Eq(verify(dpopp, <DPoP_Params, password>, pkC), true),
219         DPoP_Used(DPoP_Params),
220         DPoP_Verified(DPoP_Params, pkC),
221         ROPC_Auth_Success(ClientID),
222         AS_Request_Accepted(<'ropc_auth', DPoP_Params>),
223         AS_Auth_Completed(ClientID),
224         AuthzCode_Issued(~authorization_code, ClientID, pkC, iss) ]->
225     [ Out_S($AS,$Backend,<'auth_completed_as2b', ClientID,
    ~authorization_code>),
226       AuthzCode(~authorization_code, ClientID, pkC, iss) ]
227
228 //
    ================================================================================
229 // TOKEN EXCHANGE & RESOURCE ACCESS (Steps 12-19)
230 //
    ================================================================================
231
232 rule Backend_RequestDPoP_ForTokenExchange[role="Backend"]:
233     [ In_S($AS,$Backend, <'auth_completed_as2b', ClientID,
    authorization_code>) ]
234     -->
235     [ Out_S($Backend, $Client, <'dpop_needed_for_token_b2c', ClientID>),
236       Backend_PendingTokenExchange(ClientID, authorization_code) ]
237
238 rule Client_CreateDPoP_ForTokenExchange[role="Client"]:
239     [ In_S($Backend, $Client, <'dpop_needed_for_token_b2c', ClientID>),
240       !Client_Key(ClientID, skC, pkC),
241       Fr(~DPoP_Params_TE) ]
242     --[ Client_Created_DPoP(pkC, ~DPoP_Params_TE,'token_te') ]->
243     [ Out_S($Client, $Backend, <'dpop_for_token_c2b',
244             sign(<~DPoP_Params_TE,'token_endpoint'>, skC), pkC,
    ~DPoP_Params_TE, ClientID>) ]
245
246 rule Backend_TokenRequest_ToAS_Step12[role="Backend"]:
247     [ In_S($Client, $Backend, <'dpop_for_token_c2b', dpopp_te, pkC,
    DPoP_Params_TE, ClientID>),
248       Backend_PendingTokenExchange(ClientID, authorization_code) ]
249     -->
250     [ Out_S($Backend, $AS, <'token_req_b2as', ClientID,
    authorization_code,
251             dpopp_te, pkC, DPoP_Params_TE>) ]
252
253 rule AS_TokenAns_ToBackend_Step13[role="AS"]:
254     [ In_S($Backend, $AS, <'token_req_b2as', ClientID, authorization_code,
```

```
255             dpopp_te, pkC, DPoP_Params_TE>),
256       AuthzCode(authorization_code, ClientID, pkC, iss),
257       !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
258       Fr(~constrained_access_token),
259       Fr(~DPoP_reference_token),
260       Fr(~refresh_token),
261       !AS_Identity(iss) ]
262     --[ Eq(verify(dpopp_te, <DPoP_Params_TE,'token_endpoint'>, pkC),
    true),
263         DPoP_Used(DPoP_Params_TE),
264         DPoP_Verified(DPoP_Params_TE, pkC),
265         AS_Request_Accepted(<'token_exchange', authorization_code>),
266         AuthzCode_Redeemed(authorization_code, ClientID, pkC),
267         Token_Issued(~constrained_access_token, ClientID, pkC),
268         RefToken_Issued(~DPoP_reference_token, ClientID, pkC),
269         RefreshToken_Issued(~refresh_token, ClientID, pkC),
270         Token_Issued_By_Iss(ClientID, iss) ]->
271     [ !ConstrainedAccess(~constrained_access_token, pkC),
272       !ConstrainedReference(~DPoP_reference_token, pkC, iss),
273       !ConstrainedRefresh(~refresh_token, pkC),
274       Out_S($AS, $Backend, <'token_resp_as2b', ClientID,
    ~DPoP_reference_token,
275              ~constrained_access_token, pkC, iss, ~refresh_token>) ]
276
277 rule Backend_SendReferenceToken_ToClient_Step14[role="Backend"]:
278     [ In_S($AS, $Backend, <'token_resp_as2b', ClientID,
    DPoP_reference_token,
279             constrained_access_token, pkC, iss, refresh_token>) ]
280     -->
281     [ Out_S($Backend, $Client, <'ref_token_b2c', ClientID,
    DPoP_reference_token, iss>),
282       Backend_AccessTokenHeld(DPoP_reference_token,
    constrained_access_token, pkC, ClientID),
283       Backend_RefreshTokenHeld(refresh_token, pkC, ClientID) ]
284
285 rule Client_GenerateDPoP_Step15_SendDPoP_ToBackend_Step16[role="Client"]:
286     [ In_S($Backend, $Client, <'ref_token_b2c', ClientID,
    DPoP_reference_token, iss>),
287       !Client_Key(ClientID, skC, pkC),
288       !Client_Expected_Iss(ClientID, iss),
289       Fr(~DPoP_Params2) ]
290     --[ Client_Verified_Iss(ClientID, iss),
291         Client_Created_DPoP(pkC, ~DPoP_Params2,'resource') ]->
292     [ Out_S($Client, $Backend, <'dpop_resource_c2b',
293             sign(<~DPoP_Params2, DPoP_reference_token>, skC),
294             pkC, ~DPoP_Params2, DPoP_reference_token>),
295       Client_CanRefresh(ClientID) ]
296
297 rule Backend_ResourceRequest_ToRS_Step17[role="Backend"]:
```

```
298     [ In_S($Client,$Backend, <'dpop_resource_c2b', dpopp, pkC,
    DPoP_Params2, DPoP_reference_token>),
299       Backend_AccessTokenHeld(DPoP_reference_token,
    constrained_access_token, pkC, ClientID) ]
300     -->
301     [ Out_S($Backend, $RS, <'rs_verify_b2rs', constrained_access_token,
    dpopp, pkC,
302             DPoP_Params2, DPoP_reference_token, ClientID>) ]
303
304 rule RS_ForwardIntrospection_ToAS[role="RS API"]:
305     [ In_S($Backend, $RS, <'rs_verify_b2rs', constrained_access_token,
    dpopp, pkC,
306            DPoP_Params2, DPoP_reference_token, ClientID>)]
307     -->
308     [ Out_S($RS, $AS, <'introspect', constrained_access_token, dpopp, pkC,
309             DPoP_Params2, DPoP_reference_token, ClientID>) ,
310             RS_Client_Tied_Access(ClientID,constrained_access_token)]
311
312 rule AS_VerifyDPoPPAndTokens_Step18[role="AS"]:
313     [ In_S($RS, $AS, <'introspect', constrained_access_token, dpopp, pkC,
314            DPoP_Params2, DPoP_reference_token, ClientID>),
315       !ConstrainedAccess(constrained_access_token, pkC),
316       !ConstrainedReference(DPoP_reference_token, pkC, iss),
317       !AS_Client_Tied_PublicKey(ClientID, pkC, iss) ,
318       !AS_Identity(iss) ]
319     --[ Eq(verify(dpopp, <DPoP_Params2, DPoP_reference_token>, pkC),
    true),
320         DPoP_Used(DPoP_Params2),
321         DPoP_Verified(DPoP_Params2, pkC),
322         AS_Request_Accepted(<'resource_access', DPoP_Params2>),
323         Token_Used_For_Access(constrained_access_token, ClientID, pkC) ]->
324     [ Out_S($AS, $RS, <'verified', ClientID, constrained_access_token>) ]
325
326 rule RS_SendResource_Step19[role="RS API"]:
327     [ In_S($AS, $RS, <'verified', ClientID, constrained_access_token>),
328       !RS_Resource(data),
329       RS_Client_Tied_Access(ClientID, constrained_access_token) ]
330     --[ RS_Sent_Resource(data, ClientID, constrained_access_token),
331         RS_Access_Granted(data, ClientID) ]->
332     [ Out_S($RS, $Client, <'resource_sent', data, ClientID>) ]
333
334 rule Client_Received_Data[role="Client"]:
335    [In_S($RS,$Client,<'resource_sent',data,ClientID>)]
336    --[Client_Recourse_Received(data,ClientID)]->
337    []
338
339
340 //
    ================================================================================
```

```
// REFRESH TOKEN EXCHANGE
//
    ================================================================================

rule Client_RequestRefresh[role="Client"]:
    [ Client_CanRefresh(ClientID),
      !Client_Key(ClientID, skC, pkC),
      Fr(~DPoP_Params_Refresh) ]
    --[ Client_Created_DPoP(pkC, ~DPoP_Params_Refresh,'refresh'),
        Client_Requested_Refresh(ClientID) ]->
    [ Out_S($Client, $Backend, <'refresh_req_c2b',
            sign(<~DPoP_Params_Refresh, 'refresh'>, skC),
            pkC, ~DPoP_Params_Refresh, ClientID>) ]

rule Backend_RefreshRequest_ToAS[role="Backend"]:
    [ In_S($Client, $Backend, <'refresh_req_c2b', dpopp, pkC,
    DPoP_Params_R, ClientID>),
      Backend_RefreshTokenHeld(refresh_token, pkC, ClientID) ]
    -->
    [ Out_S($Backend, $AS, <'refresh_req_b2as', refresh_token, dpopp,
    pkC, DPoP_Params_R, ClientID>) ]

rule AS_VerifyRefresh_IssueToken[role="AS"]:
    [ In_S($Backend, $AS, <'refresh_req_b2as', refresh_token, dpopp, pkC,
    DPoP_Params_R, ClientID>),
      !ConstrainedRefresh(refresh_token, pkC),
      !AS_Client_Tied_PublicKey(ClientID, pkC, iss),
      !AS_Identity(iss)  ]
    --[ Eq(verify(dpopp, <DPoP_Params_R,'refresh'>, pkC), true),
        DPoP_Used(DPoP_Params_R),
        DPoP_Verified(DPoP_Params_R, pkC),
        AS_Request_Accepted(<'refresh', DPoP_Params_R>),
        RefreshToken_Used(refresh_token, ClientID, pkC) ]->
    [ AS_RefreshCompleted(ClientID) ]

//
    ================================================================================
// LEAK / ATTACKER EVENTS
//
    ================================================================================

rule Leak_Access_Token:
    [ !ConstrainedAccess(token, pkC) ]
    --[ LeakToken(token, pkC) ]->
    [ Out(token) ]

rule Leak_Reference_Token:
    [ !ConstrainedReference(ref_token, pkC, iss) ]
    --[ LeakRefToken(ref_token, pkC) ]->
```

```
384     [ Out(ref_token) ]
385
386 rule Leak_Refresh_Token:
387     [ !ConstrainedRefresh(refresh_token, pkC) ]
388     --[ LeakRefreshToken(refresh_token, pkC) ]->
389     [ Out(refresh_token) ]
390
391 rule Reveal_AA_Key:
392     [ !AA_Ltk(aa_sk) ]
393     --[ RevealAA(aa_sk) ]->
394     [ Out(aa_sk) ]
395
396 rule Reveal_Client_Key:
397     [ !Client_Key(ClientID, skC, pkC) ]
398     --[ RevealClientKey(ClientID, skC) ]->
399     [ Out(skC) ]
400
401 rule Reveal_Backend_Key:
402     [ !Backend_Ltk(backend_sk) ]
403     --[ RevealBackend(backend_sk) ]->
404     [ Out(backend_sk) ]
405
406 rule Reveal_User_Password:
407     [ !UserPassword(password) ]
408     --[ RevealPassword(password) ]->
409     [ Out(password) ]
410
411 //
    ================================================================================
412 // RESTRICTIONS
413 //
    ================================================================================
414
415 restriction Equality:
416     "All x y #i. Eq(x,y) @ #i ==> x = y"
417
418 restriction unique_dpopp:
419     "All params #i #j.
420        DPoP_Used(params) @ #i & DPoP_Used(params) @ #j
421        ==> #i = #j"
422
423 restriction no_replay:
424     "All id #i #j.
425        AS_Request_Accepted(id) @ #i & AS_Request_Accepted(id) @ #j
426        ==> #i = #j"
427
428 restriction unique_attestation_per_nonce:
429     "All nonce pkC1 pkC2 #i #j.
430        AA_Signed_Attestation(nonce, pkC1) @ #i
```

```
       & AA_Signed_Attestation(nonce, pkC2) @ #j
       ==> #i = #j"

//
    ================================================================================
// SOURCES
//
    ================================================================================


lemma typing [sources]:
  " (All t cid pk #i #j.
       Token_Issued(t, cid, pk) @ #j & !KU(t) @ #i
       ==> (Ex #l. LeakToken(t, pk) @ #l & #l < #i))
  & (All t cid pk #i #j.
       RefToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
       ==> (Ex #l. LeakRefToken(t, pk) @ #l & #l < #i))
  & (All t cid pk #i #j.
       RefreshToken_Issued(t, cid, pk) @ #j & !KU(t) @ #i
       ==> (Ex #l. LeakRefreshToken(t, pk) @ #l & #l < #i))
  & (All c cid pk iss #i #j.
       AuthzCode_Issued(c, cid, pk, iss) @ #j & !KU(c) @ #i
       ==> F)
  "

lemma token_implies_code [reuse]:
  "All t cid pk #i. Token_Issued(t, cid, pk) @ #i
   ==> (Ex c iss #j. AuthzCode_Issued(c, cid, pk, iss) @ #j & #j < #i)"

lemma code_implies_accept [reuse]:
  "All c cid pk iss #i. AuthzCode_Issued(c, cid, pk, iss) @ #i
   ==> (Ex #j. AS_Accepted_Client_Key(cid, pk) @ #j & #j < #i)"

lemma access_implies_issued [reuse, use_induction]:
  "All t cid pk #i. Token_Used_For_Access(t, cid, pk) @ #i
   ==> (Ex #j. Token_Issued(t, cid, pk) @ #j & #j < #i)"

lemma refresh_used_implies_issued [reuse]:
  "All t cid pk #i. RefreshToken_Used(t, cid, pk) @ #i
   ==> (Ex #j. RefreshToken_Issued(t, cid, pk) @ #j & #j < #i)"

lemma accept_unique_pkC [reuse, use_induction]:
  "All cid pk1 pk2 #i #j.
      AS_Accepted_Client_Key(cid, pk1) @ #i
    & AS_Accepted_Client_Key(cid, pk2) @ #j
    ==> pk1 = pk2"

lemma client_iss_unique [reuse, use_induction]:
  "All cid iss1 iss2 #i #j.
```

```
      Client_Learned_Iss(cid, iss1) @ #i
    & Client_Learned_Iss(cid, iss2) @ #j
    ==> iss1 = iss2 & #i = #j"

//
    ================================================================================
// UNIQUE PROOFS
//
    ================================================================================

lemma L_unique_dpopp_proof:
    all-traces
    "All params #i #j.
       DPoP_Used(params) @ #i & DPoP_Used(params) @ #j
       ==> #i = #j"

lemma L_unique_attestation_proof:
    all-traces
    "All nonce pkC1 pkC2 #i #j.
       AA_Signed_Attestation(nonce, pkC1) @ #i
       & AA_Signed_Attestation(nonce, pkC2) @ #j
       ==> #i = #j & pkC1 = pkC2"


//
    ================================================================================
// SANITY
//
    ================================================================================

lemma SANITY_keygen[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key,
   hide_lemma=accept_unique_pkC,
   hide_lemma=client_iss_unique,
   hide_lemma=token_implies_code,
   hide_lemma=code_implies_accept,
   hide_lemma=access_implies_issued,
   hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex cid pk #t. Client_Learned_Iss(cid, pk) @ #t"

lemma SANITY_token_issued[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key,
   hide_lemma=accept_unique_pkC,
```

```
    hide_lemma=client_iss_unique,
    hide_lemma=token_implies_code,
    hide_lemma=code_implies_accept,
    hide_lemma=access_implies_issued,
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex t cid pk #i. Token_Issued(t, cid, pk) @ #i"

lemma SANITY_dpop_verified[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
    hide_lemma=B3_token_implies_attestation,
    hide_lemma=I1_token_requires_hardware_key,
    hide_lemma=accept_unique_pkC,
    hide_lemma=client_iss_unique,
    hide_lemma=token_implies_code,
    hide_lemma=code_implies_accept,
    hide_lemma=access_implies_issued,
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex p pk #i. DPoP_Verified(p, pk) @ #i"

lemma SANITY_token_used[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
    hide_lemma=B3_token_implies_attestation,
    hide_lemma=I1_token_requires_hardware_key,
    hide_lemma=accept_unique_pkC,
    hide_lemma=client_iss_unique,
    hide_lemma=token_implies_code,
    hide_lemma=code_implies_accept,
    hide_lemma=access_implies_issued,
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex t cid pk #i. Token_Used_For_Access(t, cid, pk) @ #i"

lemma SANITY_refresh_used[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
    hide_lemma=B3_token_implies_attestation,
    hide_lemma=I1_token_requires_hardware_key,
    hide_lemma=accept_unique_pkC,
    hide_lemma=client_iss_unique,
    hide_lemma=token_implies_code,
    hide_lemma=code_implies_accept,
    hide_lemma=access_implies_issued,
    hide_lemma=refresh_used_implies_issued]: exists-trace
  "Ex t cid pk #i. RefreshToken_Used(t, cid, pk) @ #i"

lemma SANITY_resource_received[hide_lemma=R2_dpop_requires_creation,
    hide_lemma=B1_attestation_unforgeable,
    hide_lemma=B2_as_requires_attestation,
```

```
572     hide_lemma=B3_token_implies_attestation,
573     hide_lemma=I1_token_requires_hardware_key,
574     hide_lemma=accept_unique_pkC,
575     hide_lemma=client_iss_unique,
576     hide_lemma=token_implies_code,
577     hide_lemma=code_implies_accept,
578     hide_lemma=access_implies_issued,
579     hide_lemma=refresh_used_implies_issued]: exists-trace
580   "Ex data cid #t. Client_Recourse_Received(data, cid) @ #t"
581
582 lemma SANITY_full_chain[hide_lemma=R2_dpop_requires_creation,
583     hide_lemma=B1_attestation_unforgeable,
584     hide_lemma=B2_as_requires_attestation,
585     hide_lemma=B3_token_implies_attestation,
586     hide_lemma=I1_token_requires_hardware_key,
587     hide_lemma=accept_unique_pkC,
588     hide_lemma=client_iss_unique,
589     hide_lemma=token_implies_code,
590     hide_lemma=code_implies_accept,
591     hide_lemma=access_implies_issued,
592     hide_lemma=refresh_used_implies_issued]:
593      exists-trace
594      "Ex data ClientID nonce pkC code token iss
595          #t1 #t2 #t3 #t4 #t5 #t6 #t7 #t8 #t9 #t10 .
596         AS_Issued_Nonce(nonce)                                  @ #t1
597       & Client_Sent_Attestation_Request(ClientID, nonce, pkC) @ #t2
598       & AA_Signed_Attestation(nonce, pkC)                       @ #t3
599       & Backend_Issued_CAT(ClientID, pkC, nonce)              @ #t4
600       & AS_Accepted_Client_Key(ClientID, pkC)                 @ #t5
601       & ROPCAuth_Initiated(ClientID)                          @ #t6
602       & AuthzCode_Issued(code, ClientID, pkC, iss)            @ #t7
603       & ROPC_Auth_Success(ClientID)                           @ #t7
604       & Token_Issued(token, ClientID, pkC)                    @ #t8
605       & Token_Used_For_Access(token, ClientID, pkC)           @ #t9
606       & Client_Recourse_Received(data, ClientID)              @ #t10
607       & #t1 < #t2 & #t2 < #t3 & #t3 < #t4 & #t4 < #t5
608       & #t5 < #t6 & #t6 < #t7 & #t7 < #t8 & #t8 < #t9
609       & #t9 < #t10 "
610
611 //
      ================================================================================
612 // S: SECRET KEY SECRECY
613 //
      ================================================================================
614
615 lemma S1_client_key_secrecy:
616     all-traces
617     "All ClientID k #i. Secret_ClientKey(ClientID, k) @ #i
618       ==> not (Ex #j. K(k) @ #j)
```

```
619         | (Ex #r. RevealClientKey(ClientID, k) @ #r)"
620
621 lemma S2_access_token_secrecy:
622     all-traces
623     "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
624      ==> not (Ex #j. K(token) @ #j)
625         | (Ex #l. LeakToken(token, pkC) @ #l)"
626
627 lemma S3_reference_token_secrecy:
628     all-traces
629     "All token ClientID pkC #i. RefToken_Issued(token, ClientID, pkC) @ #i
630      ==> not (Ex #j. K(token) @ #j)
631         | (Ex #l. LeakRefToken(token, pkC) @ #l)"
632
633 lemma S4_aa_key_secrecy:
634     all-traces
635     "All k #i. Secret_AAKey(k) @ #i
636      ==> not (Ex #j. K(k) @ #j)
637         | (Ex #r. RevealAA(k) @ #r)"
638
639 lemma S5_refresh_token_secrecy:
640     all-traces
641     "All token ClientID pkC #i. RefreshToken_Issued(token, ClientID, pkC)
    @ #i
642      ==> not (Ex #j. K(token) @ #j)
643         | (Ex #l. LeakRefreshToken(token, pkC) @ #l)"
644
645 lemma S6_backend_key_secrecy:
646     all-traces
647     "All k #i. Secret_BackendKey(k) @ #i
648      ==> not (Ex #j. K(k) @ #j)
649         | (Ex #r. RevealBackend(k) @ #r)"
650
651
652 //
    ================================================================================
653 // Z: AUTHORIZATION CORRECTNESS
654 //
    ================================================================================
655
656
657 lemma Z1_code_after_attestation:
658     all-traces
659     "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
    iss) @ #i
660      ==> (Ex nonce #j #k.
661        AS_Issued_Nonce(nonce) @ #j
662        & AA_Signed_Attestation(nonce, pkC) @ #k & #j < #k & #k < #i)
663         | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
```

```
664
665 lemma Z2_no_double_redeem:
666     all-traces
667     "not (Ex code ClientID1 ClientID2 pkC1 pkC2 #i #j.
668             AuthzCode_Redeemed(code, ClientID1, pkC1) @ #i
669             & AuthzCode_Redeemed(code, ClientID2, pkC2) @ #j
670             & not (#i = #j))"
671
672 lemma Z3_no_auth_code_substitution:
673     all-traces
674     "All code ClientID1 ClientID2 pkC1 pkC2 iss #i #j.
675        AuthzCode_Issued(code, ClientID1, pkC1, iss) @ #i
676        & AuthzCode_Redeemed(code, ClientID2, pkC2) @ #j
677        ==> ClientID1 = ClientID2 & pkC1 = pkC2"
678
679
680 lemma Z4_code_unique_issuance:
681     all-traces
682     "All code ClientID1 ClientID2 pkC1 pkC2 iss1 iss2 #i #j.
683        AuthzCode_Issued(code, ClientID1, pkC1, iss1) @ #i
684        & AuthzCode_Issued(code, ClientID2, pkC2, iss2) @ #j
685        ==> #i = #j & ClientID1 = ClientID2 & pkC1 = pkC2 & iss1 = iss2"
686
687
688 //
    ================================================================================
689 // T: TOKEN BINDING / PROOF-OF-POSSESSION
690 //
    ================================================================================
691
692 lemma T1_stolen_token_unusable:
693     all-traces
694     "All token pkC ClientID #i #j.
695        LeakToken(token, pkC) @ #i
696        & Token_Used_For_Access(token, ClientID, pkC) @ #j
697        ==> (Ex params ctx #k #l.
698               Client_Created_DPoP(pkC, params,ctx) @ #k
699               & DPoP_Verified(params, pkC) @ #l
700               & #k < #l & #l < #j)
701            | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
702
703 lemma T2_token_access_requires_pop:
704     all-traces
705     "All token ClientID pkC #i. Token_Used_For_Access(token, ClientID,
    pkC) @ #i
706      ==> (Ex params #j #k.
707             DPoP_Verified(params, pkC) @ #j
708             & AS_Accepted_Client_Key(ClientID, pkC) @ #k
709             & (#k < #j | #k = #j)
```

```
            &  #j < #i)
        | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

lemma T3_no_token_identity_crossing:
    all-traces
    "All token ClientID1 ClientID2 pkC1 pkC2 #i #j.
       Token_Issued(token, ClientID1, pkC1) @ #i
       & Token_Used_For_Access(token, ClientID2, pkC2) @ #j
       ==> ClientID1 = ClientID2 & pkC1 = pkC2"

lemma T4_stolen_refresh_token_unusable:
    all-traces
    "All token pkC ClientID #i #j.
       LeakRefreshToken(token, pkC) @ #i
       & RefreshToken_Used(token, ClientID, pkC) @ #j
       ==> (Ex params ctx #k. Client_Created_DPoP(pkC, params,ctx) @ #k &
   #k < #j)
           | (Ex skC #k. RevealClientKey(ClientID, skC) @ #k)"

lemma T5_stolen_refresh_needs_dpop:
    all-traces
    "All token pkC ClientID #i #j.
       LeakRefreshToken(token, pkC) @ #i
       & RefreshToken_Used(token, ClientID, pkC) @ #j
       ==> (Ex params ctx #k #l.
              Client_Created_DPoP(pkC, params, ctx) @ #k
              & DPoP_Verified(params, pkC) @ #l
              & #k < #l & #l = #j)
           | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

//
    ================================================================================
// R: REPLAY RESISTANCE AND FRESHNESS
//
    ================================================================================

lemma R1_dpop_requires_creation:
    all-traces
    "All params pkC #i. DPoP_Verified(params, pkC) @ #i
     ==> (Ex ctx #j. Client_Created_DPoP(pkC, params, ctx) @ #j & #j < #i)
         | (Ex ClientID skC #r. RevealClientKey(ClientID, skC) @ #r)"

//
    ================================================================================
// B: ATTESTATION BINDING
//
    ================================================================================

lemma B1_attestation_unforgeable[reuse]:
```

```
754     all-traces
755     "All nonce pkC #i. AA_Signed_Attestation(nonce, pkC) @ #i
756      ==> (Ex ClientID #j. Client_Sent_Attestation_Request(ClientID,
    nonce, pkC) @ #j & #j < #i)
757           | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
758
759 lemma B2_as_requires_attestation[reuse]:
760     all-traces
761     "All ClientID pkC #i. AS_Accepted_Client_Key(ClientID, pkC) @ #i
762      ==> (Ex nonce #j. AA_Signed_Attestation(nonce, pkC) @ #j & #j < #i)
763          | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
764
765 lemma B3_token_implies_attestation[reuse]:
766     all-traces
767     "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
768      ==> (Ex nonce #j #k.
769        AA_Signed_Attestation(nonce, pkC) @ #j
770        & AS_Accepted_Client_Key(ClientID, pkC) @ #k
771        & #j < #k & #k < #i)
772         | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
773
774
775 lemma B4_attestation_key_binding:
776     all-traces
777     "All ClientID1 ClientID2 pkC1 pkC2 nonce #i #j #k1 #k2.
778        AS_Accepted_Client_Key(ClientID1, pkC1) @ #i
779        & AS_Accepted_Client_Key(ClientID2, pkC2) @ #j
780        & AA_Signed_Attestation(nonce, pkC1) @ #k1
781        & AA_Signed_Attestation(nonce, pkC2) @ #k2
782        ==> pkC1 = pkC2"
783
784 lemma B5_attestation_nonce_single_use:
785     all-traces
786     "All nonce #i #j.
787        AA_NonceUsed(nonce) @ #i & AA_NonceUsed(nonce) @ #j
788        ==> #i = #j"
789
790
791 lemma B6_cat_unforgeable:
792     all-traces
793     "All ClientID pkC #i.
794        AS_Verified_CAT(ClientID, pkC) @ #i
795        ==> (Ex nonce #j. Backend_Issued_CAT(ClientID, pkC, nonce) @ #j &
    #j < #i)
796             | (Ex backend_sk #r. RevealBackend(backend_sk) @ #r)"
797
798 lemma B7_dpop_jkt_matches_instance_key:
799     all-traces
800     "All ClientID pkC #i.
```

```
801         AS_Verified_dpop_jkt(ClientID, pkC) @ #i
802         ==> (Ex nonce #j.
803                Client_Sent_Attestation_Request(ClientID, nonce, pkC) @ #j
804                & #j < #i)
805             | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
806
807 //
    ==========================================================================
808 // K: KEY CONTINUITY
809 //
    ==========================================================================
810
811
812 lemma K1_key_continuity_attestation_to_token:
813     all-traces
814     "All ClientID pkC_accept token pkC_token #i #j.
815        AS_Accepted_Client_Key(ClientID, pkC_accept) @ #i
816        & Token_Issued(token, ClientID, pkC_token) @ #j
817        & #i < #j
818        ==> pkC_accept = pkC_token
819            | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
820
821 lemma K2_key_continuity_token_to_access:
822     all-traces
823     "All token ClientID pkC1 pkC2 #i #j.
824        Token_Issued(token, ClientID, pkC1) @ #i
825        & Token_Used_For_Access(token, ClientID, pkC2) @ #j
826        ==> pkC1 = pkC2"
827
828
829 lemma K3_dpop_key_matches_accepted:
830     all-traces
831     "All params pkC_dpop ClientID pkC_accept token #i #j #k.
832        DPoP_Verified(params, pkC_dpop) @ #i
833        & AS_Accepted_Client_Key(ClientID, pkC_accept) @ #j
834        & Token_Used_For_Access(token, ClientID, pkC_dpop) @ #k
835        & #j < #i & #i = #k
836        ==> pkC_dpop = pkC_accept
837            | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
838
839
840 //
    ==========================================================================
841 // I: HARDWARE BOUND KEY CONTINUITY
842 //
    ==========================================================================
843
844 lemma I1_token_requires_hardware_key [reuse]:
845   all-traces
```

```
846   "All token ClientID pkC #i.
847     Token_Issued(token, ClientID, pkC) @ #i
848     ==>
849       (Ex #j #k.
850          Client_Key_In_Hardware(pkC) @ #j
851          & AS_Accepted_Client_Key(ClientID, pkC) @ #k
852          & #j < #k & #k < #i)
853       | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)
854       | (Ex aa_sk #r. RevealAA(aa_sk) @ #r)"
855
856 lemma I2_access_requires_hardware_key:
857     all-traces
858     "All token ClientID pkC #i #j.
859        Token_Used_For_Access( token, ClientID, pkC ) @ #i
860        & Token_Issued( token, ClientID, pkC ) @ #j
861        & #j < #i
862        ==> (Ex #k #l.
863               Client_Key_In_Hardware( pkC ) @ #k
864               & AS_Accepted_Client_Key( ClientID, pkC ) @ #l
865               & #k < #l & #l < #j)
866            | (Ex skC #r. RevealClientKey( ClientID, skC ) @ #r)
867            | (Ex aa_sk #r. RevealAA( aa_sk ) @ #r)"
868
869 //
     ================================================================================
870 // E: END-TO-END
871 //
     ================================================================================
872
873 lemma E1_token_requires_authentication:
874     all-traces
875     "All token ClientID pkC #i. Token_Issued(token, ClientID, pkC) @ #i
876      ==> (Ex #j. AS_Auth_Completed(ClientID) @ #j & #j < #i)"
877
878 lemma E2_resource_matches_issued_token:
879     all-traces
880     "All data ClientID #i.
881        Client_Recourse_Received(data, ClientID) @ #i
882        ==> (Ex token pkC #j #k.
883               RS_Sent_Resource(data, ClientID, token) @ #j
884               & Token_Issued(token, ClientID, pkC) @ #k
885               & #k < #j & #j < #i)"
886
887
888 lemma
     E3_resource_requires_fresh_dpop[hide_lemma=B1_attestation_unforgeable,
889    hide_lemma=B2_as_requires_attestation,
890    hide_lemma=B3_token_implies_attestation,
891    hide_lemma=I1_token_requires_hardware_key]:
```

```
    all-traces
    "All data ClientID #i. Client_Recourse_Received(data, ClientID) @ #i
     ==> (Ex pkC ctx params #j #k.
            Client_Created_DPoP(pkC, params, ctx) @ #j
            & DPoP_Verified(params, pkC) @ #k
            & #j < #k & #k < #i
            & (Ex #l. AS_Accepted_Client_Key(ClientID, pkC) @ #l))
         | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"

lemma E4_refresh_requires_issuance:
    all-traces
    "All token ClientID pkC #i.
       RefreshToken_Used(token, ClientID, pkC) @ #i
       ==> (Ex #j.
              RefreshToken_Issued(token, ClientID, pkC) @ #j
              & #j < #i)"

//
    ================================================================================
// ISS: MIX-UP PREVENTION
//
    ================================================================================

lemma ISS1_consistency[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All ClientID iss #i.
       Client_Verified_Iss(ClientID, iss) @ #i
       ==> (Ex #j. Client_Learned_Iss(ClientID, iss) @ #j & #j < #i)"


lemma ISS2_no_mix_up[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
   hide_lemma=B2_as_requires_attestation,
   hide_lemma=B3_token_implies_attestation,
   hide_lemma=I1_token_requires_hardware_key]:
    all-traces
    "All code ClientID pkC iss1 iss2 #i #j.
       AuthzCode_Issued(code, ClientID, pkC, iss1) @ #i
       & AuthzCode_Redeemed(code, ClientID, pkC) @ #j
       & Token_Issued_By_Iss(ClientID, iss2) @ #j
       ==> iss1 = iss2"


lemma ISS3_token_origin[hide_lemma=R2_dpop_requires_creation,
   hide_lemma=B1_attestation_unforgeable,
```

```
939     hide_lemma=B2_as_requires_attestation,
940     hide_lemma=B3_token_implies_attestation,
941     hide_lemma=I1_token_requires_hardware_key]:
942      all-traces
943      "All ClientID iss #i. Token_Issued_By_Iss(ClientID, iss) @ #i
944       ==> (Ex code pkC #j. AuthzCode_Issued(code, ClientID, pkC, iss) @ #j
        & #j < #i)"
945
946 lemma ISS4_client_rejects_wrong_iss[hide_lemma=R2_dpop_requires_creation,
947     hide_lemma=B1_attestation_unforgeable,
948     hide_lemma=B2_as_requires_attestation,
949     hide_lemma=B3_token_implies_attestation,
950     hide_lemma=I1_token_requires_hardware_key]:
951      all-traces
952      "All ClientID iss_learned iss_verified #i #j.
953         Client_Learned_Iss(ClientID, iss_learned) @ #i
954         & Client_Verified_Iss(ClientID, iss_verified) @ #j
955         & #i < #j
956         ==> iss_learned = iss_verified"
957
958 lemma ISS5_refresh_preserves_iss[hide_lemma=R2_dpop_requires_creation,
959     hide_lemma=B1_attestation_unforgeable,
960     hide_lemma=B2_as_requires_attestation,
961     hide_lemma=B3_token_implies_attestation,
962     hide_lemma=I1_token_requires_hardware_key]:
963      all-traces
964      "All ClientID iss_orig token pkC #i #j.
965         Token_Issued_By_Iss(ClientID, iss_orig) @ #i
966         & RefreshToken_Used(token, ClientID, pkC) @ #j
967         & #i < #j
968         ==> (Ex code #k.
969                AuthzCode_Issued(code, ClientID, pkC, iss_orig) @ #k
970                & #k < #j)"
971
972 //
      ==========================================================================
973 // F: AS FLOW
974 //
      ==========================================================================
975
976
977 lemma F1_resource_access_requires_token_exchange:
978      all-traces
979      "All params #i.
980         AS_Request_Accepted(<'resource_access', params>) @ #i
981         ==> (Ex code #j.
982                AS_Request_Accepted(<'token_exchange', code>) @ #j
983                & #j < #i)"
984
```

```
lemma F2_token_exchange_requires_authorization:
    all-traces
    "All code #i.
       AS_Request_Accepted(<'token_exchange', code>) @ #i
       ==> (Ex nonce #j.
              AS_Request_Accepted(<'authorization', nonce>) @ #j
              & #j < #i)"

lemma F3_authorization_requires_challenge:
    all-traces
    "All nonce #i.
       AS_Request_Accepted(<'authorization', nonce>) @ #i
       ==> (Ex #j.
              AS_Request_Accepted(<'challenge', nonce>) @ #j
              & #j < #i)"

//
    ================================================================================
// ROPC-SPECIFIC
//
    ================================================================================

lemma ROPC1_authcode_after_init:
    all-traces
    "All code ClientID pkC iss #i. AuthzCode_Issued(code, ClientID, pkC,
    iss) @ #i
     ==> (Ex #j. ROPCAuth_Initiated(ClientID) @ #j & #j < #i)"

lemma ROPC2_complete_after_init:
    all-traces
    "All ClientID #i. AS_Auth_Completed(ClientID) @ #i
     ==> (Ex #j. ROPCAuth_Initiated(ClientID) @ #j & #j < #i)"

lemma ROPC3_password_secrecy:
    all-traces
    "All pw #i. Secret_UserPassword(pw) @ #i
     ==> not (Ex #j. K(pw) @ #j)
         | (Ex #r. RevealPassword(pw) @ #r)"

lemma ROPC4_auth_requires_credentials:
    all-traces
    "All ClientID #i. ROPC_Auth_Success(ClientID) @ #i
     ==> (Ex pw #j. ROPC_Credentials_Used(ClientID, pw) @ #j & #j < #i)
         | (Ex pw #r. RevealPassword(pw) @ #r)"

lemma ROPC5_ropc_requires_dpop:
    all-traces
    "All ClientID #i. ROPC_Auth_Success(ClientID) @ #i
     ==> (Ex params pkC #j. DPoP_Verified(params, pkC) @ #j
```

```
1031            & (Ex #k. AS_Accepted_Client_Key(ClientID, pkC) @ #k & #k <
    #i))"
1032
1033 lemma ROPC6_auth_binds_to_attested_client:
1034     all-traces
1035     "All ClientID #i. ROPC_Auth_Success(ClientID) @ #i
1036      ==> (Ex pkC #j. AS_Accepted_Client_Key(ClientID, pkC) @ #j & #j <
    #i)"
1037
1038 lemma ROPC7_password_not_leaked_by_protocol:
1039     all-traces
1040     "All ClientID pw #i. ROPC_Credentials_Used(ClientID, pw) @ #i
1041      ==> not (Ex #j. K(pw) @ #j)
1042          | (Ex #r. RevealPassword(pw) @ #r)
1043          | (Ex skC #r. RevealClientKey(ClientID, skC) @ #r)"
1044
1045 lemma ROPC8_dpop_params_bind_password_use:
1046     all-traces
1047     "All ClientID #i.
1048        ROPC_Auth_Success(ClientID) @ #i
1049        ==> (Ex pw params pkC #j.
1050               ROPC_Credentials_Used(ClientID, pw) @ #j
1051               & DPoP_Verified(params, pkC) @ #i
1052               & #j < #i)"
1053
1054 end
```