Formal Specification and Verification of the OTA Matter Protocol

Simone Sambataro s331812

Prof. Riccardo Sisto

PhD Student Simone Bussa

02TYAUV – Security verification and testing

September 1, 2025

Contents

1 Introduction

The Internet of Things (IoT) now spans homes, cities and industry. Its rapid growth driven by many vendors and platforms, from small microcontrollers to full Systems-on-Chips (SoCs), has often favored performance and speed to market over robustness and security. Always-on connectivity exposes devices to common threats such as eavesdropping, unauthorized access, spoofing, network compromise and it enables large scale attacks (e.g. botnet driven DDoS). As a result, overall reliability depends on protections at every layer, from onboarding to day-to-day operation.

A core requirement for resilience is the ability to fix vulnerabilities quickly with trustworthy firmware updates, delivered over a cable or, increasingly, Over-the-Air (OTA). Updates may be delivered as full images or as deltas and in both cases must perform cryptographic signature verification and integrity checks, with explicit confirmation of successful delivery and installation. While a single reference architecture is still missing, organizations such as ENISA and the IETF are defining requirements and standards. In this context, the Connectivity Standards Alliance (CSA) developed Matter to make devices interoperable and to simplify commissioning (pairing) across ecosystems, with a strong focus on security. Matter specifies two main key establishment protocols: PASE for onboarding new devices and CASE for already commissioned devices, based on SPAKE2+ and SIGMA respectively. Given the rapid adoption 8.696 unique Matter certified devices already listed worldwide the cryptographic guarantees of these protocols warrant careful and rigorous evaluation.

In this project, I present a security and formal analysis of Matter's OTA update process. I model the workflow in the applied pi-calculus and use ProVerif to produce automatic, reproducible proofs. I assess the system's robustness and the security impact of its design choices. The study includes an abstract model of the update executed within a pre-established CASE secured session and of the underlying Bulk Data Exchange (BDX) protocol used for operational file and data transfers plus two threat models inspired by the concepts, assumptions and methodology of the *What's the Matter?* [2] paper, which I extend to advance its results.

2 Matter Description

Matter is a secure, interoperable communication standard that simplifies the user experience and the management of IoT devices from different vendors in home and commercial networks. Built on an open architecture and a robust security model, Matter unifies device commissioning, management and communication, ensuring that data and commands travel securely across the network. With built-in Over-the-Air (OTA) update capabilities, it also enables feature upgrades and security patches without service interruptions or elevated operational risk.

Building Blocks of Matter

Devices and Nodes

A *Device* is a physical unit (e.g. a light bulb or thermostat). A device may host one or more *Nodes*, which are logical, addressable entities exposing distinct functions. During commissioning, each Node is assigned an Operational Node ID and runs the Matter protocol stack independently of other Nodes on the same device (e.g. a smart hub can host multiple Nodes for different roles).

Fabrics

A *Fabric* is a logically isolated and cryptographically protected network in which Matter devices

operate under a shared trust model. A device can belong to multiple Fabrics simultaneously, preserving isolation while enabling cross-ecosystem control (*multi-admin*).

Controllers and Administrators
Controllers manage Nodes, coordinate communications and perform device operations within a Fabric. *Administrators* define access control policies and act as the root of trust by issuing certificates. When joining a Fabric, a device receives an Operational Certificate and stores the Administrator's root certificate. Multi-admin allows controllers from different ecosystems (e.g. Apple HomePod, Amazon Echo) to control the same device [**matter-core-spec**].

Commissioners
A *Commissioner* onboards new devices into a Fabric, verifies credentials and provisions Operational Certificates. A Commissioner may also act as a Controller or Administrator; during commissioning, it is implicitly granted administrative rights on the device.

Security Model (Overview)

Matter's security model relies on two authenticated key establishment protocols [1]:

- **PASE** (Passcode-Authenticated Session Establishment): used for secure initial provisioning (commissioning) of new devices.
- **CASE** (Certificate-Authenticated Session Establishment): used for certificate-based operational sessions among already commissioned devices.

Once a device is commissioned into a Fabric, it can initiate or accept secure communications with other devices in the same Fabric using the CASE protocol.

2.1 Key Establishment Protocols

PASE (Passcode-Authenticated Session Establishment) PASE is based on SPAKE2+, an augmented PAKE that derives a high-entropy shared secret from a low-entropy passcode. In Matter, the *commissioner* (prover) obtains the passcode out-of-band (typically via QR code), while the *commissionee* (verifier) stores only derived verifier material (e.g. (w_0, L)). In deployed devices, the passcode is commonly a static 8 digit value provisioned at manufacturing. Moreover, the device usually does not store the raw passcode, only keying material derived from it.

Message flow (Matter PASE).

1. **PBKDF parameters** The commissioner sends `PBKDFParamRequest`; the commissionee replies with `PBKDFParamResponse` containing salt/iteration parameters.
2. **SPAKE2+ exchange** The parties run SPAKE2+ with Matter-specific deviations to compute a shared secret.
3. **Key derivation** Both sides derive the session key from the SPAKE2+ secret and complete secure channel setup.

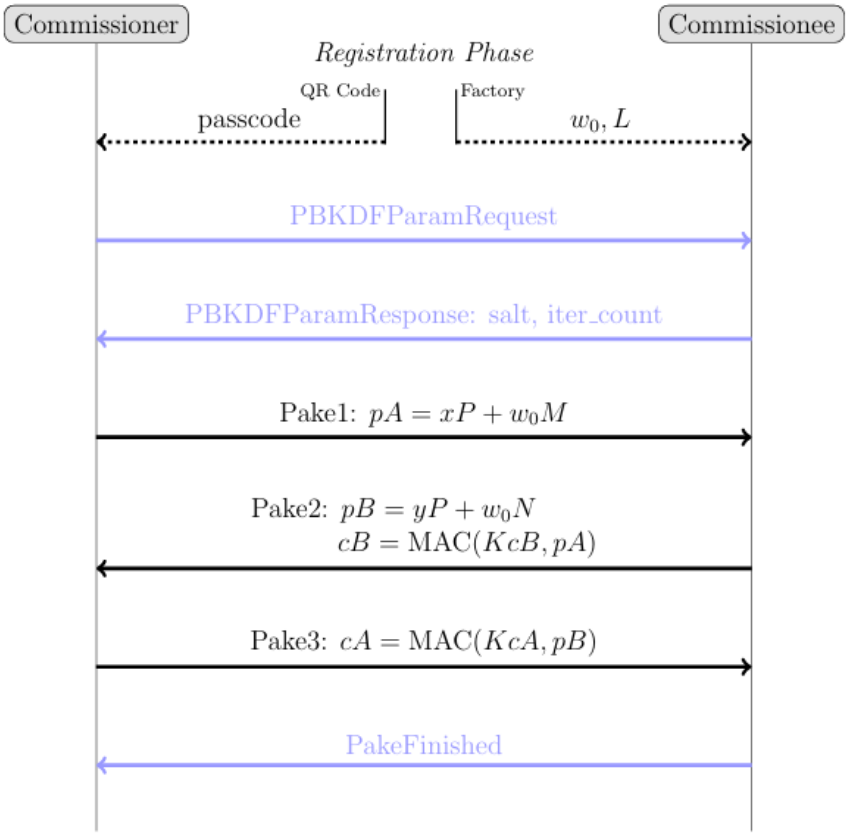

Figure 1: Message sequence chart of Matter's PASE. The offline registration process is represented with dotted lines. Messages related to the PBKDF parameters and the final status report are highlighted in purple, as they are not part of the original SPAKE2+ specification. Matter combines pB and cB into a single message, which, as noted in the SPAKE2+ specification, does not introduce security concerns.

CASE (Certificate-Authenticated Session Establishment) PASE runs between a non commissioned device (commissionee) and a commissioner to create a protected channel for onboarding, culminating in installation of the device's Node Operational Certificate (NOC). By contrast, CASE follows the SIGMA family of authenticated Diffie–Hellman protocols, requires a valid NOC and is therefore used only after commissioning to establish mutually authenticated, secure channels between peers within the same fabric to:

i. exchange ephemeral EC public keys to derive a shared secret,

ii. present identities via NOC certificates,

iii. prove possession of the NOC private key by signing the handshake transcript (ephemerals g^x, g^y and relevant context).

Matter's CASE closely tracks SIGMA-I: the initiator reveals its identity only after verifying the responder, providing initiator identity protection. Application payloads are AEAD protected with keys derived from the shared secret (often denoted $S2K$ and $S3K$ in the spec).

Session resumption Matter supports resuming a previously established CASE session using a resumption identifier and keys derived during the original handshake. Resumption omits

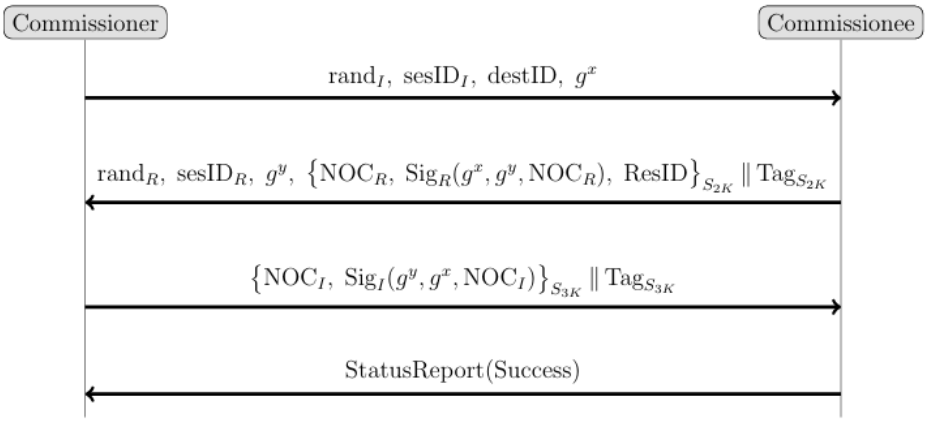

Figure 2: Message sequence chart of the CASE protocol.

fresh signatures to reduce computational cost on constrained devices, enabling fast channel re-establishment using the stored resumption state. While efficient, this mode provides weaker guarantees than a full CASE run (analyzed in Section 2.3).

2.2 Over-the-Air (OTA) Software Update

Matter performs OTA updates *within an established CASE secure session.* Application messages are protected end-to-end with AEAD using direction-specific keys—`I2RKey` (initiator→responder) and `R2IKey` (responder→initiator)— per-session counters and identifiers to provide confidentiality, integrity and replay protection. The session context also retains a resumption secret derived during CASE for fast re-establishment.

2.2.1 Roles

OTA Requestor is the node seeking an update, it acts as client of the *OTA Software Update Provider* cluster and server of the *OTA Software Update Requestor* cluster.

OTA Provider advertises availability, selects suitable images and serves them directly or through proxy (e.g. BDX or via HTTPS). Moreover, Providers are expected to serve devices from multiple vendors to promote interoperability.

2.2.2 Workflow

1. **Discover/Select Provider** The Requestor discovers a Provider (e.g. commissioned records, dynamic discovery, optional `AnnounceOTAProvider`).

2. **Query** The Requestor sends `QueryImage`. The Provider determines availability (optionally consulting the Distributed Compliance Ledger) and replies with status, a download `ImageURI`, `SoftwareVersion` and an `UpdateToken`.

3. `User Consent` Before returning a URI for a Software Image, or allowing an OTA Requestor to apply a previously downloaded update, the OTA Provider SHOULD obtain "User Consent." Within the OTA Cluster, "User Consent" SHALL mean any signal, obtained through implementation specific logic, that conveys informed approval from a User who is administratively authorized to give it. However, because the Core Specification's examples of consent

mechanisms are illustrative rather than normative, I did not model those mechanisms in my ProVerif specification.

4. **Transfer** The Requestor downloads the image using a supported protocol (BDX is mandatory on Providers, HTTPS may be used when appropriate). Providers may proxy or serve cached images so that sleepy devices are supported.

5. **Apply** The Requestor signals readiness via `ApplyUpdateRequest`, that includes the previously issued update token. The Provider responds with an action (*Proceed*, *AwaitNextAction*, or *Discontinue*) and optional delay. Finally, the Requestor applies the image accordingly and then sends the optional message `NotifyUpdateApplied`.

Security Firmware images *must* be signature verified by the Requestor using a vendor key before activation. Anti-rollback checks are enforced both during Provider selection for new firmware and again when the Requestor processes a `QueryImage` response. As mentioned above, data in transit is protected by AEAD within the CASE session using the session's AEAD keys, counter and Session ID, while any at-rest encryption of images is vendor defined.

2.3 Prior Works

What's the Matter? [2] provides the first protocol level security and formal analysis of Matter's core key establishment protocols PASE (a Matter revised version of SPAKE2+) and CASE (a SIGMA-I version with mutual identity protection) modeled in the applied pi-calculus and verified with ProVerif. The authors confirm several intended properties but uncover design and implementation weaknesses. In PASE, low entropy 8 digit passcodes, static salts, missing group membership checks and low PBKDF2 iteration counts make brute-force and pre-computation attacks plausible. They report certified SDK issues (e.g. weak enforcement of attempt limits and acceptance of concurrent PASE sessions), demonstrate proof-of-concept attacks and responsibly disclose vulnerabilities. For CASE, the resumption mechanism weakens guarantees compared to a full handshake. Specifically, compared to the full CASE handshake, CASE resumption drops signature based peer authentication and weakens the binding to the Fabric Identity Protection Key (IPK) . Consequently, compromise of a single ephemeral key can expose resumed session keys even when the IPK remains secret, but peer identities are still hidden. Their ProVerif models [3] confirm mutual authentication and confidentiality for full CASE, but demonstrate this degradation for resumption, under an active Dolev–Yao adversary with scenarios including ephemeral key leakage and insider access to the IPK.

3 Onboarding in ProVerif

This section presents the ProVerif model used to formally specify and verify the OTA workflow executed under a CASE-secured session (that ensure mutual authentication). In order, I describe the cryptographic primitives and equations, the communicating roles and transports, the main orchestration and the security queries that capture secrecy, firmware authentication, integrity, anti-replay and anti-rollback goals.

3.1 Function Declarations

The model declares cryptographic primitives and typed messages that act as the building blocks for confidentiality, authenticity and anti-replay.

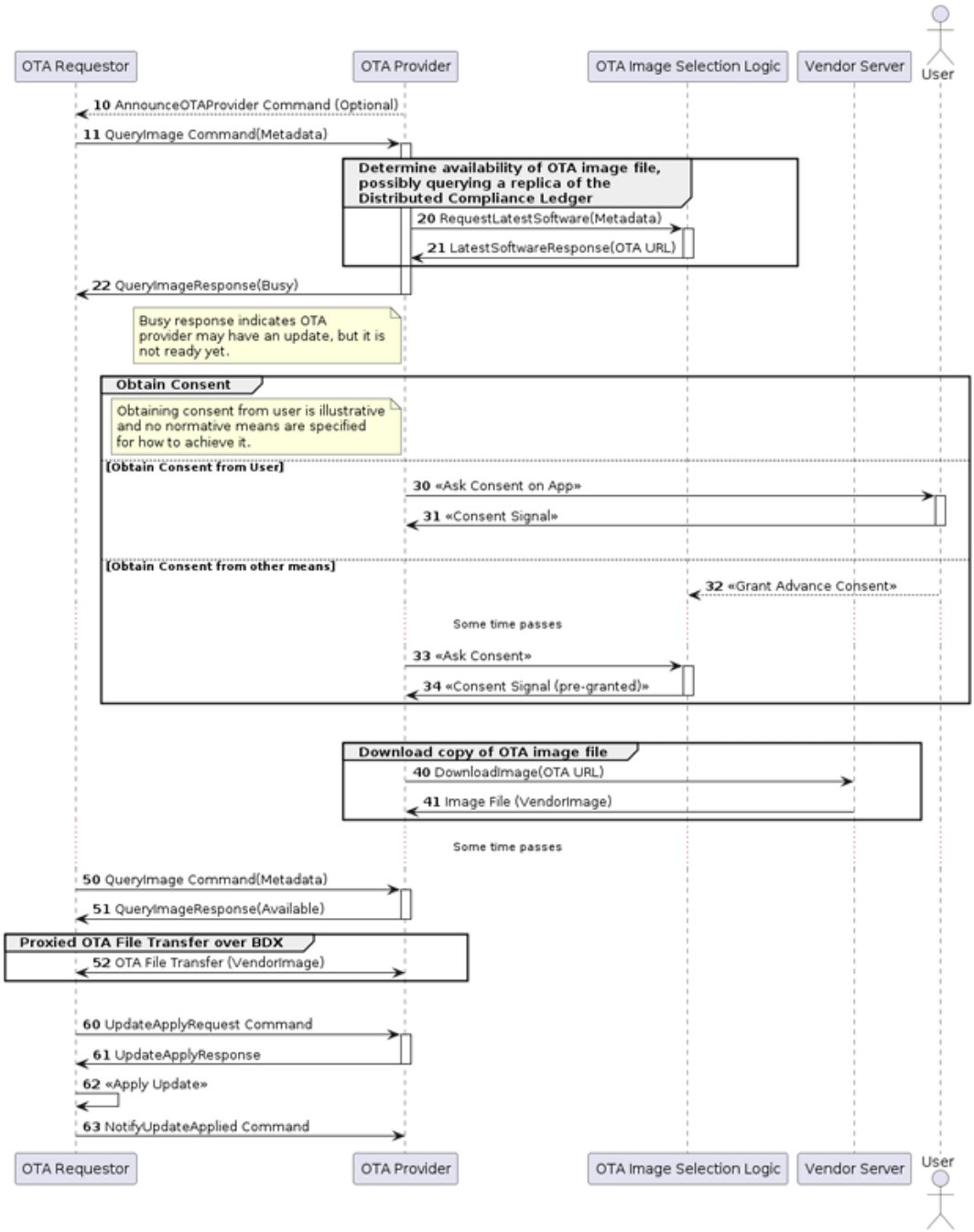

Figure 3: Detailed OTA Software Update Workflow.

AEAD Transport Application frames are protected with AEAD in both directions (`I2R`, `R2I`). Correctness and tag validation are modeled by:

```
reduc forall k m n ad; aead_dec(k, aead_enc(k,m,n,ad), n, ad) = m.
reduc forall k m n ad; aead_dec_validate(k, aead_enc(k,m,n,ad), n, ad) =
    ok().
```

What goes into the AAD. In Matter, the Additional Authenticated Data (AAD) is exactly the on-wire message header:

$$\texttt{MessageFlags} \,\|\, \texttt{SessionID} \,\|\, \texttt{SecurityFlags} \,\|\, \texttt{MessageCounter} \,\|\, [\texttt{SourceNodeID}] \,\|\, [\texttt{DestinationNodeID}].$$

In *secure-unicast*, $\texttt{S} = 0$ and $\texttt{DSIZ} = 0$, so *both* NodeIDs are omitted from the header (and thus from the AAD). In my model I capture this as:

$$\texttt{AAD} = (\texttt{MSGFLAGS}, \texttt{sid}, \texttt{SECFLAGS\_UNICAST}, \texttt{ctr}, \texttt{nil}, \texttt{nil}).$$

What goes into the nonce. The AEAD nonce is derived as

$$\texttt{SecurityFlags} \,\|\, \texttt{MessageCounter} \,\|\, \texttt{SourceNodeID}$$

(with little-endian layout for scalar fields). In this application case the `SourceNodeID` used in the nonce *does not* come from the header (which omits it), but from the *Secure Session Context* bound to `SessionID`. In the model:

$$n = \texttt{nonce}(\texttt{SECFLAGS\_UNICAST}, \texttt{ctr}, \texttt{src\_ctx}).$$

Firmware integrity and authenticity According to the specification, the *ImageDigestField shall* contain the digest of the entire payload of length `PayloadSize` that follows the header. In the model, integrity checks are captured by a collision-resistant `hash` applied to the serialized image, while authenticity is modeled via a perfect signature theory using the Vendor's private key material `KM_VENDOR`:

```
reduc forall m y; getmess(sign(m, sk(y))) = m.
reduc forall m y; checksign(sign(m, sk(y)), pk(y)) = ok().
```

These axioms allow the Requestor to prove that the verified image digest is exactly the value signed by the Vendor, thereby binding the payload to the Vendor's signature.

Typed Application Messages The protocol uses strong tags for each phase: `M_QI` (query), `QI_AVAIL` (update available), `M_DL` (image+digest+sig), `M_APPLY_REQ/RESP` (apply handshake).

Version Monotonicity and URI Sanity Anti-rollback is abstracted by `ver_gt`, axiomatizing that a `NEW()` version is strictly newer than `CUR()`:

```
equation ver_gt(NEW(), CUR()) = ok().
```

URI handling and download I capture the usability of the download hint with the predicate `uri_ok`, which lets the Requestor proceed only when the `ImageURI` is present (when `Status=UpdateAvailable`) and syntactically valid per RFC 3986, as required by the specification. Consistent with Matter's interoperability rules *all* OTA Providers *shall* support BDX and OTA Requestors *should* support BDX, I conservatively assume Providers return `bdx://` URIs so that devices without public Internet HTTPS can still update. Accordingly, `uri_ok` abstracts the BDX constraints.

3.2 Channels

Two channels reflect deployment reality:

- `net`: public fabric channel carrying AEAD protected traffic.
- `cv`: a private Vendor→Provider provisioning backchannel to presort (`digest`, `image`, `sig`, `NEW()`), to model the case where the firmware image is cached at the Provider before the Requestor's query.

3.3 Process Macros

The model instantiates three roles:

Vendor Generates a fresh image, computes its digest, signs it with `KM_VENDOR`, announces `NEW()` and provisions the tuple over `cv`. It raises `issued(fw, NEW())`, which later supports authenticity correspondence. Note that the software version is not included in the digest, since integrity is assumed to be ensured by the secure session.

OTA Provider

1. **Receive `QueryImage`** (I→R): decrypts and parses `M_QI` if `aead_dec_validate` succeeds; records `RecvI2R(sid,ctr,...)`.
2. **Monotonicity gate**: serves an update only if `ver_gt(nsv, csv) = ok()` (prevents rollbacks at the source and allow Providers to select the correct new Software version).
3. **Advertise update**: replies with `QI_AVAIL(nsv, utok, dly, URI, vstr)` over R→I; emits `QueryAnsweredP` and `TokenIssuedSID(sid,utok)`.
4. **Send image**: transmits `M_DL(fw, digest, sig)` over R→I; emits `DownloadStart`.
5. **Handle Apply**: upon `M_APPLY_REQ(utok, nv)`, enforces coherence $nv = nsv$ then returns `M_APPLY_RESP(ACT_PROCEED, dly)`. Consistent with Matter 1.4, an invalid `UpdateToken` *must not* be used as a reason to continuously deny or delay an `ApplyUpdateRequest`, the token is intended only for short term bookkeeping of deferred application by the Provider. In the model I therefore do not explicitly validate the token and treat it purely as a lightweight tracker for the update flow, which is faithful to the specification's "UpdateToken usage" language and to the `ApplyUpdateRequest` processing rules (state, if kept, may relate only to the token and prior apply requests).

OTA Requestor

1. **Query**: sends `M_QI(VID, PID, CUR())`, which are respectively VendorID, ProductID and current Software version of the requestor and records `OTABegin`, `QuerySent`.
2. **Process response**: if `QI_AVAIL`, checks `ver_gt(nsv, CUR())` ⇒ `VersionEligible` and `uri_ok(URI)`.
3. **Download & verify**: receives `M_DL`, verifies `checksign(sigf, pk(KM_VENDOR))` and binds `getmess(sigf)` to `hash(image_to_bitstring(fw))`, plus digest equality. On success: `authenticated(fw,nsv)`, `VersionAcceptedStrong`, `ReadyToApply`.
4. **Apply**: sends `M_APPLY_REQ(utok,nsv)`, expects `M_APPLY_RESP`; if `ACT_PROCEED`, raises `OTAEnd`.

3.4 Main Process

The orchestration runs `Vendor | Requestor | Provider` in parallel under replication, modeling many independent OTA interactions.

```
process
  !(
     new k_i2r : key;
     new k_r2i : key;
     new ReqId : bitstring;
     new ProvId: bitstring;
     new SESSID: bitstring;
     new SRC_CTX: bitstring;
     ( Vendor()
     | Requestor(ReqId, ProvId, k_i2r, k_r2i, SESSID, SRC_CTX)
     | Provider(ProvId, k_i2r, k_r2i, SESSID, SRC_CTX) )
   )
```

3.5 Queries

The model verifies secrecy, authenticity, correct ordering, anti-replay and apply-phase discipline via correspondence queries.

- **Key Secrecy**: the Vendor's private signing key remains unknown:

```
query attacker(sk(KM_VENDOR)).
```

- **Firmware Authenticity**: any installed/authenticated image must have been issued by the Vendor:

```
query fw:image, nv:bitstring; event(authenticated(fw,nv)) ==> event
   (issued(fw,nv)).
```

- **Ordering**: a download can start only after the Provider answered `QueryImage`; `DownloadDone` implies a prior `DownloadStart` for the same tuple.
- **Injective Anti-Replay**: each received frame (I→R and R→I) has a unique matching send event keyed by (`sid`, `ctr`), ensuring per-session counter freshness.

- **Apply-Phase Authentication**: an `ApplyRespRcv` implies a fresh, matching `ApplyRespSent` and a preceding `ApplyReqRcv`; `ApplyReqSent` is allowed only after `ReadyToApply`.
- **Token Freshness**: every `ApplyReqSent(sid,tok,nsv)` references a `TokenIssuedSID(sid,tok)`, tying authorization to the correct session.
- **Reachability**: `endP` and `endR` ensure test reachability.

3.6 Query Results

Under the modeling stance of Section 3, ProVerif discharged all goals, no attacks were found. Concretely, key secrecy holds (`not attacker(sk(KM_VENDOR))`); authenticated firmware is always vendor issued (`event(authenticated(fw,nv)) ⇒ event(issued(fw,nv))`); downloads begin only after a `QueryImage` answer (`DownloadStart ⇒ QueryAnsweredP`); completion implies a prior start for the same Provider/URI/image (`DownloadDone ⇒ DownloadStart`); injective anti-replay is ensured in both directions, keyed by (`sid, ctr`) (`RecvI2R ⇒ inj-event(SendI2R)` and `RecvR2I ⇒ inj-event(SendR2I)`); apply responses are uniquely authenticated (`inj-event(ApplyRespRcv) ⇒ inj-event(ApplyRespSent)`); apply requests are sent only after readiness (`ApplyReqSent ⇒ ReadyToApply`); the apply sub-protocol is end-to-end linked (`inj-event(ApplyRespRcv) ⇒ (inj-event(ApplyRespSent) ⇒ inj-event(ApplyReqRcv))`); token usage is session scoped (`ApplyReqSent ⇒ TokenIssuedSID`); and both roles can complete at least once (`event(endP())`, `event(endR())` are reachable).

3.7 Bulk Data Exchange (BDX) for OTA Downloads

For completeness, I augmented the OTA workflow analysis with a separate formal treatment of the underlying Bulk Data Exchange (BDX) protocol.

Overview and roles Matter defines the *Bulk Data Exchange (BDX)* protocol to move opaque "files" (byte streams plus metadata) between nodes, for example, to deliver an OTA image from a Provider to a Requestor. BDX borrows some semantics from TFTP but runs over Matter's reliable transports with MRP providing reliability when needed. The protocol distinguishes the *BDX Sender* (the node that has data), the *BDX Receiver* (the node that receives data) and the Initiator/Responder roles, which depend on who starts the transfer (download vs. upload). In downloads relevant to OTA, the Requestor acts as *BDX Receiver & Initiator*, it sends `ReceiveInit`, while the Provider acts as *BDX Sender & Responder*, it answers with `ReceiveAccept`.

Transfer modes BDX supports two modes. In *synchronous* mode, one party, the *Driver*, gates progress and each protocol message *must* be acknowledged before the next is sent. Moreover, the specification mandates that the follower, determined by the transfer direction, cannot be a sleepy device. In *asynchronous* mode, the messages can be sent back-to-back and the underlying transport ensures flow control. For OTA over a secure channel, Matter constrains transports as follows: Receiver-Drive (synchronous) *shall* be used on non-TCP and Asynchronous on TCP, it also specifies idle timeouts and negotiated block-size bounds. Specifically, the proposed model targets the receiver-driven case mandated for non-TCP OTA transfers.

Message flow of the model Receiver-driven download:

1. `ReceiveInit` → `ReceiveAccept` (parameter agreement).
2. `BlockQuery` → `Block` (the next query with incremented message counter acts as an implicit driver's ack).
3. `BlockQuery` → `BlockEOF` (end-of-file marker) → `BlockAckEOF`.

This captures the "one request—one response" discipline of synchronous BDX: every Sender message is triggered by and paired with, a preceding Receiver query/ack.

Modeled variant and abstraction To keep the state space tractable while remaining faithful to the spec semantics, I adopt a minimal variant:

- **Single-block payload** The actual firmware image, digest and vendor signature are abstracted as one `Block` carrying a typed chunk (`bdx_payload(fw, dg, sig)`).
- **Explicit EOF credit** The Sender emits `BlockEOF` only after a *second* `BlockQuery` from the Receiver, with a distinct sequence (the way I model the incremented counter), mirroring the Driver's requirement to acknowledge each step before the next message is sent. This enforces the "one request—one response" constraint the spec imposes in synchronous mode.
- **Security context** Confidentiality/integrity are inherited from the established secure session. The proposed model therefore uses abstract AEAD primitives and focuses BDX on flow control and ordering rather than on cryptographic key exchange.

Why this abstraction Receiver-driven BDX is the normative choice for OTA over non-TCP and its message pairing discipline (query↔block, query↔EOF) is exactly what my correspondence queries check: (i) every received `Block` has a prior `BlockQuery`; (ii) `BlockEOF` is sent only after a distinct final query; and (iii) transfer completion implies an `EOF` was sent. Modeling a single data block and an explicit EOF credit preserves these safety properties while avoiding incidental complexity (segmentation, retries) that would not affect the verified guarantees.

OTA binding Within OTA, when the Provider returns a `bdx://` URI in the QueryImageResponse, the Requestor *shall* retrieve the image via BDX under the transport constraints modeled above.

4 Threat Model

The analyzed adversarial model is aligned with the Matter R1.4 specification and its catalog *Threats and Countermeasures* and is further informed by the scenarios discussed in the *What's the Matter?* [2]. In particular, I formally test the claim that *CASE session resumption* constitutes a security degradation by instantiating their scenarios within my OTA analysis, which as said inherits keys and session state from the preceding key establishment phases.

Network adversary In particular, I model an external adversary with standard Dolev–Yao capabilities over the network. More in details, a *passive* attacker can eavesdrop and record protocol transcripts, subsequent ciphertexts and can perform offline analysis or reordering of recorded data. An *active* attacker can additionally inject, replay, delay and drop messages before they reach their intended recipients. These capabilities are used to exercise attacks relevant to OTA (e.g. replay, rollback, ordering, message tampering). Additionally, I suppose that the external adversary does not have physical access to devices: no hardware tampering, key extraction from secure elements, side-channel attacks or invasive debugging are considered. Further, the model assumes perfect cryptography (no cryptanalytic breaks) and concentrates on protocol level flaws and binding errors, including those potentially introduced by CASE resumption.

4.1 Security Degradation from CASE–Resumption Leaks

Prior work argues that *CASE with resumption weakens security*: if *either* party's ephemeral private key (say, x or y) leaks, an adversary can recompute the `SharedSecret` for the resumed handshake and, from it, the operational transport keys. By contrast, in a full (non-resumed) CASE run, recovering transport keys requires knowledge of the Fabric's `IPK` (e.g. by being on the same Fabric), which is a stronger prerequisite.

In my reading of the specification I found that:

```

I2RKey || R2IKey || AttestationChallenge = Crypto_KDF(
inputKey = SharedSecret,
salt     = Sigma1.initiatorRandom || ResumptionID,
info     = "SessionResumptionKeys",
len      = 3 * keylen
)
```

Hence, leakage of one ephemeral together with the peer's ephemeral *public* value and the risk of eavesdropping, since the CASE session-establishment messages are transmitted in clear text, suffices to reconstruct `SharedSecret` and derive `I2RKey`/`R2IKey` under resumption. Additionally, as further confirmation of the paper's observations for a full CASE run, the used of the IPK is essential to obtain the transport keys:

```

TranscriptHash = Crypto_Hash(Msg1 || Msg2 || Msg3)
    I2RKey || R2IKey || AttestationChallenge =
        Crypto_KDF( inputKey = SharedSecret,
                    salt = IPK || TranscriptHash,
                    info = "SessionKeys", len = 3 * keylen )
```

Residual protection in secure unicast Even with those transport keys, the adversary still lacks the session's *SourceNodeID*, which is *not* carried in clear in the secure–unicast header. In fact, Matter's AEAD nonce is built as `nonce(SECFLAGS_UNICAST, MessageCounter, SourceNodeID)`, and the `SourceNodeID` component is obtained from the *secure session context*, not from the wire header. Consequently, keys alone do not enable decryption/forgery unless the attacker can also learn the session's node identifiers (or otherwise reconstruct the nonce stream). This aligns with the observation that, as long as `IPK` (and with it the operational identity material) remains unknown, previously harvested ciphertexts of party identities are not decryptable.

Model check (keys leaked, context unknown) I encode this scenario by *explicitly leaking* the operational AEAD keys while keeping the session context private and I obtain that all safety queries continue to hold under this stress test:

```
process
!(
new k_i2r : key;              (\* AEAD key for I->R direction *)
new k_r2i : key;              (* AEAD key for R->I direction *)
new ReqId : bitstring;        (* Requestor correlation id *)
new ProvId: bitstring;        (* Provider correlation id *)
new SESSID: bitstring;        (* Session identifier bound in AAD *)
new SRC_CTX: bitstring;       (* Session SourceNodeID for nonces *)
( Vendor()
 | Requestor(ReqId, ProvId, k_i2r, k_r2i, SESSID, SRC_CTX)
 | Provider(ProvId, k_i2r, k_r2i, SESSID, SRC_CTX)
 | (  (* === SCENARIO A: leak operational AEAD keys only === \*)
out(net, k_i2r);
out(net, k_r2i)
)
)
)
```

Listing 1: Scenario A: leak transport keys, keep session context private

4.2 Security Degradation under Severe CASE Leaks

In this scenario ("**Scenario B**"), I consider a stronger compromise than simple resumption key leakage. Besides learning one ephemeral private key used during CASE establishment or resumption, the adversary is also assumed to be *in the same Fabric* and so, to have obtained the Fabric's *Identity Protection Key (IPK)* (i.e. the Operational Group Key). Under these conditions, the attacker can recover peers' operational identities (Source/Destination Node IDs) and the session context that is otherwise unavailable on the wire in secure unicast. Consequently, the adversary can reconstruct the per–message nonce `nonce(SECFLAGS_UNICAST, ctr, src_ctx)`, bind it to the visible AAD (which carries `sid` and `ctr`) and *forge valid AEAD frames* in both directions, taking effective control of the channel.

To reflect this in the model, I explicitly leak the transport keys and the session context:

```
process
!(
new k_i2r : key;              (* AEAD key for I->R direction *)
new k_r2i : key;              (* AEAD key for R->I direction *)
new ReqId : bitstring;        (* Requestor identity / correlation id *)
new ProvId: bitstring;        (* Provider identity / correlation id *)
new SESSID: bitstring;        (* Session identifier used in AAD/events *)
new SRC_CTX: bitstring;       (* Represents the NodeIDs from the session
                                 context *)
( Vendor()
 | Requestor(ReqId, ProvId, k_i2r, k_r2i, SESSID, SRC_CTX)
 | Provider(ProvId, k_i2r, k_r2i, SESSID, SRC_CTX)
 | (  (* === SCENARIO B: keys + source context are leaked to attacker
    === *)
out(net, k_i2r);
out(net, k_r2i);
out(net, SRC_CTX)
)
)
)
```

Listing 2: Scenario B: leak operational AEAD keys and session context (NodeID)

Query outcomes (Scenario B) ProVerif confirms that once both the AEAD transport keys and the nonce context (`SRC_CTX`) are compromised, *channel-level* guarantees collapse, while *signature based provenance* of the firmware continues to hold.

What still holds (i) **Vendor key secrecy** remains intact (`not attacker(sk(KM_VENDOR))` is **true**). (ii) **Issuance binding:** any `authenticated(fw,nv)` implies `issued(fw,nv)` (**true**); the attacker cannot forge a Vendor signature over an arbitrary image. (iii) **Local ordering at the Provider** is preserved: `DownloadStart` implies a prior `QueryAnsweredP` (**true**). (iv) **Requester discipline** remains local: `ApplyReqSent` implies `ReadyToApply` (**true**). (v) **Liveness:** the Provider's termination is reachable (`not event(endP)` is **false**); Requestor termination is not proved, that means that some adversarial traces can block progress.

What breaks (and why) With `k_i2r`, `k_r2i` and `SRC_CTX` leaked:

- **Download completion without a genuine start:** `DownloadDone` ⇒ `DownloadStart` is **false**; attacker-crafted `R2I` frames can drive the Requestor to "complete" a download that the Provider never started.
- **Anti-replay (injective) fails in both directions:** `RecvI2R` ⇒ `inj-event(SendI2R)` and `RecvR2I` ⇒ `inj-event(SendR2I)` are **false**; fresh forgeries pass AEAD checks.
- **Apply-phase authentication collapses:** `inj-event(ApplyRespRcv)` ⇒ `inj-event(ApplyRespSent)` is **false**; the attacker can inject a plausible `M_APPLY_RESP` (e.g. flipping `ACT_PROCEED` to `ACT_DISCONTINUE`) to suppress updates.
- **Full apply linkage breaks:** `inj-event(ApplyRespRcv)` ⇒ (`inj-event(ApplyRespSent)` ⇒ `inj-event(ApplyReqRcv)`) is **false**; forged responses need not correspond to any Provider-side request.
- **Token discipline fails:** `ApplyReqSent(sid,tok,·)` ⇒ `TokenIssuedSID(sid,tok)` is **false**; a fake `QI_AVAIL` lets the attacker inject an arbitrary `tok`.

This matches the intended threat model: once transport keys and nonce context are compromised, channel level guarantees collapse, but signature based provenance of the firmware still protects against content forgery.

4.3 PASE Passcode Exposure: Commissioning and OTA Implications

SDK weaknesses can make PASE passcodes retrievable, raising the question: what does a compromised PASE buy an attacker for OTA? Although PASE precedes CASE during commissioning, a compromised PASE session does not automatically translate into compromised operational access. Before a device can join a fabric (and thus use CASE), the commissioner performs Device Attestation (DAC/PAI chain + Certification Declaration) over a fresh challenge. If attestation fails or is untrusted, commissioning must halt or present a clear user warning and require explicit consent, no NOC is issued and no operational session is established.

Implicit PASE grant During commissioning, the PASE channel confers an *implicit*, temporary `Administer` right (modeled as a non persistent ACL entry with `AuthMode=PASE` that covers the whole node) and only for the commissioning window. After `AddNOC`, the device automatically installs a persistent ACL that grants `Administer` under `AuthMode=CASE` to the designated `CaseAdminSubject`, from that point onward, all administration proceeds over CASE.

No persistent PASE ACLs The Access Control Cluster forbids writing ACL entries with `AuthMode=PASE`, such writes *shall* fail with `CONSTRAINT_ERROR`. PASE is reserved for bootstrapping/implicit behavior and persistent administrative policy must be expressed under CASE. Consequently, one cannot "enable OTA over PASE" by authoring PASE-based ACLs, OTA operations are authorized via CASE-scoped ACLs (as reflected in the specification's examples that grant privileges under `AuthMode=CASE`).

Security note With the PASE passcode, an attacker could momentarily hijack commissioning, enroll their own NOC via `AddNOC` and then install CASE-based ACLs (e.g. pointing the device to an attacker controlled OTA Provider). However, absent additional failures, most notably if DAC checks still hold, a PASE compromise alone neither directly nor silently compromises a CASE session. Moreover, arbitrary firmware cannot be installed: the Requestor accepts an image only if its signature verifies against the vendor public key provisioned at manufacturing.

5 Conclusion

The purpose of the project was to formalize Matter's OTA workflow on top of a CASE-secured channel in the applied π-calculus and verified it with ProVerif. Under baseline assumptions (established CASE, perfect cryptography and the message-layer bindings mandated by Matter), all goals were discharged: vendor key secrecy, end-to-end firmware authenticity, strict request/response ordering, injective anti-replay in both directions and sound apply-phase correspondences. In brief, with uncompromised session state the OTA workflow is secure.

Next, I stress-tested CASE to understand its implications on the OTA workflow. Prior work correctly notes that session resumption is a security downgrade relative to a fresh CASE run. However, in secure-unicast, key exposure alone may still be insufficient to decrypt or forge, because AEAD nonces bind the session's *SourceNodeID*, which is not on the wire. The "keys-only" leak (exposing `I2RKey`/`R2IKey` but not session context) preserved all safety properties. By contrast, a *severe* leak that also exposes Fabric identity material (e.g. IPK) undermines channel-level security, enabling active attacks rebuild nonces and craft valid frames while vendor signatures still prevent content forgery. The boundary is clear: provenance remains robust, yet the channel remains vulnerable to leaks of fabric-level identities. Nonetheless, setting aside the deliberately harsh stress-test assumptions, Matter's OTA Software updates procedure is well designed and the formal analysis supports the properties targeted by the authors.

5.1 Future work

A concrete next step is to operationalize these proofs as SDK conformance checks: automatically verifying that OTA clients respect the specification's minimum retry/backoff interval ($\geq 120\,\mathrm{s}$) and that Providers implement idempotent handlers. Violating these constraints can trigger correlated retry storms that overload the Provider and jeopardize updates across an entire Fabric, a risk echoed by prior findings of missing enforcement in SDKs for the PASE inadequate brute-force throttling.

References

[1] Connectivity Standards Alliance (CSA). *Matter Core Specification, Version 1.4*. Available online. 2024. URL: https://csa-iot.org/developer-resource/specifications-download-request/.

[2] Sayon Duttagupta et al. *What's the Matter? An In-Depth Security Analysis of the Matter Protocol*. Cryptology ePrint Archive, Paper 2025/1268. https://eprint.iacr.org/2025/1268. July-2025.

[3] KU Leuven COSIC. *What's the Matter? Artifacts and Code*. https://github.com/KULeuven-COSIC/whats-the-matter. GitHub repository. July-2025.