Formal specification and verification of the OTA Chain-Of-Custody Protocol

Leonardo Rizzo s328764

Prof. Riccardo Sisto
PhD student Simone Bussa
02TYAUV - Security verification and testing

June 11, 2025

Contents

1 Introduction

The Internet of Things (IoT) is characterized by a rapidly growing network of connected devices. These devices are increasingly becoming targets for various attacks, exploiting software vulnerabilities and design flaws. This highlights the critical importance of designing robust architectures for managing cryptographic keys, which are essential for both the initial configuration (onboarding) of IoT devices and the secure, automatic updating of their software and firmware.

A significant challenge lies in updating low-level IoT devices. These devices often have constrained processors with limited registers and caches, making it difficult to perform the complex public-key cryptographic operations commonly used for key establishment and software update authentication. This report addresses this challenge by presenting an architecture designed specifically for onboarding and secure software updates of these low-level IoT devices, such as those using 8-bit microcontrollers. The proposed architecture leverages elliptic curve cryptography, authenticated key establishment, and a continuity-based key-locking mechanism to ensure secure updates, even for IoT devices with limited capabilities.

2 Protocol Description

This section provides a detailed description of the proposed protocol for secure onboarding and software updates for IoT devices. As shown in Figure 1, the system design involves four main components: a SmartApp (Smartphone Application) used for device registration, a Software Update Provider (SUP) responsible for providing software updates, a Gateway device facilitating communication, and the IoT Device itself, the target of the updates. The protocol encompasses processes and interactions between these four actors to achieve secure onboarding and software update.

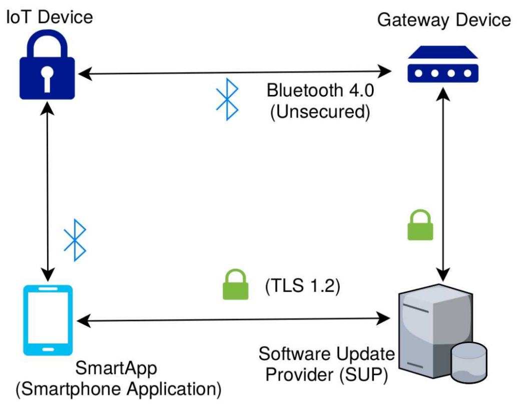

Figure 1: System Design for Onboarding and Software Update.

2.1 Onboarding Description

The onboarding process focuses on establishing trust and secure communication between the IoT device and the Gateway device. Given that Class 0 IoT devices typically lack a user input/output interface, a method is needed to securely introduce the device to the network. The proposed solution uses a ”chain-of-custody” concept to achieve authenticated key exchange.

Communication between the SmartApp, the SUP, and the Gateway occurs over TLS 1.2. However, the communication between the Gateway and the IoT device is carried out over Bluetooth 4.0, which, unlike TLS, is inherently insecure. This necessitates a careful design of the onboarding process to ensure secure communication. To address this, precautions must be taken, such as the negotiation via ECDH of a symmetric key to encrypt the traffic that would otherwise be broadcast in Bluetooth.

The onboarding process involves the following steps:

1. The user downloads and installs the SmartApp from a trusted source and creates an account with the SUP.
2. The user initializes the Gateway device and configures it with the SUP's URL, username, and password using a laptop. During initialization, the Gateway generates an EC public-private key pair ($e1_G$, $d1_G$) and an RSA public-private key pair (e_G, d_G).
3. The user scans the RSA public key (e_G) of the Gateway device with the SmartApp and saves it.
4. The user initializes the IoT device. The device generates an EC key pair ($e1_D$, $d1_D$) for ECDH. The user enters the serial number (N_D) and IoT data (unique password w_D and Bluetooth MAC-address B_D) by scanning a QR code from the device's booklet.
5. The SmartApp encrypts the IoT data with the Gateway's public key (e_G) and sends it to the SUP over TLS, along with the device's serial number (N_D). The SUP stores this information.
6. The Gateway device requests IoT data from the SUP over TLS. The SUP retrieves and sends the encrypted IoT data to the Gateway. The Gateway decrypts the IoT data using its RSA private key (d_G) and stores them.
7. The Gateway sends its EC public key ($e1_G$) to the IoT device via Bluetooth. The IoT device receives ($e1_G$) and generates a shared secret key (K_D) using ECDH. It also generates a random symmetric session key (k) and encrypts it using K_D.
8. The IoT device sends its public key ($e1_D$) and the encrypted session key to the Gateway. The Gateway receives ($e1_D$) and generates shared secret key ($K_G = K_D = K$) and decrypts the session key (k).
9. The Gateway sends a double hash of the session key (k) concatenated with W (where $W = f(w_D)$) to the IoT device. The IoT device verifies this hash.
10. The IoT device sends a single hash of the session key (k) concatenated with W to the Gateway. The Gateway verifies this hash and accepts K as the session key.
11. The Gateway sends a request for known device data to the IoT device, encrypted with K. The IoT device responds with the requested data (device model, manufacturer, current version, required version), encrypted with the session key.
12. The IoT device sends the encrypted known device data to the Gateway. The Gateway decrypts and stores this data.

This process establishes a secure communication channel between the IoT device and the Gateway, enabling secure software updates.

2.2 Secure Update Description

After successful onboarding, the devices can proceed with their intended functionality, including software updates. The integrity of the software update image is maintained using digital signatures.

The software update process involves the following steps:

1. The Gateway device sends a request for manifest data ($M_{T(D)}$) to the SUP, including known device data.
2. The SUP checks if the request is for an upgrade by examining the $reqVersion$ field. If it's a request for an upgrade, the SUP searches its database for a suitable update, generates a timestamp, and signs the manifest data ($M_{T(D)}$).

3. The SUP sends the signed manifest data to the Gateway via TLS. The Gateway forwards this data to the IoT device, encrypted with the session key.
4. The IoT device verifies the signature and validates the device model, manufacturer, software version, and timestamp.
5. If the verification and validation are successful, the Gateway requests the software image ($I_{T(D)}$) from the SUP.
6. The SUP retrieves the software image from its database, which includes a new public verification key for the next update. The SUP signs the software image ($I_{T(D)}$) with the same private key used to sign $M_{T(D)}$.
7. The SUP sends the signed software image to the Gateway, which forwards it to the IoT device, encrypted with the session key.
8. The IoT device decrypts the image, verifies the signature using the stored verification key ($verNextKey_D$), and validates the device information.
9. After successful verification and validation, the IoT device installs the new software image. It then updates the verification keys ($verCurrKey_D$ and $verNextKey_D$) and sends an acknowledgment to the Gateway.
10. The Gateway stores the updated device data.

In case of an update installation failure, the IoT device reverts to the base image and notifies the Gateway.

2.3 Key Locking Mechanism

The secure update process relies on a **key locking** mechanism. This mechanism embeds a verification key in each software image, used to verify the integrity and authenticity of the **next** software image.

- Each software version has its own public-private key pair for signing and verification.
- The verification key for a software version is included in the **previous** version.
- The manufacturer embeds a base image with an initial verification key in the device's ROM during manufacturing.
- When an update is received, the IoT device uses the verification key from the **current** received image to verify the signature of the **next** image.
- After a successful update, the device stores the verification key from the **new** image for future updates.

Example:

1. The device is initially manufactured with software version 1.0, which includes a verification key $VKey\_2$ in its base image in ROM. $VKey\_2$ is the verification key corresponding to version 2.0.
2. When version 2.0 of the software is available, the SUP signs the update using the signing key $SKey\_2$ (corresponding to $VKey\_2$).
3. The IoT device uses $VKey\_2$ (stored in ROM) to verify the signature of the 2.0 update.
4. If the verification succeeds, the device installs the 2.0 update and stores the verification key $VKey\_3$ (which is included **in** the 2.0 update). $VKey\_3$ will be used to verify the signature of version 3.0, and so on.

This key-locking mechanism ensures a chain of trust, as each update verifies the next, preventing malicious or unauthorized updates from being installed.

2.4 Other Security Properties

Beyond key locking, the protocol incorporates other security properties:

- **Authenticated Key Exchange:** The onboarding process uses ECDH to securely establish a shared key between the IoT device and the gateway, preventing unauthorized devices from communicating.
- **Data Integrity and Authentication:** Digital signatures are used to ensure the integrity and authenticity of software updates, preventing tampering and verifying the source of the update.
- **Replay Attack Prevention:** Timestamps are included in the manifest data and software images to prevent replay attacks, where an attacker attempts to install an older version of the software.

3 Onboarding in ProVerif

This section details the ProVerif model used to formally specify and verify the onboarding protocol. It describes the functions, processes, main process, and queries defined in the ProVerif code.

3.1 Function Declarations

The ProVerif model begins by declaring the cryptographic functions and data types used in the protocol. These declarations define the basic building blocks for secure communication and key agreement.

- **Symmetric Key Encryption:** The function `senc` models symmetric key encryption, taking a bitstring and a key (of type `key`) as input and returning an encrypted bitstring. The corresponding decryption function `sdec` satisfies the reduction rule:

  ```
  reduc forall m: bitstring, k: key; sdec(senc(m, k), k) = m.
  ```

  This ensures that decryption recovers the original message.
- **Asymmetric Key Encryption:** The types `skey` and `pkey` represent secret and public keys respectively. The function `aenc` models asymmetric encryption, taking a bitstring and a public key as input. The corresponding decryption function `adec` satisfies the reduction rule:

  ```
  reduc forall m: bitstring, sk: skey; adec(aenc(m, pk(sk)), sk) = m.
  ```

- **Digital Signatures:** The functions `sign` and `checksign` (with associated types `sskey` and `spkey`) are declared to model digital signatures and their verification.
- **Hash and Key Derivation Functions:** The hash function `h` is used to compute a digest of bitstrings, while `f` is defined as a key derivation function.
- **Elliptic Curve Diffie-Hellman (ECDH):** The types `G` and `exponent` model the group and exponents used in ECDH. The function `exp` models modular exponentiation, and the following equation captures the commutative property of exponentiation:

  ```
  equation forall x: exponent, y: exponent;
  exp(exp(g, x), y) = exp(exp(g, y), x).
  ```

This models the property that $(g^x)^y = (g^y)^x$, which is fundamental to the Diffie-Hellman key exchange.

- **Concatenation Functions:** The functions `concat` and `concat_key` are used to concatenate bitstrings and keys, respectively.

3.2 Channels

The onboarding process utilizes three communication channels:

- `c`: This channel represents the communication between the SmartApp and the SUP (Software Update Provider). It is modeled as a private channel (`free c: channel [private].`) to reflect the use of TLS for secure communication.
- `c1`: This channel represents the communication between the SUP and the Gateway. Similar to the previous channel, it is also modeled as a private channel (`free c1: channel [private].`) due to the use of TLS.
- `c2`: This channel represents the communication between the Gateway and the IoT Device. It is modeled as a public channel (`free c2: channel.`) to reflect the use of Bluetooth, which is inherently insecure in its standard form.

3.3 Process Macros

The ProVerif model defines four process macros to represent the different roles involved in the onboarding protocol:

- **SmartApp:** This process models the behavior of the smartphone application. It takes as input the Gateway's public key (`e_G`), the IoT device's serial number (`N_D`), password (`w_D`), and Bluetooth MAC address (`B_D`). It encrypts the sensitive tuple (`w_D, B_D`) using asymmetric encryption with `e_G` and sends the resulting message along with the serial number to the SUP.

```
let SmartApp(e_G: pkey, N_D: bitstring, w_D: bitstring, B_D: bitstring)=
  let M = aenc((w_D, B_D), e_G) in
  out(c, (M, N_D));
  0.
```

- **SUP:** This process models the behavior of the Software Update Provider (SUP). The SUP receives the encrypted message from the SmartApp and simply forwards the encrypted data (the ciphertext M along with the serial number N_D) to the Gateway via a secure channel. This relaying is critical to maintain end-to-end integrity without exposing the plaintext.

```
let SUP(e_G: pkey) =
  in(c, (M: bitstring, N_D: bitstring));
  out(c1, M);
  0.
```

- **Gateway:** The Gateway plays a more active role. Upon receiving the message M from the SUP:
  1. **Decryption and Credential Recovery:** It decrypts M using its secret key d_G to get (w_D, B_D).
  2. **Challenge Generation:** It generates a random challenge (a bitstring) and records the event begin_auth_G$(\text{challenge}, w_D)$.

3. **Sending Challenge:** The challenge is sent over a Bluetooth channel (c2) to the IoT Device.
4. **Response Verification:** After the IoT Device replies, the Gateway verifies the response by computing the expected hash of the concatenated challenge and password.
5. **ECDH Key Agreement:** Once authenticated, the Gateway sends a group element ($e1_G$) to the device. The Gateway later receives the encrypted shared key k from the device, computes the ECDH key K, and obtains k.
6. **Hash Comparison:** It uses additional hash operations and the key derivation function to ensure that both parties have derived the same shared key before requesting additional data.

- **IoT_Device:** The IoT device waits for the challenge from the Gateway. Once received:

  1. **Challenge Response:** It computes the response by hashing the concatenation of the challenge and its password and sends it back.
  2. **Key Agreement:** After receiving the group element from the Gateway, it computes the shared key K using its own exponent.
  3. **Key Generation and Encryption:** The device generates a key k, records an event ev_key_D(k), and sends the encrypted key to the Gateway.
  4. **Verification through Hash Comparison:** It receives a double-hash from the Gateway, performs its own computation, and if the values match, signals that the Gateway has the correct key with event ev_key_G(k). Finally, the IoT Device completes its exchange by responding with the computed (single) hash.

3.4 Main Process

The main process initializes the system by generating the necessary keys and data, and then starts the four processes. Each entity runs in parallel (`|!()`), representing multiple simultaneous instances

```
process
  new d1_G: exponent;
  let e1_G = exp(g, d1_G) in
  new d_G: skey;
  let e_G = pk(d_G) in
  new d1_D: exponent;
  let e1_D = exp(g, d1_D) in
  new N_D: bitstring;
  new w_D: bitstring;
  new B_D: bitstring;
  (
    (!SmartApp(e_G, N_D, w_D, B_D))
    | (!SUP(e_G))
    | (!Gateway(e1_G, d1_G, e_G, d_G))
    | (!IoT_Device(e1_D, d1_D, w_D, B_D, N_D))
  )
```

3.5 Queries

The ProVerif model includes the following queries to verify the security properties of the onboarding protocol:

- **Secrecy Queries:** These queries check that the IoT device's private information (the password `w_D`, serial number `N_D`, and Bluetooth MAC address `B_D`) cannot be obtained by an attacker:

```
query attacker(w_D).
query attacker(N_D).
query attacker(B_D).
```

- **Mutual Authentication Queries:** These queries ensure that the challenge–response mechanism works properly for both the Gateway and IoT Device, by verifying that if one party concludes authentication, the corresponding start event has occurred on the other side:

```
query challenge: bitstring, w_D: bitstring;
event(end_auth_G(challenge, w_D)) ==> event(begin_auth_D(challenge, w_D)).
```

```
query challenge: bitstring, w_D: bitstring;
event(end_auth_D(challenge, w_D)) ==> event(begin_auth_G(challenge, w_D)).
```

- **Key Consistency Query:** This query ensures that if the Gateway ends up possessing the symmetric key k (event `ev_key_G`), then that key must have been generated exactly once by the IoT Device (event `ev_key_D`). The injective event requirement guarantees a one-to-one correspondence:

```
query k:key; event(ev_key_G(k)) ==> inj-event(ev_key_D(k)).
```

3.6 Query Results

This section details the results of the verification queries performed. Each query and its corresponding result (which was *true* in all cases) are presented, followed by an explanation of what each query aimed to prove about the security properties of the Gateway and IoT device interaction.

3.6.1 Query: not attacker(w_D[]) is true.

This query aimed to verify that the initial password/credential (w_D) of the IoT device is not known to the attacker. The *true* result indicates that, under the assumed security model, the attacker cannot obtain the secret password w_D through any of the modeled attack vectors. This is a fundamental security requirement for the authentication process to be reliable.

3.6.2 Query: not attacker(N_D[]) is true.

This query aimed to verify that the serial number (N_D) of the IoT device is not known to the attacker. The *true* result suggests that the attacker cannot obtain the serial number, which could potentially be used as a static identifier for tracking or targeting the device. Keeping the serial number confidential can contribute to the overall security and privacy of the IoT device.

3.6.3 Query: not attacker(B_D[]) is true.

This query aimed to verify that the Bluetooth MAC address (B_D) of the IoT device is not known to the attacker. The *true* result reinforces the assumption that the attacker does not possess this specific hardware address. Knowledge of the Bluetooth MAC address could potentially be exploited for tracking the device's physical presence or attempting Bluetooth-based attacks. Therefore, its confidentiality is an important security consideration.

3.6.4 Query: event(end_auth_G(challenge,w_D)) $\Longrightarrow$ event(begin_auth_D (challenge,w_D)) is true.

This query aimed to verify that if the Gateway successfully completes its authentication procedure *end_auth*$_G$ with the specific challenge and the correct password *w_D*, then the IoT device must have initiated its authentication procedure *begin_auth*$_D$ with the same challenge and password. The *true* result indicates a correspondence between the Gateway's successful authentication completion and the IoT device's initiation of the process, suggesting a synchronized and consistent authentication flow.

3.6.5 Query: event(end_auth_D(challenge,w_D)) $\Longrightarrow$ event(begin_auth_G (challenge,w_D)) is true.

This query aimed to verify the reverse direction of the previous one. It checks that if the IoT device successfully completes its authentication procedure *end_auth_D* with the specific challenge and the correct password *w_D*, then the Gateway must have initiated the authentication *begin_auth_G* with the same challenge and password. The *true* result confirms the reciprocal relationship in the authentication process: successful completion by one party implies the initiation by the other with the correct parameters.

3.6.6 Query: inj-event(ev_key_G(k)) $\Longrightarrow$ inj-event(ev_key_D(k)) is true.

This query aimed to verify that if the Gateway successfully injects (or establishes) a specific key k through the event *ev_key_G*, then the IoT device must have also successfully injected (or established) the same key k through the event *ev_key_D*. The $inj - event$ likely signifies that the event has occurred and the key is now established and available for secure communication. A *true* result indicates that the key agreement process is consistent, and both parties end up with the same shared key. This is crucial for secure end-to-end communication.

4 Secure Update Analysis

Below is a complete, structured analysis of the secureUpdate model. This analysis is organized into five sections: Protocol Description, Declaration of Functions, Process Macros, Queries, and Analysis of Results.

4.1 Protocol Description

The secureUpdate protocol is designed to securely deliver a new software image and manifest to an IoT device for an update. Building on the onboarding process, secureUpdate provides additional guarantees by:

- **Ensuring Freshness:** Each update carries a timestamp. The SUP (server) signs the manifest together with a new timestamp (using a concatenation function) so that the device can verify the manifest's freshness and avoid replay attacks. Two successive timestamps are checked via a reduction function (`check_succ`) and corresponding events (`freshness_ok` and `freshness_ok_1`).
- **Updating with Key Locking:** The software image not only contains the new update but also embeds the "next public verification key." This key locking mechanism ensures that for every new update, the SUP generates a public key (with `next_key_generated`) that the device later extracts (`next_key_extracted`) from the image. This one-to-one relationship is critical to ensure that the future update will be authenticated against the verified key.
- **Digital Signature Authentication:** Both the manifest and the image are digitally signed. The SUP uses its signature keys to sign the manifest (ensuring authenticity) and then signs the software image (which embeds the next key). The IoT device verifies these signatures using the SUP public key (or the previously extracted key for further updates).

- **Illustrative Two-Update Sequence:**

  To demonstrate the update mechanism, the model simulates two consecutive update operations. This simulation highlights the key-locking mechanism and the transition between verification keys:

  1. *Initial Update:* The SUP sends a manifest (M1) with a new timestamp, and subsequently sends an image (M2) that is signed and embeds the next verification key. This initial update uses the manufacturer's key pair for verification, as it's the first update the device receives.
  2. *Subsequent Update:* The SUP processes a second update by signing a new manifest with a further updated timestamp and sending another updated image that carries a new embedded key. This update uses the verification key embedded in the previous update image.

  This sequence illustrates the general update process, where each image contains the necessary verification key to authenticate the **next** update, ensuring a chain of trust.

Overall, the protocol guarantees that the IoT device accepts only fresh, authenticated updates and that the chain of keys is maintained securely via key extraction and key locking.

4.2 Declaration of Functions

The secureUpdate model reuses many cryptographic primitives from the onboarding phase but also introduces specific functions to support update semantics:

- **ECDH and Group Operations:**
  - The function `exp(G, exponent): G` models exponentiation in a cyclic group. This function, along with the equation ensuring commutativity (i.e., both parties derive the same shared secret), represents the Elliptic Curve Diffie-Hellman (ECDH) key agreement.

    It's important to note that the ECDH key agreement process, which leads to the establishment of the shared key K (`K`, `K1` in the model) between the Gateway and the IoT Device (used, for example, to encrypt the symmetric key k for Bluetooth communication), is **abstracted** in this secure update model. This process is assumed to have been established and verified in the previous **onboarding** model. The secure update model focuses on the secure distribution and verification of software updates **after** this secure channel has been established.

- **Symmetric Encryption:**
  - The functions `senc` and its corresponding decryption `sdec` use keys of type `key` so that encrypted messages (including manifests and images) are protected over the channels.

- **Asymmetric Encryption and Digital Signatures:**
  - Functions such as `aenc`/`adec` protect sensitive data, while the signing operation `sign` (and its verification via `checksign`) ensures that both the manifest and the software image are bound to the SUP's keys.

- **Converters and Embedding Functions:**
  - `image_to_bitstring` and `manifest_to_bitstring` are used to translate structured data into bitstrings for signing and verification.
  - `embed_key(image, spkey)` embeds a public verification key into a software image, and its reduction rule (using `extract_key`) ensures that the device can recover the embedded key.

- **Timestamp Operations:**
  - The function `succ(timestamp)` produces a fresh timestamp, and by using a reduction rule (with `check_succ`), it confirms that a new timestamp is indeed the successor of a previous one.
  - The function `concatManifestTimestamp(manifest, timestamp)` combines a manifest with a timestamp so that the timestamp can be later extracted (via `get_timestamp`).

These declarations form the cryptographic foundation of the secure update, providing confidentiality, integrity, and freshness guarantees.

4.3 Process Macros

The secureUpdate model is divided into three main processes simulating the behavior of the SUP, Gateway, and IoT Device (the SmartApp does not play an active role here):

- **SUP Process:** The SUP process handles the update by:
  - Receiving a metadata request from the Gateway.
  - Generating a new timestamp (`new_ts = succ(ts)`) and concatenating it with the manifest.
  - Signing the manifest (using `sign`) and sending it to the Gateway. An event `manifest_sent(manifest, timestamp)` is emitted.
  - Upon receiving a confirmation (or a request) from Gateway, the SUP then generates a new signing key for the next update. It calculates the corresponding public verification key (via `spk`) and embeds it in the software image (using `embed_key`).
  - The SUP sends the signed image to the Gateway and emits `image_sent(image)`.
  - For the *second update*, a similar series of steps occur: a new manifest is created (with a fresh timestamp) and sent, followed by generation of an additional next key, image signing, and embedding before forwarding the update.
- **Gateway Process:** Acting as a relay between the SUP and the device, the Gateway:
  - Sends the metadata request to the SUP.
  - Receives the signed manifest from the SUP and forwards it to the device (encrypted under a session key `K`).
  - Receives a confirmation from the device indicating that the manifest was accepted.
  - Requests the software image from the SUP.
  - Receives the signed software image from the SUP, then forwards it to the IoT device (again, encrypted) over a different session key (`K` for the first update or `K1` for the second).
- **IoT_Device Process:** The IoT device verifies every element of the update:
  - It decrypts the manifest and verifies the digital signature using the SUP's public key.
  - It extracts the timestamp (using `get_timestamp`) and checks its freshness using the `check_succ` function. If successful, it emits events `manifest_verified` and `freshness_ok`.
  - On successful verification, the device sends a confirmation to the Gateway.
  - Next, it receives the signed software image, decrypts it, and verifies its signature.
  - It then extracts the embedded public verification key with `extract_key` and emits the event `next_key_extracted`.
  - In the second modeled update phase, the device similarly checks the manifest for freshness and verifies the image, ensuring all new keys and timestamps follow the correct order.

4.4 Queries

The model includes several queries to ensure the security properties of the update process:

- **Secrecy Queries:**
  - These verify that sensitive data such as the IoT device password (`w_D`) and session keys (`K`, `K1`) are not available to an attacker:

    ```
    query attacker(w_D).
    query attacker(K).
    query attacker(K1).
    ```

- **Authenticity of the Manifest and Image:**
  - Two queries enforce that if a manifest or image is verified by the IoT device, then it must have been sent by the SUP:

    ```
    query m: manifest, t: timestamp;
    event(manifest_verified(m,t)) ==> inj-event(manifest_sent(m,t)).

    query i: image;
    event(image_verified(i)) ==> inj-event(image_sent(i)).
    ```

- **Freshness (Anti-Replay) of Timestamps:**
  - To prevent replay attacks, a query ensures that the freshness event (used during the second update) corresponds in an injective manner to a previous freshness check:

    ```
    query t1: timestamp, t2: timestamp;
    event(freshness_ok_1(t1, t2)) ==> inj-event(freshness_ok(t1, t2)).
    ```

- **Key Locking (Update Key Consistency):**
  - Finally, a query asserts that if the IoT device extracts a "next" key from the image, then it must have been generated by the SUP in a unique, one-to-one fashion:

    ```
    query k: spkey;
    event(next_key_extracted(k)) ==> inj-event(next_key_generated(k)).
    ```

    This query ensures the chain of trust for updates: every embedded key used for future verification is properly bound to a corresponding generation event.

4.5 Analysis of Results

Obtaining *true* for all the queries, it indicates that the model satisfies the intended security properties. A detailed interpretation of these results is given below:

- **Secrecy:**
  - The queries protecting sensitive elements (`w_D`, `K`, `K1`) being proven true confirms that the attacker cannot obtain the values of the IoT device password (`w_D`), the initial session key (`K`) established during onboarding, or the subsequent session key (`K1`) used for the second update. This means that these secret values remain confidential and are not leaked during the update process, protecting the communication and the device's credentials.

- **Authenticity of the Manifest and Image:**

- The manifest and image authenticity queries being proven true confirms that every verified artifact (manifest and software image) was indeed sent by the SUP. This means that the IoT device only accepts manifests and images that are verifiably from the trusted SUP, thanks to the digital signatures. This prevents the device from installing updates from unauthorized sources, ensuring integrity and origin authenticity.

- **Freshness of Timestamps:**
  - The freshness query being proven true guarantees that the timestamp associated with each update is strictly increasing. This confirms that the protocol effectively prevents replay attacks, where an attacker might try to re-send an older manifest or image to trick the device into installing a previous version. Each update is verified to be fresh and not a replay of a previous transmission.

- **Key Locking:**
  - The key locking query being proven true enforces that every extracted ”next” key (used to verify future software images) corresponds uniquely to a generation event by the SUP. This demonstrates that the update chain remains unbroken. The device always receives a valid next key embedded in the image, and this key is verifiably tied to a key generated by the SUP. This ensures the continuity of trust in the update chain, where each update's authenticity is linked to the previous valid update.

5 Perfect Forward Secrecy Analysis

5.1 Motivation and design limitation

The original specification by Gupta and van Oorschot ([2], Sect. *Limitations*, item L5) explicitly states that the Elliptic-Curve Diffie–Hellman (ECDH) key pairs $(e1_G, d1_G)$ and $(e1_D, d1_D)$ are *not* ephemeral: they are generated once at initialisation or device reset and reused afterwards. Consequently, the long-term shared secret $K_{ECDH} = \exp(e1_D, d1_G)$ is constant for the whole device lifetime, so Perfect Forward Secrecy (PFS) is not guaranteed *a priori*. Our goal is twofold:

1. **Confirm** the absence of PFS in the *Onboarding* phase when the long-term exponent leaks.
2. **Verify** that, assuming K_{ECDH} itself remains secret, the two symmetric session keys k (first update) and k_1 (second update) *do* enjoy PFS in the *Secure Update* phase.

5.2 Onboarding – PFS violation

Experimental model. We cloned the original onboarding file and added a single process that leaks the Gateway's exponent in a new phase:

```
... | phase 1; out(c2,d1_G)
```

All other roles run unchanged in phase 0, so at least one session key k is negotiated before the leak.

Query.

```
query attacker(k).   % is k recoverable after the leak?
```

Result. ProVerif outputs `RESULT not attacker(k_1) is false`. The attacker records the ciphertext `senc_key(k,K_ECDH)` in phase 0, obtains `d1_G` in phase 1, reconstructs K_{ECDH} and decrypts the ciphertext, thus learning k. **Conclusion:** onboarding fails to provide PFS with static ECDH keys, confirming the limitation already mentioned in the original paper.

5.3 Secure Update – PFS preserved

Experimental model. In `secureUpdate_PFS.pv` we abstract K_{ECDH} as uncompromised and make the second session key `k1` public to the attacker:

```
free k1: key             % dropped the [private] attribute
```

Queries.

```
query attacker(k1).        % sanity check: attacker knows k1
query attacker(image0).    % can image0 be decrypted?
```

Results. `attacker(k1)` is (expectedly) `true`.
The main query returns `RESULT not attacker(image0[]) is true`: the attacker, even with full knowledge of k_1, cannot derive the first-update key k nor decrypt `image0`, which is protected with k.

Conclusion. Provided that K_{ECDH} is never revealed, the update protocol maintains PFS across successive symmetric keys.

Summary of findings

Table 1: PFS outcome in the two protocol phases.

Scenario	Key / Data	PFS Result
Onboarding (static ECDH)	k	**Broken**
Secure Update (leak of k_1)	`image0`	**Preserved**

These experiments formally support the informal claim in [2]: PFS is absent if ECDH keys are reused, but, once K_{ECDH} is assumed uncompromised, the key-locking mechanism successfully isolates past software images from future key disclosure.

6 Conclusion

This report presented the chain-of-custody protocol designed for secure onboarding and software updates of IoT devices, particularly focusing on devices with limited resources. The protocol leverages elliptic curve cryptography, authenticated key exchange, and a key-locking mechanism to address the challenges of secure communication and update delivery in constrained environments.

The onboarding process establishes a secure channel between the IoT device and the Gateway, using ECDH to derive a shared key and protect subsequent communication. This process ensures that only authorized devices can participate in the network.

The secure update protocol builds upon this foundation by providing mechanisms for secure software distribution. Digital signatures ensure the integrity and authenticity of updates, timestamps prevent replay attacks, and the key-locking mechanism maintains a chain of trust between consecutive updates.

The formal verification of the onboarding and secure update protocols using ProVerif provides assurance that the protocols satisfy the described security properties. The queries defined in ProVerif and their successful verification demonstrate that:

- Sensitive information, such as device credentials and session keys, remains confidential.
- The authentication processes for onboarding and updates are robust and prevent unauthorized access.

- Software updates originate from the trusted Software Update Provider and are not tampered with.
- The update process prevents replay attacks by ensuring the freshness of updates.
- The key-locking mechanism guarantees the continuity of trust in the update chain.

In summary, the analyzed protocol and its formal verification contribute to enhancing the security of small IoT device ecosystem. The combination of secure onboarding and secure update mechanisms provides an useful foundation for managing and maintaining IoT devices securely throughout their lifecycle.

References

[1] Saad El Jaouhari, Eric Bouvet ”Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions”

- **URL:** `https://www.sciencedirect.com/science/article/abs/pii/S2542660522000142?via%3Dihub`

[2] G. Gupta and P. van Oorschot, ”Onboarding and Software Update Architecture for IoT Devices” *IEEE Access*, vol. 7, pp. 182761-182786, 2019, doi: 10.1109/ACCESS.2019.2958934.

- **URL:** `https://ieeexplore.ieee.org/document/8949023`
- **URL:** `https://people.scs.carleton.ca/~paulv/papers/PST2019-gupta.pdf`