FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Design Document Mobile Embedded Security

Miguel Arcilla
Chiara Fedrizzi
Kilian Hnidek
Viktor Nawrata
Stefan Ohnewith
Ernst Schwaiger

Contents

1 Project description

This section provides a brief overview of the project, covering the basic idea with simple sketches for visualization.

1.1 Problem description

The aim of the project is to develop a system which allows to distribute newer versions of a piece of firmware onto a number of client nodes. The firmware update is created on a *Build System*, then transferred to a *Firmware Management Server*, which subsequently sends packages containing the updated firmware to a number of *Client* nodes which are installing the updates.

Figure 1 displays an overview of the system. A self-signed *Root/CA* certificate is used for signing a *Build_Signing* certificate which is stored together with the private key on a crypto token. When a new firmware update package is built, it is signed using that key to prove its authenticity to the *Firmware Management Server* which is receiving that package via ssh/scp.

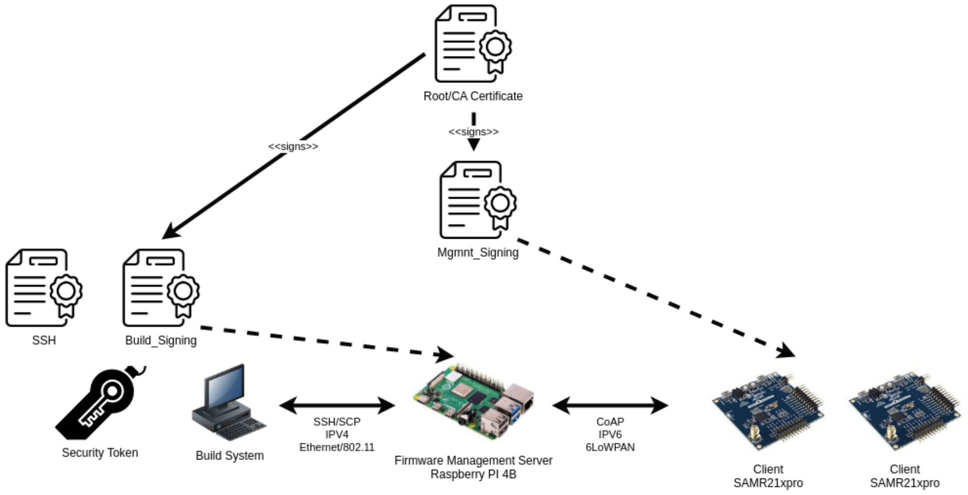

Figure 1: System Description

Process: A service on the *Firmware Management Server* detects a newly arrived firmware update, verifies the validity of the *Build_Signing* certificate and the validity of the signature. The service puts the received firmware into a SUIT update package enclosing all required meta data, signs the package using its private signing key (of the *Mgmnt_Signing* certificate), then generates a session AES key, which it uses to encrypt the SUIT update package. For each *Client* node it then encrypts the AES key using the *Client* public key and stores it along with the package signature.

The encrypted SUIT update package, along with the encrypted AES key and the SUIT signature is then sent to all *Client* nodes via *6LoWPAN*.

On the client system, both the encrypted firmware and the encrypted key and signature arrive. The client verifies the validity of the *Mgmnt_Signing* certificate, then the signature of the received package using the certificate. If the verification was successful, the client first decrypts the AES key using its private key, then it decrypts the package using the AES key. In the next step, the client can install/flash the firmware update.

For the handling of certificates that either will expire soon or have been compromised, a second process takes care of replacing/revoking them on the respective Build System and *Client* nodes.

2 IoT Security

This section provides details on how the firmware update is secured.

2.1 Secure Firmware Update Over-The-Air (SFOTA)

Before it is possible to have a SFOTA process, it is important to define on what grounds e.g. network layout and cryptographic primitives the update is based upon.

Network: The network will be a locally run IPv6 network created with the support of RIOT-OS' tool "Ethernet over Serial Driver" (ethos). The tools helps creating a network instance with a custom interface name and a defined prefix. In our case this project's configuration may look like this:

- Interface: **riot0**
- Prefix: **2001:db8::/64**

Note: Although, the devices are serially connected to the device hosting the IPv6 network this way, the update still remains over-the-air as IPv6 packets are exchanged.

Encryption and signature: The firmware must be encrypted and signed on the firmware management server and later decrypted by the client devices that process and execute the update. The encryption and signature for the firmware on the management server is done via a Python script with the library PyCryptodome[1].

The decryption and signature on the client nodes is implemented using the existing implementations of RIOT-OS' Crypto library. Due to its simplicity, the algorithm AES-CBC is chosen. AEAD ciphers were considered, however, in this case, it is not necessary as integrity is provided by signatures.

The signature algorithm used is Ed25519 since both RIOT-OS and PyCryptodome support it. Table 1 shows what data is exchanged from the management server and the client nodes and which are either plaintext, encrypted and/or signed.

SUIT: To update the firmware, the SUIT procedure is used where the signed manifest and the encrypted and signed firmware are uploaded in the firmware management server. The server triggers a notification and the client nodes begin the update procedure by fetching the manifest first.

The client nodes use the manifest to execute the necessary steps to fetch the image, validate, decrypt and finally flash the device with the new firmware image. Fetching the necessary data such as the SUIT manifest or the encrypted firmware relies on CoAP with the library nanoCoaP.

The SUIT manifest will be extended by adding steps to decrypt the firmware and session AES key as well as validations. The encrypted session key resides in the SUIT manifest.

Figure 2 shows how an overview how the update procedure is done via notification.

[1] https://pycryptodome.readthedocs.io/en/latest/src/introduction.html (Accessed 16-12-2025)

Data sent to client	**Protected**
Trigger notification by FW management server	Plaintext
SUIT Manifest	Plaintext, Signed
Firmware image	Encrypted, Signed
Session AES key (Within Manifest)	Encrypted, Signed

Table 1: Data transferred between FW Management server and Client nodes

Category	**Main Risk**	**Mitigation**
Spoofing	Spoofed update server	Signed SUIT manifest
Tampering	Modified SUIT manifest	Signed SUIT manifest
	Modified firmware image	End-to-end encryption
Repudiation	Server denies update action	Audit logs
	Device denies update action	Audit logs
Info disclosure	SSH interception	Strong SSH keys, host verification
Denial of Service	Fake update notifications	Not handled
Elevation of privilege	Root Certificate compromise	Not handled

Table 2: STRIDE Threat Model

STRIDE: Potential threats need to be considered when implementing this update process. Therefore, table 2 presents some main risks across the STRIDE categories and mitigations that are done in the implementation. However, some risks were identified that are not going to be handled due to complexity and time limitations.

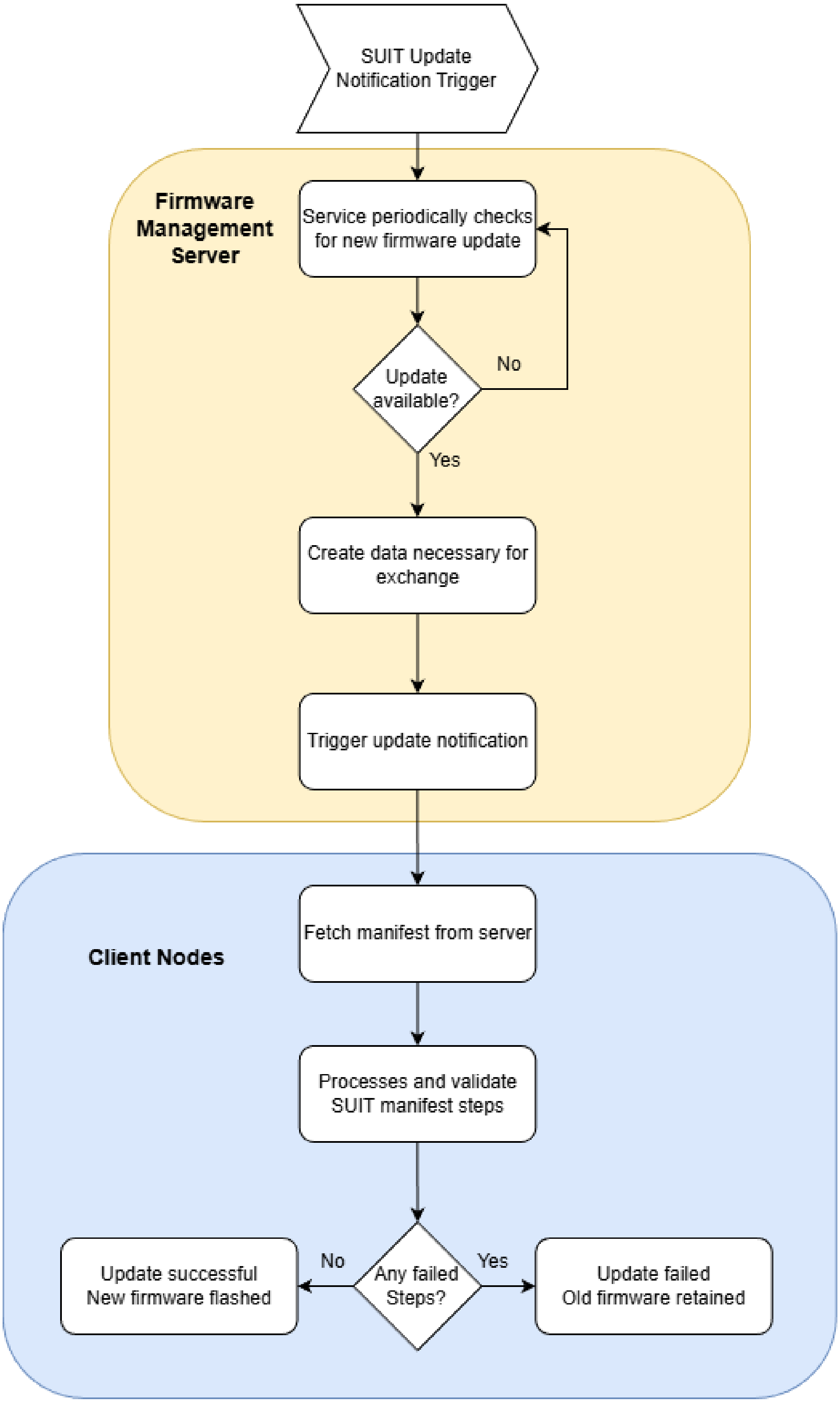
SUIT Update Notification Trigger
Firmware Management Server
Service periodically checks for new firmware update
Update available?
No
Yes
Create data necessary for exchange
Trigger update notification
Client Nodes
Fetch manifest from server
Processes and validate SUIT manifest steps
Update successful New firmware flashed
No
Any failed Steps?
Yes
Update failed Old firmware retained

Figure 2: Overview - SFOTA process with SUIT

3 Key Management

The key management section defines which cryptographic keys exist, how these are generated, stored, distributed, and how the revocation and issuance are managed. The main focus is on ensuring that only authentic firmware updates are accepted and that confidentiality of these images and their respective update keys are preserved.

The key management design distinguishes between the following main actors:

- Root Certification Authority (Root CA)
- Build System
- Firmware Management Server
- Client Nodes

The following subsections cover the topics of key types, key generation, key storage, key distribution and ending with the management along their lifecycle-span.

3.1 Key Types and Roles

The defined system will be limited to a small set of well defined key types, with clearly assigned purpose and owner for each key type.

Root Certification Authority key pair

- Type: Asymmetric key pair for a Certification Authority, RSA.
- Owner: Root Certification Authority, operated offline.
- Role:
  - Signs the Build_Signing certificate that is used on the Build System.
  - Signs the Mgmnt_Signing certificate that is used on the Firmware Management Server.
  - Signs the Client_Encryption certificates that bind each client public key to a client identity.
- Properties:
  - Self-signed and never used directly for firmware signatures or encryption.
  - Stored offline and only used to issue or revoke subordinate certificates.
  - Acts as the single trust anchor for certificate validation in the system.

Build System signing key pair (Build_Signing)

- Type: Asymmetric signing key pair, RSA.
- Owner: Build System.
- Role:
  - Signs firmware update packages that are produced by the Build System before they are transferred to the Firmware Management Server.
  - Proves that a firmware update originated from an authorized maintainer.
- Properties:
  - The private key is created and stored on a hardware crypto token as described in Section 4.1.
  - The private key never leaves the crypto token and cannot easily be extracted by an attacker. The used hardware crypto token, however, only supports the creation of RSA key pairs.
  - The corresponding certificate is signed by the Root CA and installed on the Firmware Management Server.
  - The certificate is used on the server side to verify signatures on received firmware update packages.

Firmware Management Server signing key pair (Mgmnt_Signing)

- Type: Asymmetric signing key pair, Ed25519.
- Owner: Firmware Management Server.
- Role:
  - Signs SUIT update packages and the associated data that is transmitted to the clients, including the pair (Q_E, C) from the ECIES encryption of the session key K_{enc} (see Section 4.4).
  - Provides integrity and authenticity of firmware updates from the Firmware Management Server towards the Client Nodes.
- Properties:
  - The corresponding certificate is signed by the Root CA.
  - The certificate or its public key is installed on all Client Nodes as a trust anchor for firmware updates.
  - The private key is kept in a protected key store on the Firmware Management Server.

Client node encryption key pair and certificate (Client_Encryption)

- Type: Asymmetric key pair for Elliptic Curve Diffie-Hellman based on Curve25519 (X25519) supported by the RIOT-OS cryptographic library C25519. The key pair is denoted (d_R, Q_R) in Section 4.4.
- Owner: Individual Client Node.
- Role:
  - Acts as the long term receiver key pair for the ECIES scheme used to protect the Advanced Encryption Standard session key K_{enc} that encrypts the firmware image.
  - Provides the receiver public key Q_R that the Firmware Management Server uses for ECIES encryption of K_{enc}.
- Properties:
  - The private key d_R remains on the Client Node and is never sent over the network.
  - The public key Q_R is bound to the client identity by an X.509 certificate (Client_Encryption) signed by the Root Certification Authority.
  - The Firmware Management Server stores the Client_Encryption certificate for each client and extracts Q_R from that stored certificate when performing ECIES.
  - The Client Node stores the private key d_R and, optionally, its own Client_Encryption certificate for identification and consistency checks.

Ephemeral ECIES key pairs

- Type: Ephemeral Elliptic Curve key pair (d_E, Q_E) generated for each client and for each firmware update, as described in Section 4.4.
- Owner: Firmware Management Server.
- Role:
  - Used to derive a per client shared secret $S = d_E \cdot Q_R$ from which the wrapper key K_{wrap} is derived by HKDF.
  - Provides forward secrecy for K_{enc} with respect to a passive attacker observing the network.
- Properties:
  - Generated freshly for each update and each client.
  - Private part d_E is not stored persistently and is erased after completion of the encryption step.

Symmetric session keys

The system uses symmetric keys only as short lived session keys.

- AES firmware encryption key K_{enc}:
  - Type: Symmetric key for the Advanced Encryption Standard in Cipher Block Chaining mode.
  - Owner: Firmware Management Server during update processing; Client Nodes after decryption.
  - Role:
    - Encrypts the SUIT firmware package before transmission.
    - Prevents unauthorized parties from learning the firmware content from intercepted traffic.
  - Properties:
    - Generated randomly for every new firmware update version.
    - Not reused across different firmware versions.
- Wrapper key K_{wrap}:
  - Type: Symmetric key derived from the shared secret S using HKDF as described in Section 4.4.
  - Owner: Firmware Management Server and corresponding Client Node.
  - Role:
    - Encrypts and decrypts the AES session key K_{enc}.
  - Properties:
    - Not stored persistently.
    - Recomputed by both parties when needed during the update protocol.

3.2 Key Generation

Key generation must use cryptographically secure random numbers and must be reproducible as a process, but not in terms of output. The following generation procedure is used.

Root Certification Authority and signing certificate keys

The Root Certification Authority and the two signing key pairs are generated once during system initialization.

1. Generate a Root Certification Authority RSA key pair on a secure, offline machine.
2. Create a self-signed Root Certification Authority certificate that identifies the authority and contains the Root Certification Authority public key.

3. On the Build System, create an RSA key pair for the Build_Signing certificate on a hardware crypto token. Export only the public key.
4. On the Firmware Management Server, generate an Ed25519 key pair for the Mgmnt_Signing certificate. Export only the public key.
5. On the Root Certification Authority machine, issue two certificates: one for the Build_Signing key and one for the Mgmnt_Signing key. Both certificates are signed with the Root Certification Authority key.
6. Install the Root Certification Authority certificate and the Build_Signing certificate on the Firmware Management Server.
7. Install the Root Certification Authority certificate and the Mgmnt_Signing certificate on all Client Nodes.

Client node encryption key pairs and certificate issuance

Client encryption keys are generated during provisioning of each device using a trusted provisioning channel. The Firmware Management Server stores a client certificate for each client in order to obtain the public key for ECIES encryption.

1. A provisioning station (for example the Build System in a controlled environment) generates a Curve25519 compatible key pair (d_R, Q_R) for the Client Node.
2. The private key d_R is installed onto the Client Node during provisioning and stored in non volatile memory. The private key is marked as secret and is never transmitted during normal operation.
3. The public key Q_R, together with a device identifier, is transferred to the Firmware Management Server and used to request certificate issuance.
4. The Root Certification Authority issues an X.509 Client_Encryption certificate that contains Q_R as subject public key and signs this certificate with the Root Certification Authority private key.
5. The Firmware Management Server stores the issued Client_Encryption certificate in a database entry that is associated with the identity of the Client Node.
6. The issued Client_Encryption certificate is optionally transferred to the Client Node over the trusted provisioning channel and stored on the Client Node for traceability and consistency checks.

Session key K_{enc} and ephemeral ECIES keys

For each firmware update version the Firmware Management Server generates a new AES session key K_{enc} as part of the processing described in Section 4.2.

1. When a new firmware update arrives and has been verified, the Firmware Management Server generates a random AES key K_{enc} of suitable length, for example one hundred and twenty eight bits. The key is derived from a secure random source provided by the operating system, for example `/dev/urandom`.

2. For each Client Node that will receive the update, the Firmware Management Server generates a fresh ephemeral ECC key pair (d_E, Q_E).

3. For each Client Node the Firmware Management Server loads the stored Client_Encryption certificate for that client, extracts the public key Q_R from the certificate, computes the shared secret $S = d_E \cdot Q_R$, derives K_{wrap} from S via HKDF, and encrypts K_{enc} to obtain C, as described in Section 4.4.

4. After completion of the encryption and storage of (Q_E, C), the ephemeral private keys d_E are securely erased from memory.

3.3 Key Storage

Secure key storage is essential because compromise of long term keys undermines the entire update scheme. The storage concept is separated by component.

Build System

- Build_Signing private key:
  - Stored exclusively on a hardware crypto token.
  - The token is protected by a personal identification number that is known only to the person responsible for building and signing firmware.
  - The private key never leaves the token in plain form. All signature operations are executed on the token.
- Root CA certificate:
  - Stored as a trust anchor file for verification on the Firmware Management Server, not on the Build System itself.
  - Optionally stored read only on the Build System for documentation.

Firmware Management Server

- Mgmnt_Signing private key:
  - Stored in a protected key store on the server host.
  - Access limited to the firmware management service process, with restrictive file system permissions.
  - Optionally additionally protected by a software based key encryption mechanism that requires a passphrase at service start.

- Root Certification Authority certificate and subordinate certificates:
  - Stored in a dedicated directory that is read only for the firmware management process.
  - Used to verify the Build_Signing certificate during the verification of incoming firmware updates.
- Client certificates (Client_Encryption):
  - Stored in a database or configuration store that maps each Client Node identity to its Client_Encryption certificate.
  - The Firmware Management Server extracts Q_R from the stored certificate when performing ECIES encryption of K_{enc}.
  - Access controlled so that only the firmware management process can read this information.
- Session keys:
  - K_{enc}, K_{wrap}, and ephemeral private keys d_E are kept only in volatile memory for the duration of an update processing run.
  - These keys are overwritten in memory as soon as no longer needed and are not written to persistent storage.

Client Nodes

- Long term private key d_R:
  - Stored in non volatile memory that is not externally readable over the normal communication interfaces.
  - Access restricted to the firmware update component that performs ECIES decryption.
  - If supported by the hardware, placed in a dedicated secure storage region.
- Public keys and trust anchors:
  - Root CA certificate and the Mgmnt_Signing certificate or its public key are stored in read only memory together with the firmware or in a dedicated configuration section.
  - Updates to these certificates are only accepted as part of a signed firmware update process.

3.4 Key Distribution and Provisioning

Key distribution defines how keys reach the respective parties in a trustworthy way.

Initial provisioning phase

During initial provisioning the following steps take place.

1. The Root Certification Authority certificate is embedded into the Client Node firmware image or written into a dedicated read only configuration partition.
2. The Mgmnt_Signing certificate is installed on the Client Nodes as part of the initial firmware image.
3. Each Client Node gets their encryption key pair (d_R, Q_R) as described in Section 3.1.
4. The public key Q_R, together with an identifier for the Client Node, is transmitted over a trusted provisioning channel to the Firmware Management Server.
5. The Firmware Management Server issues a Client_Encryption certificate containing Q_R, signed by the Root Certification Authority.
6. The Firmware Management Server stores the Client_Encryption certificate associated with the device identity in its client database.
7. The issued Client_Encryption certificate is transferred to the Client Node over the trusted provisioning channel and stored on the Client Node.

Distribution during firmware updates

During a firmware update, the following key related data is transferred.

1. The Build System sends the firmware update package and the detached signature generated with the Build_Signing key to the Firmware Management Server via Secure Shell.
2. The Firmware Management Server verifies the signature using the Build_Signing certificate that is already installed on the Firmware Management Server and validates that certificate against the Root Certification Authority certificate.
3. After verification and after packaging of the firmware into a SUIT update, the Firmware Management Server encrypts the SUIT firmware with K_{enc} and encrypts K_{enc} for each Client Node as (Q_E, C) using ECIES. For ECIES, the Firmware Management Server extracts each client public key Q_R from the stored Client_Encryption certificate.
4. The Firmware Management Server signs the SUIT package and, per Client Node, the pair (Q_E, C) with its Mgmnt_Signing key.
5. The Client Node receives the encrypted firmware package, the pair (Q_E, C), and the signature. The Client Node verifies the signature against the stored Mgmnt_Signing public key, computes K_{wrap} using its private key d_R and the received Q_E, recovers K_{enc}, and decrypts the firmware.

3.5 Key Lifetime, Rotation, and Revocation

Key lifetime management ensures that cryptographic material does not remain in use beyond reasonable time frames and that compromised keys can be withdrawn from service.

Key lifetime and rotation

The following lifetimes are proposed.

- Root CA key pair:
  - Long lifetime, for example ten years.
  - Only used to sign subordinate certificates and to issue revocations.
- Build_Signing key pair:
  - Medium lifetime, for example two to three years.
  - When rotated, a new Build_Signing key pair and certificate are generated and signed by the Root CA.
  - The Firmware Management Server is updated to trust both the old and the new Build_Signing certificate for a transition period.
- Mgmnt_Signing key pair:
  - Medium lifetime similar to the Build_Signing key pair.
  - When rotated, the new certificate is distributed to Client Nodes as part of a signed firmware update.
  - The SUIT manifest can carry an identifier of the signing key to support coexistence of old and new keys during transition.
- Client node key pairs (d_R, Q_R):
  - Long lifetime constrained by the physical lifetime of the device.
- Symmetric session keys K_{enc} and K_{wrap}:
  - Very short lifetime.
  - Created per firmware version and per client interaction and discarded immediately after use.

Revocation and compromise handling

The scheme needs simple but practical steps to handle key compromise.

- Compromise of Build_Signing key:
  - The Root CA issues a revocation record for the affected Build_Signing certificate.

  - The Firmware Management Server is updated to reject firmware packages that are signed with the revoked certificate.
  - A new Build_Signing key and certificate are generated and installed.

- Compromise of Mgmnt_Signing key:

  - A new Mgmnt_Signing key pair and certificate are generated and signed by the Root CA.
  - A special emergency firmware update is created that is signed with both the old and the new Mgmnt_Signing key. The clients still trusting the old certificate install the update and record the new certificate as trusted.
  - After this transition, further updates are signed only with the new Mgmnt_Signing key.

- Compromise of a Client Node key d_R:

  - The Firmware Management Server marks the corresponding Client Node as compromised and stops sending encrypted updates to this device under the old key.
  - If physical access to the device is possible, a re provisioning process can install a new key pair and restore operation.
  - If re provisioning is not possible, the device is considered permanently untrusted.

3.6 Summary

This proposed key management design has a complete set of keys with clearly separated roles between the individual components at play. Long term signing keys are protected by hardware tokens or protected key stores. Client Nodes hold long term Elliptic Curve key pairs and trust anchors for the Firmware Management Server. Short lived session keys provide confidentiality for firmware images and are generated freshly for each update. Key rotation and simple revocation rules complete the scheme and provide a realistic and implementable basis for the Secure Firmware Update Over The Air process described in Sections 2 and 4.

4 Firmware Management

4.1 Build System

Generation of a firmware update and testing A firmware update, either triggered by a necessary bug fix or by a new software feature, starts by adding the necessary changes to the source code, then executing a test suite on the changed code. For bug fix releases, regression tests are introduced which prove that the bug is not present any more and which prevent the bug from re-entering the code base in future code changes. For new features, the test suite is extended to prove that the new feature is working.

Build and sign firmware update package to Firmware Management Server After the code changes have been applied and all tests are passing, the revision in the source code control system is tagged, and the binary, along with its version meta data is built. In the context of this project, the context consists of a three-component version number `x.y.z` and a date `YY-MM-DD`. In the next step, the binary and meta data are put into a binary archive which is in turn signed by using a dedicated key on a security token. That key belongs to a signing certificate which has been signed by a root certificate installed on the Firmware Management server, for details, refer to chapter 3. The person in charge for security updates holds that token and plugs it in to the Build System when a firmware update is to be delivered. For accessing the key on the token, a PIN must be entered which is only known to the person holding the token. This ensures that every firmware update package can be attributed to the (trusted) person holding the security token.

Deliver update package to Firmware Management Server The firmware update is delivered to the Firmware Management Server using `ssh/scp` together with the signature and the certificate. The mutual trust between the user on the Build System and the Firmware Management Server is established as follows: The user trusts the servers host key when setting up the ssh connection for the first time (Trust on First Use/TOFU). The Firmware Management Server trusts the user due to an installed public SSH key for which the user holds the corresponding private key. That private key can be installed on the same security token as the one for signing the firmware update package, or on a different token in case the tasks of signing and transferring the update is done by different persons.

4.2 Firmware Management Server

Processing the update package on the Firmware Management Server On the Firmware Management Server, a process is cyclically polling for firmware update packages. Once a package arrives it executes the following steps

1. Verification of certificate against root certificate
2. Verification of signature using verified certificate
3. Generation of Manifest file of firmware update using the enclosed meta data

4. Packaging firmware update and manifest for delivery via `SUIT`

5. Generation of AES key K_{enc} for encryption of firmware update package

6. Encryption of firmware package using the AES key K_{enc}

7. Store encrypted firmware package on file system with meta data

8. For every client: Create ephemeral ECC key pair for encrypting K_{enc}, create (Q_E, C), see 4.4

9. For every client: Sign firmware package and (Q_E, C) using Firmware Management Server private key

10. For every client: Store (Q_E, C) and signature on the file system

4.3 Client Node

Chapter 2 describes how the client nodes will obtain firmware updates.

4.4 Encryption Scheme for K_{enc}

For the encryption of the AES key K_{enc}, which was used to encrypt the firmware update, the ECIES encryption scheme is used, which is built on top of a private/public ECC key pair (d_R, Q_R) of the receiving client. The scheme consists of the following steps:

1. The sender generates an ephemeral ECC key pair (d_E, Q_E)

2. The sender generates a shared secret $S := (d_E * Q_R)$

3. The sender generates a wrapper key $K_{wrap} := HKDF(S)$

4. The sender encrypts K_{enc} using $C := E(K_{wrap}, K_{enc})$

5. The sender sends public ECC and encrypted AES key (Q_E, C) to the receiver

6. The receiver computes the shared secret $S := d_R * Q_E$

7. The receiver generates the wrapper key like the sender

8. The receiver decrypts K_{enc} using $K_{enc} := D(K_{wrap}, C)$