Uptane Standard for Design and Implementation

Uptane Standard for Design and Implementation
uptane-standard-design

Abstract

This document describes a framework for securing ground vehicle software update systems.

Table of Contents

1. Introduction

Uptane is a secure software update framework for ground vehicles. This document describes procedures to enable programmers for OEMs and suppliers to securely design and implement this framework in a manner that better protects connected units on ground vehicles. Integrating Uptane as outlined in the sections that

follow can reduce the ability of attackers to compromise critical systems. It also assures a faster and easier recovery process when a compromise occurs.

These instructions delineate the set of requirements necessary for specific ECU implementations to satisfy all conformance stipulations of the Uptane Standard. ISO/IEC 13210:1999 Information Technology, as cited in the ISO Online Browsing Platform defines a "conformance requirement" as "a requirement stated in a *base standard* that identifies a specific requirement in a finite, measurable, and unambiguous manner. A *conformance requirement* by itself or in conjunction with other conformance requirements corresponds to an *assertion.*" Individual implementers can make their own technological choices within those requirements. This flexibility makes Uptane adaptable to the many customized update solutions used by manufacturers. If implementers wish to have compatible formats, they can use POUFs. POUFs contain a description of implementation choices as well as data binding formats. An implementer who adopts a POUF, as well as the Uptane Standard, will be able to interoperate with other implementations using that POUF.

2. Terminology

With the exception of the Conformance terminology and Uptane role terminology presented below, please refer to the glossary in the Deployment Best Practices volume for definitions of all terms used in this Standard.

2.1. Conformance terminology

The keywords REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in [RFC2119]. Given the importance of interpreting these terms correctly, we present these definitions here. Note that when referring to actions in the Standard that mandate conformance, the word SHALL will be used, rather than the word MUST.

SHALL This word or the term "REQUIRED" mean that the definition is an absolute requirement of the specification. *SHALL NOT* This phrase means that the definition is an absolute prohibition of the specification. *SHOULD* This word or the adjective "RECOMMENDED" mean that, in particular circumstances, there could exist valid reasons to ignore a particular item, but the full implications will be understood and carefully weighed before choosing a different course. *SHOULD NOT* This phrase or the phrase "NOT RECOMMENDED" mean that there could exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications will be understood and the case carefully weighed before implementing any behavior described with this label. *MAY* This word or the adjective "OPTIONAL," mean that an item is truly optional.

In order to be considered conformant to the Uptane Standard, an implementation SHALL follow all of these rules as specified in the document.

Note that, following the recommendations of [RFC2119], imperatives of the type defined here will be used only when essential for security.

2.2. Uptane role terminology

These terms are defined in greater detail in Section 5.1.

Delegation: A process by which the responsibility of signing metadata about images is assigned to another party.

Role: A party (human or machine) responsible for signing a certain type of metadata. The role controls keys and is responsible for signing the metadata entrusted to it with these keys. The roles mechanism of Uptane allows the system to distribute signing responsibilities so that the compromise of one key does not necessarily impact the security of the entire system.

- *Root role*: Signs metadata that distributes and revokes public keys used to verify the Root, Timestamp, Snapshot, and Targets role metadata.
- *Snapshot role*: Signs metadata that indicates which images the repository has released at the same time.
- *Targets role*: Signs metadata used to verify the image, such as cryptographic hashes and file size.
- *Timestamp role*: Signs metadata that indicates if there are any new metadata or images on the repository.

2.3. Acronyms and abbreviations

CDN: Content Delivery Network

ECUs: Electronic Control Units, the computing units on a vehicle

LIN Bus: Local Interconnect Bus

OBD: On-board diagnostics

SOTA: Software Updates Over-the-Air

UDS: Unified Diagnostic Services

VIN: Vehicle Identification Number

3. Rationale for and scope of the Uptane Standard

This Standard document provides the essential components for the secure design, implementation, and deployment of Uptane by OEMs and suppliers. These

guidelines contribute to compromise resilience, or the ability to minimize the extent of the threat posed by any given attack.

However, this specification is intended as an implementation guide, and not as a detailed technical argument about the security properties that Uptane provides. Readers interested in such documentation can refer to published papers that cover this topic. [UPTANEESCAR]

3.1. Why Uptane requires a standards document

A standards document that can guide the safe design, integration, and deployment of Uptane in vehicles is needed at this time because:

- The number of connected units on the average vehicle continues to grow, with mainstream cars now containing up to 100 million lines of code. [USATODAY]
- The expanded use of software over-the-air strategies creates new attack surfaces for malicious parties. [CR-OTA]
- Legacy update strategies, such as SSL/TLS or GPG/RSA, are not feasible for use on vehicle ECUs because they force manufacturers to chose between enhanced security and customizability.
- Conventional strategies are also complicated by the differing resources of the ECUs, which can vary greatly in memory, storage space, and Internet connectivity.
- The design of Uptane makes it possible to offer improved design flexibility without sacrificing security.
- This added design flexibility, however, could be a liability if the framework is implemented incorrectly.
- Standardization of crucial steps in the design, implementation, and use of Uptane can assure that customizability does not impact security or functionality.

3.2. Scope of Standard coverage

This document sets guidelines for implementing Uptane in most systems capable of updating software on connected units in ground vehicles, including passenger vehicles, light-duty trucks, heavy-duty trucks, and motorcycles. Uptane could potentially also be applied to other ground vehicles, such as automated shuttles, recreational vehicles, and military ground vehicles. Uptane could even be applied to domains such as IoT devices, medical devices, and autonomous aerial vehicles. In this section, we define the scope of that applicability by providing sample use cases and possible exceptions, aspects of software update security that are not applicable to Uptane, and the design requirements governing the preparation of these standards.

3.2.1. Assumptions

We assume the following system preconditions for Uptane:

- Vehicles have the ability to establish connectivity to required backend services. For example, this could be done through cellular, Wi-Fi, or hard-wired mechanisms.
- ECUs are either directly connected to the communication channel, or are indirectly connected via some sort of network gateway.
- ECUs are programmable and provide sufficient performance to be updated.
- ECUs SHALL be able to perform public key cryptography operations and calculate hashes of images and metadata files.
- There are state-of-the-art secure servers in place, such as the Director and Image repository servers.

It is important that any bugs detected in Uptane implementations be patched promptly. Failure to do so could interfere with the effectiveness of Uptane's operations.

3.2.2. Use cases

The following use cases provide a number of scenarios illustrating the manner in which software updates could be accomplished using Uptane.

3.2.2.1. OEMs initializing Uptane at the factory using SOTA

An OEM plans to install Uptane on new vehicles. This entails the following components: code to perform full and partial verification, the latest copy of the relevant metadata, the public keys, and an accurate attestation of the latest time. The OEM then either requires its tier-1 suppliers to provide these materials to the suppliers' assembly lines or can choose to add the materials later at the OEM's assembly lines. The OEM's in-vehicle implementation is Uptane-conformant if:

1. all Primaries perform full verification;
2. all Secondaries that are updated via OTA at least perform partial verification; and
3. any ECUs that do not perform any type of verification cannot be updated via OTA.

3.2.2.2. Updating one ECU with a complete image

A tier-1 supplier completes work on a revised image for an electronic brake control module. This module will control the brakes on all models of an SUV produced by the OEM mentioned above. Assuming supplier delegation is supported by the

OEM for this ECU, each tier-1 supplier digitally signs the image, then delivers the signature, all of its metadata, including delegations, and associated images to the OEM. The OEM adds these metadata and images to its Image repository, along with information about any dependencies and conflicts between this image and those for other ECUs used in the OEM's vehicles. The OEM also updates the inventory database, so that the Director repository can instruct the ECU on how to install these updated images.

3.2.2.3. Updating individual ECUs on demand

An OEM has issued a recall to address a problem with a keyless entry device that has been locking people out of their cars. The OEM prepares an updated flash image in the manner described above. The OEM then ships USB flash drives to vehicle owners and dealerships that allow those parties to update the firmware of their vehicles.

3.2.2.4. Update one ECU with multiple deltas

The OEM wants to use delta updates to save over-the-air bytes. The delta images contain only the code and/or data that has changed from the previous image version. To do so, the OEM will first modify the Director repository, using the vehicle version manifest and dependency resolution to determine the differences between the previous and latest images. The OEM will then add the following to the custom Targets metadata used by the Director repository: (1) the algorithm used to apply a delta image, and (2) the Targets metadata about the delta image. The OEM will also check whether the delta images match the Targets metadata from the Director repository.

3.3. Exceptions

There are a number of factors that could impede the completion of the above scenarios:

- ECUs can be lacking the necessary resources to function as designated. These insufficient resources could include limited CPU or RAM inadequate for performance of public key cryptography; a lack of sufficient storage to undo installation of bad software; or a location on a low-speed network (e.g., LIN).
- ECUs can reside on different network segments, and cannot directly reach each other, requiring a gateway to facilitate communication.
- A user can replace OEM-installed ECUs with aftermarket ECUs.
- A vehicle can download only a limited amount of data via a cellular channel (e.g. due to limits on a data plan).
- A system can lack sufficient power to download or install software updates.

- Vehicles can be offline for extended periods of time, thus missing required updates (e.g., key rotations).
- OEMs can be unwilling to implement costly security or hardware requirements.

3.4. Out of scope

The following topics will not be addressed in this document, as they represent threats outside the scope of Uptane:

- Physical attacks, such as manual tampering with ECUs outside the vehicle.
- Compromise of the packaged software, such as malware embedded in a trusted package.
- Compromise of the supply chain (e.g., build system, version control system, packaging process). The focus of Uptane is end device security and secure delivery. It addresses one part of the solution, but it is designed to pair well with more holistic solutions, like in-toto [IN-TOTO], git signing, TPMs, etc. Recently, the Uptane community approved Scudo as an Uptane augmentation that could be adopted as a formal recommendation in the Uptane Deployment Best Practices in the future.
- Problems associated with OBD or UDS programming of ECUs, such as authentication of communications between ECUs.
- Malicious mirrors of package repositories, which could substitute original packages with malicious packages with matching version numbers [MERCURY].

3.5. Design requirements

The design requirements for this document are governed by the following principal parameters:

- to clearly mandate the design and implementation steps that are security critical and will be followed as is, while offering flexibility in the implementation of non-critical steps. In this manner, users can adapt to support different use models and deployment scenarios.
- to ensure that, if Uptane is implemented, the security practices mandated or suggested in this document do not interfere with the functionality of ECUs, vehicles, or the systems that maintain them.
- to delineate guidelines to ensure that, when any part of the SOTA mechanism of a vehicle is attacked, an attacker has to compromise two or more modules to breach the SOTA mechanism.

4. Threat model and attack strategies

The overarching goal of Uptane is to provide a system that is resilient in the face of various types of compromise. In this section, we describe the goals an attacker could have (Section 4.1) and the capabilities they could have or could develop (Section 4.2). We then describe and classify types of attacks on the system according to the attacker's goals (Section 4.3).

4.1. Attacker goals

We assume that attackers could want to achieve one or more of the following goals, in increasing order of severity:

- Read the contents of updates to discover confidential information, reverse-engineer firmware, or compare two firmware images to identify security fixes and hence determine the fixed security vulnerability.
- Deny installation of updates to prevent vehicles from fixing software problems.
- Cause one or more ECUs in the vehicle to fail, denying use of the vehicle or of certain functions.
- Control ECUs within the vehicle, and possibly the vehicle itself.

4.2. Attacker capabilities

Uptane is designed with resilience to compromise in mind. We assume that attackers could develop one or more of the following capabilities:

- Intercept and modify network traffic (i.e., perform man-in-the-middle attacks). This capability could be developed in two domains:
  - Outside the vehicle, intercepting and modifying traffic between the vehicle and software repositories.
  - Inside the vehicle, intercepting and modifying traffic on one or more vehicle buses (e.g., via an OBD port or by using a compromised ECU as a vector).
- Compromise and control either a Director repository or Image repository server, and any keys stored on that repository, but not both the Director and Image repositories.
- Compromise either a Primary ECU or a Secondary ECU, but not both in the same vehicle.

4.3. Description of threats

Uptane's threat model includes the following types of attacks, organized according to the attacker goals listed in Section 4.1.

4.3.1. Read updates

- *Eavesdrop attack:* Read sensitive or confidential information from an update intended to be encrypted for a specific ECU. (Note: Not all implementations will have a need to protect information in this way.)

4.3.2. Deny installation of updates

An attacker seeking to deny the installation of updates could attempt one or more of the following strategies:

- *Drop-request attack:* Block network traffic outside or inside the vehicle.
- *Slow retrieval attack:* Slow down network traffic, in the extreme case sending barely enough packets to avoid a timeout. Similar to a drop-request attack, except that both the sender and receiver of the traffic still think network traffic is unimpeded.
- *Freeze attack:* Continue to send a properly signed, but old, update bundle to the ECUs, even if newer updates exist.
- *Partial bundle installation attack:* Install a valid (signed) update bundle, and then block selected updates within the bundle.
- *Denial of service attack* against the Uptane repositories or infrastructure.

4.3.3. Interfere with ECU functionality

Attackers seeking to interfere with the functionality of vehicle ECUs in order to cause an operational failure or unexpected behavior could do so in one of the following ways:

- *Rollback attack:* Cause an ECU to install a previously valid software revision that is older than the currently installed version.
- *Endless data attack:* Send a large amount of data to an ECU until it runs out of storage, possibly causing the ECU to fail to operate.
- *Mix-and-match attack:* Install a malicious software bundle in which some of the updates do not interoperate properly. This could be accomplished even if all of the individual images being installed are valid, as long as valid versions exist that are mutually incompatible.

4.3.4. Control an ECU or vehicle

Full control of a vehicle, or one or more ECUs within a vehicle, is the most severe threat.

- *Arbitrary software attack:* Cause an ECU to install and run arbitrary code of the attacker's choice.

5. Detailed design of Uptane

Uptane does not specify implementation details. Instead, this Standard describes the components necessary for a conformant implementation and leaves it up to individual implementers to make their own technology choices within those requirements.

At a high level, Uptane requires:

- Two software repositories:
  - An Image repository containing binary images to install and signed metadata about those images.
  - A Director repository connected to an inventory database that can sign metadata on demand for images in the Image repository.
- Repository tools for generating Uptane-specific metadata about images.
- A public key infrastructure supporting the required metadata production and signing roles on each repository:
  - Root - The certificate authority for the Uptane ecosystem. Distributes public keys for verifying all the other roles' metadata.
  - Timestamp - Indicates whether there are new metadata or images.
  - Snapshot - Indicates images released by the repository at a point in time via signing metadata about Targets metadata.
  - Targets - Indicates metadata about images, such as hashes and file sizes.
- A secure way for ECUs to know the time.
- An ECU capable of downloading images and associated metadata from the Uptane servers.
- An in-vehicle client on a Primary ECU capable of verifying the signatures on all update metadata and downloading updates on behalf of its associated Secondary ECUs. The Primary ECU can be the same ECU that communicates with the server.
- A client or library on each Secondary ECU capable of performing either full or partial verification of metadata.

5.1. Roles on repositories

A repository contains images and metadata. Each role has a particular type of metadata associated with it, as described in Section 5.2.

5.1.1. The Root role

A repository's Root role SHALL produce and sign Root metadata as described in Section 5.2.2.

5.1.2. The Targets role

A repository's Targets role SHALL produce and sign metadata about images and delegations as described in Section 5.2.3.

5.1.2.1. Delegations

The Targets role on the Image repository can delegate the responsibility of signing metadata to other, custom-defined roles referred to as delegated targets. If it does, it SHALL do so as specified in Section 5.2.3.2.

As responsibility for signing images or a subset of images could be delegated to more than one role, it is possible that two different roles will be trusted to sign a particular image. For this reason, delegations SHALL be prioritized.

A particular delegation for a subset of images could be designated as **terminating**. For terminating delegations, the client SHALL NOT search any further if it does not find validly signed metadata about those images. Delegations SHOULD NOT be terminating by default; terminating delegations SHOULD only be used when there is a compelling technical reason to do so.

A delegation for a subset of images could be a multi-role delegation [TAP-3]. A multi-role delegation indicates that multiple roles are needed to sign a particular image and each of the delegatee roles SHALL sign the same metadata.

Delegations only apply to the Image repository. The Targets role on the Director repository SHALL NOT delegate metadata signing responsibility.

5.1.3. The Snapshot role

A repository's Snapshot role SHALL produce and sign metadata about all Targets metadata the repository releases, including the current version number of the top-level Targets metadata, and the version numbers of all delegated Targets metadata, as described in Section 5.2.4.

5.1.4. The Timestamp role

A repository's Timestamp role SHALL produce and sign metadata indicating whether there are new metadata or images on the repository. It SHALL do so by signing the metadata about the Snapshot metadata file.

5.2. Metadata structures

Uptane's security guarantees all rely on properly created metadata that follows a designated structure. The Uptane Standard **does not** mandate any particular

format or encoding for the metadata as a whole. ASN.1 (with any encoding scheme like BER, DER, XER, etc.), JSON, XML, or any other encoding format that is capable of providing the required structure can be used.

However, string comparison is required as part of metadata verification. To ensure an accurate basis for comparing strings, all strings SHALL be encoded in the Unicode Format for Network Interchange as defined in [RFC5198], including normalization into Unicode Normalization Form C ([NFC]).

The *Deployment Best Practices* ([DEPLOY]), Joint Development Foundation Projects, LLC, Uptane Series provides some examples of metadata structures in ASN.1 and JSON that conform to the Standard.

5.2.1. Common metadata structures

Every public key SHALL be represented using a public key identifier. A public key identifier is EITHER all of the following:

- The value of the public key itself (which could be, for example, formatted as a PEM string)
- The public key cryptographic algorithm used by the key (such as RSA or ECDSA)
- The particular scheme used to verify the signature (such as `rsassa-pss-sha256` or `ecdsa-sha2-nistp256`)

OR a secure hash over at least the above components (such as the keyid mechanism in TUF).

All four Uptane roles (Root, Targets, Snapshot, and Timestamp) share a common structure. They SHALL contain the following two attributes:

- A payload of metadata to be signed
- An attribute containing the signature(s) of the payload, where each entry specifies:
  - The public key identifier of the key being used to sign the payload
  - A signature with this key over the payload

The payload differs depending on the role. However, the payload for all roles shares a common structure. It SHALL contain the following four attributes:

- An indicator of the type of role (Root, Targets, Snapshot, or Timestamp)
- An expiration date and time
- An integer version number, which SHOULD be incremented each time the metadata file is updated
- The role-specific metadata for the role indicated

The following sections describe the role-specific metadata. All roles SHALL follow the common structures described here.

5.2.2. Root metadata

A repository's Root metadata distributes the public keys of the top-level Root, Targets, Snapshot, and Timestamp roles, as well as revocations of those keys. It SHALL contain two attributes:

- A representation of the public keys for all four roles. Each key SHALL have a unique public key identifier.
- An attribute mapping each role to (1) its public key(s), and (2) the threshold of signatures required for that role.

5.2.3. Targets metadata

The Targets metadata on a repository contains all of the information about images to be installed on ECUs. This includes filenames, hashes, and file sizes. It can also include other useful information, such as what types of hardware are compatible with a particular image.

Targets metadata can also contain metadata about delegations, allowing one Targets role to delegate its authority to another. This means that an individual Targets metadata file might contain only metadata about delegations, only metadata about images, or some combination of the two. The details of how ECUs traverse the delegation tree to find valid metadata about images is specified in Section 5.4.4.7.

5.2.3.1. Metadata about images

To be available to install on clients, all images on the repository SHALL have their metadata listed in a Targets role. Each Targets role can provide a list of some images on the repository. This list SHALL provide, at a minimum, the following information about each image:

- The image filename
- The size of the image in bytes
- One or more hashes of the image file, along with the hashing function used

If there are no images included in the Targets metadata from the Director repository, then the metadata SHALL include a vehicle identifier in order to avoid a replay attack.

5.2.3.1.1. Custom metadata about images

In addition to what is required, Targets metadata files can contain extra metadata for images on the repository. This metadata can be customized for a particular use case. Examples of use cases for different types of custom metadata can be found in the *Deployment Best Practices* document ([DEPLOY]). However, there

are a few important pieces of custom metadata that SHOULD be present in most implementations. In addition, there is one element in the custom metadata that SHALL be present in the Targets metadata from the Director.

Custom metadata can also contain a demarcated field or section that SHALL match whenever two pieces of metadata are checked against each other, such as when Targets metadata from the Director repository is checked against Targets metadata from the Image repository.

The information listed below SHOULD be provided for each image on both the Image repository and the Director repository. If a "SHALL match section" is to be implemented, that is where this information SHOULD be placed.

- A release counter, to be incremented each time a new version of the image is released. This can be used to prevent rollback attacks even in cases where the Director repository is compromised.
- A hardware identifier, or list of hardware identifiers, representing models of ECUs with which the image is compatible. This can be used to ensure that an ECU cannot be ordered to install an incompatible image, even in cases where the Director repository is compromised.

The following information is CONDITIONALLY REQUIRED for each image on the Director repository IF that image is encrypted:

- Information about filenames, hashes, and file size of the encrypted image.
- Information about the encryption method, and other relevant information–for example, a symmetric encryption key encrypted by the ECU's asymmetric key could be included in the Director repository metadata.

The following information SHALL be provided from the Director repository for each image in the Targets metadata:

- An ECU identifier (such as a serial number), specifying the ECU that SHOULD install the image.

The Director repository could provide a download URL for the image file. This may be useful, for example, when the image is on a public CDN and the Director wishes to provide a signed URL.

5.2.3.2. Metadata about delegations

A Targets metadata file on the Image repository (but not the Director repository) SHALL be able to delegate signing authority to other entities. For example, it could delegate signing authority for a particular ECU's firmware to that ECU's supplier. A metadata file can contain any number of delegations and SHALL keep the delegations in prioritized order.

Any metadata file with delegations SHALL provide the following information:

- A list of public keys of all delegatees. Each key SHOULD have a unique public key identifier and a key type.
- A list of delegations, each of which contains:
  - A list of the filenames to which this role applies. This could be expressed using wildcards, or by enumerating a list, or a combination of the two.
  - An optional list of the hardware identifiers to which this role applies. If this is omitted, any hardware identifier will match.
  - An indicator of whether or not this is a terminating delegation. (See Section 5.1.2.1.)
  - A list of the roles to which this delegation applies. Each role needs to specify:
    * A name for the role (e.g., "supplier1-qa")
    * The key identifiers for each key this role uses
    * A threshold of keys that SHALL sign for this role

Note that **any** Targets metadata file stored on the Image repository can contain delegations, and these delegations can be in chains of arbitrary length.

5.2.4. Snapshot metadata

The Snapshot metadata lists version numbers and filenames of all Targets metadata files. It protects against mix-and-match attacks if a delegated supplier key is compromised.

For each Targets metadata file on the repository, the Snapshot metadata SHALL contain the following information:

- The filename and version number of the Targets metadata file.

The Snapshot metadata could also list the Root metadata filename and version number for the purpose of backwards compatibility. Historically, this was a requirement in TUF, but it is no longer required and does not provide a significant security benefit.

5.2.5. Timestamp metadata

The Timestamp metadata SHALL contain the following information:

- The filename and version number of the latest Snapshot metadata on the repository.
- One or more hashes of the Snapshot metadata file, along with the hashing function used.

5.2.6. Repository mapping metadata

As described in the introduction to Section 5, Uptane requires a Director repository and an Image repository. However, it is possible to have an Uptane-conformant implementation that has more than two repositories.

Repository mapping metadata informs a Primary ECU about which repositories to trust for images or image paths. [TAP-4] describes how to make use of more complex repository mapping metadata in order to have more than the two required repositories.

Repository mapping metadata, or the equivalent informational content, SHALL be present on all Primary ECUs, and SHALL contain the following information:

- A list of repository names and one or more URLs at which the named repository can be accessed. At a minimum, this SHALL include the Director and Image repositories.
- A list of mappings of image paths to repositories, each of which contains:
  - A list of image paths. Image paths could be expressed using wildcards, or by enumerating a list, or a combination of the two.
  - A list of repositories that SHALL provide signed Targets metadata for images stored at those paths.

For example, in the most basic Uptane case, the repository mapping metadata would contain:

- The name and URL of the Director repository.
- The name and URL of the Image repository.
- A single mapping indicating that all images (*) SHALL be signed by both the Director and Image repository.

Note that the metadata need not be in the form of a metadata file. For example, in the basic case where there is only one Director and one Image repository, and all images need to have signed metadata from both repositories, it would be sufficient to have a configuration file with URLs for the two repositories and a client that always checks for metadata matches between them. In this case, no explicit mapping would be defined, because the mapping is defined as part of the Uptane client implementation.

The *Uptane Deployment Best Practices* document ([DEPLOY]) provides more guidance on how to implement repository mapping metadata for more complex use cases. It also discusses strategies for updating repository mapping metadata, if required.

5.2.7. Rules for filenames in repositories and metadata

There is a difference between the filename in a metadata file or an ECU, and the filename on a repository. This difference exists in order to avoid race conditions, where metadata and images are read from, and written to, at the same time. For more details, the reader can read the TUF specification [TUF-spec] and PEP 458 [PEP-458].

Unless stated otherwise, all files SHALL be written to repositories in accordance with the following two rules:

1. Metadata filenames SHALL be qualified with version numbers. If a metadata file A is specified as FILENAME.EXT in another metadata file B, then it SHALL be written as VERSION.FILENAME.EXT, where VERSION is A's version number, as defined in Section 5.2.1, with one exception: If the version number of the Timestamp metadata file is not be known in advance by a client, it could be read from, and written to, a repository using a filename without a version number qualification, i.e., FILENAME.EXT.
2. If an image is specified in a Targets metadata file as FILENAME.EXT, it SHALL be written to the repository as HASH.FILENAME.EXT, where HASH is one of the hash digests of the file, as specified in Section 5.2.3.1. The file SHALL be written to the repository using n different filenames, one for each hash digest listed in its corresponding Targets metadata.
3. Filenames of images SHOULD be encoded to prevent a path traversal on the client system, either by using URL encoding or by limiting the allowed character set in the filename.

For example:

- The version number of the Snapshot metadata file is 61, and its filename in the Timestamp metadata is *snapshot.json*. The filename on the repository will be *61.snapshot.json*.
- There is an image with the *filename acme_firmware.bin* specified in the Targets metadata, with a SHA3-256 of *aaaa* and a SHA-512/224 of *bbbb*. It will have two filenames on the repository: *aaaa.acme_firmware.bin* and *bbbb.acme_firmware.bin*.

5.3. Server / repository implementation requirements

An Uptane implementation SHALL make the following services available to vehicles:

- Image repository
- Director repository

Additionally, an Uptane implementation requires ECUs to have a secure way to know the current time.

5.3.1. Image repository

The Image repository exists to allow an OEM and/or its suppliers to upload images and their associated metadata. It makes these images and their metadata available to vehicles. The Image repository is designed to be primarily controlled by human actors, and updated relatively infrequently.

The Image repository SHALL expose an interface permitting the download of metadata and images. This interface SHOULD be public.

The Image repository SHALL require authorization for writing metadata and images.

The Image repository SHALL provide a method for authorized users to upload images and their associated metadata. It SHALL check that a user writing metadata and images is authorized to do so for that specific image by checking the chain of delegations as described in Section 5.2.3.2.

The Image repository SHALL implement storage that permits authorized users to write an image file using a unique filename, and later read the same file using the same name. It could use any filesystem, key-value store, or database that fulfills this requirement.

The Image repository could require authentication for read access.

5.3.2. Director repository

The Director repository instructs ECUs as to which images will be installed by producing signed metadata on demand. Unlike the Image repository, it is mostly controlled by automated, online processes. It also consults a private inventory database containing information on vehicles, ECUs, and software revisions.

The Director repository SHALL expose an interface for Primaries to upload vehicle version manifests (Section 5.4.2.1.1) and download metadata. This interface SHOULD be public.

The Director could encrypt images for ECUs that require them, either by encrypting on-the-fly or by storing encrypted images on the repository.

The Director repository SHALL implement storage that permits an automated service to write generated metadata files. It could use any filesystem, key-value store, or database that fulfills this requirement.

5.3.2.1. Directing installation of images on vehicles

A Director repository SHALL conform to the following six-step process for directing the installation of software images on a vehicle.

1. The Director SHOULD first identify the vehicle. This could be done when the Director receives a vehicle version manifest sent by a Primary (as described in Section 5.4.2.1), decodes the manifest, and determines the unique vehicle identifier. Additionally, the Director could utilize other mechanisms to uniquely identify a vehicle (e.g., 2-way TLS with unique client certificates).
2. Using the vehicle identifier, the Director queries its inventory database (as described in Section 5.3.2.2) for relevant information about each ECU in the vehicle.
3. The Director SHALL check the manifest for accuracy compared to the information in the inventory database. If any of the required checks fail, the Director SHOULD drop the request. An implementer can make additional checks if desired. At a minimum, the Director SHOULD check the following:
   - Each ECU recorded in the inventory database is also represented in the manifest.
   - The signature of the manifest matches the ECU key of the Primary that sent it.
   - The signature of each Secondary's contribution to the manifest matches the ECU key of that Secondary.
   - The time sent in the ECU version report.
4. The Director SHOULD check that the nonce or counter in each ECU version report has not been used before to prevent a replay of the ECU version report. If the nonce or counter is reused the Director SHOULD drop the request.
5. The Director extracts information about currently installed images from the vehicle version manifest. Using this information, it determines if the vehicle is already up-to-date, and if not, determines a set of images that could be installed. The exact process by which this determination takes place is out of scope for this Standard. However, the Director SHALL take into account *dependencies* and *conflicts* between images and SHOULD consult well-established techniques for dependency resolution.
6. The Director could encrypt images for ECUs that require it.
7. The Director generates new metadata representing the desired set of images to be installed on the vehicle, based on the dependency resolution in step 4. This includes Targets (Section 5.2.3), Snapshot (Section 5.2.4), and Timestamp (Section 5.2.5) metadata. It then sends this metadata to the Primary as described in Section 5.4.2.3.

5.3.2.2. Inventory Database

The Director SHALL use a private inventory database to store information about ECUs and vehicles. An implementer could use any durable database for this purpose.

The inventory database SHALL record the following pieces of information:

- Per vehicle:
  - A unique identifier (such as a VIN)
- Per ECU:
  - A unique identifier (such as a serial number)
  - The vehicle identifier the ECU is associated with
  - An ECU key (symmetric or asymmetric; for asymmetric keys, only the public part SHOULD be stored)
  - The ECU key identifier (as defined in Section 5.2.1)
  - Whether the ECU is a Primary or a Secondary

The inventory database can record other information about ECUs and vehicles. It SHOULD record a hardware identifier for each ECU to protect against the possibility of directing the ECU to install incompatible firmware.

5.4. In-vehicle implementation requirements

An Uptane-conformant ECU SHALL be able to download and verify image metadata and image binaries before installing a new image and SHALL have a secure way of verifying the current time, or a sufficiently recent attestation of the time.

All ECUs SHALL monitor the download speed of image metadata and image binaries to detect and respond to a slow retrieval attack. If the download is slower than a pre-defined threshold, the ECU SHOULD send an alert to the Director repository, for example as part of the next vehicle version manifest.

Each ECU receiving over-the-air updates in a vehicle is either a Primary or a Secondary ECU. A Primary ECU collects and delivers to the Director vehicle manifests (Section 5.4.2.1.1) that contain information about which images have been installed on ECUs in the vehicle. It also verifies the time, and downloads and verifies the latest metadata and images for itself and for its Secondaries. A Secondary ECU verifies the time, and downloads and verifies the latest metadata and images for itself from its associated Primary ECU. It also sends signed information about its installed images to its associated Primary.

All ECUs SHALL verify image metadata as specified in Section 5.4.4 before installing an image or making it available to other ECUs. A Primary ECU SHALL perform full verification (Section 5.4.4.2). A Secondary ECU SHOULD perform full verification if possible. If a Secondary cannot perform full verification, it SHALL, at the very least, perform partial verification. In addition, it can also

perform some steps from the full verification process. See the *Uptane Deployment Best Practices* document ([DEPLOY]) for a discussion of how to choose between partial and full verification.

ECUs SHALL have a secure source of time. An OEM/Uptane implementer can use any external source of time that is demonstrably secure.

5.4.1. Build-time prerequisite requirements for ECUs

For an ECU to be capable of receiving Uptane-secured updates, it SHALL have the following data provisioned at the time it is manufactured or installed in the vehicle:

1. A sufficiently recent copy of required Uptane metadata at the time of manufacture or install. This is necessary for the ECU to authenticate that the remote repository is legitimate when it first downloads metadata in the field. See *Uptane Deployment Best Practices* ([DEPLOY]) for more information.
    - Partial verification Secondary ECUs SHALL have the Root and Targets metadata from the Director repository (to reduce the scope of rollback and replay attacks). These ECUs can also have metadata from other roles or the Image repository if they will be used by the Secondary.
    - Full verification ECUs SHALL have a complete set of metadata (Root, Targets, Snapshot, and Timestamp) from both repositories (to prevent rollback and replay attacks), as well as the repository mapping metadata (Section 5.2.6). Delegations are not required.
2. The current time, or a secure attestation of a sufficiently recent time.
3. **ECU identity keys**. These keys, which are unique to each ECU, are used to sign ECU version reports and decrypt images. ECU identity keya can be either symmetric asymmetric key. If asymmetric keys are used, there SHOULD be separate keys for encryption and signing. For the purposes of this Standard, the set of keys that an ECU uses is referred to as the ECU key (singular), even if it is actually multiple keys used for different purposes. Note that while identity keys are required to be unique to the ECU to avoid replay attacks, the secret keys used to decrypt images need not be unique.

5.4.2. What the Primary does

A Primary downloads, verifies, and distributes the latest time, metadata, and images. To do so, it SHALL perform the following seven steps:

1. Construct and send vehicle version manifest (Section 5.4.2.1)

2. Download and check current time (Section 5.4.2.2)
3. Download and verify metadata (Section 5.4.2.3)
4. Download and verify images (Section 5.4.2.4)
5. Send latest time to Secondaries (Section 5.4.2.5)
6. Send metadata to Secondaries (Section 5.4.2.6)
7. Send images to Secondaries (Section 5.4.2.7)

Note that the subsequent sections concerning requirements for a Primary do not prohibit implementing Primary capabilities on an ECU that does not communicate directly with the Uptane repositories. This allows for implementations to have multiple ECUs within the vehicle performing functions equivalent to a Primary. If multiple such Primaries are included within a vehicle, each Primary SHOULD have a designated set of Secondaries and each Secondary SHALL have at least one Primary responsible for providing its updates

5.4.2.1. Construct and send vehicle version manifest

The Primary SHALL build a *vehicle version manifest* as described in Section 5.4.2.1.1.

Once the complete manifest is built, the Primary can send the manifest to the Director repository. However, it is not strictly required that the Primary send the manifest until step three. If permitted by the implementation, a Primary could send only a diff of the manifest to save bandwidth. If an implementation permits diffs, the Director SHOULD have a way to request a full manifest.

Secondaries can send their version reports at any time so that they are already stored on the Primary when it wishes to check for updates. Alternatively, the Primary can request a version report from each Secondary at the time of the update check.

5.4.2.1.1. Vehicle version manifest

The vehicle version manifest is a metadata structure that SHALL contain the following information:

- An attribute containing the signature(s) of the payload, each specified by:
  - The public key identifier of the key being used to sign the payload
  - The signing method (i.e., ed25519, rsassa-pss, etc.)
  - A hash of the payload to be signed
  - The hashing function used (i.e., SHA3-256, SHA-512/224, etc.)
  - The signature of the hash
- A payload representing the installed versions of each software image on the vehicle. This payload SHALL contain:
  - The vehicle's unique identifier (e.g., the VIN)

  - The Primary ECU's unique identifier (e.g., the serial number)
  - A list of ECU version reports as specified in Section 5.4.2.1.2

Note that one of the ECU version reports SHOULD be the version report for the Primary itself.

5.4.2.1.2. ECU version report

An ECU version report is a metadata structure that SHALL contain the following information:

- An attribute containing the signature(s) of the payload, each specified by:
  - The public key identifier of the key being used to sign the payload
  - The signing method (i.e., ed25519, rsassa-pss, etc.)
  - A hash of the payload to be signed
  - The hashing function used (i.e., SHA3-256, SHA-512/224, etc.)
  - The signature of the hash
- A payload containing:
  - The ECU's unique identifier (e.g., the serial number)
  - The filename, length, and hashes of its currently installed image (i.e., the non-custom Targets metadata for this particular image)
  - An indicator of any detected security attack
  - The latest time the ECU can verify at the time this version report was generated
  - A nonce or counter to prevent a replay of the ECU version report. This value SHALL change each update cycle.

5.4.2.2. Download and check current time

The Primary SHALL load the current time from a secure source.

5.4.2.3. Download and verify metadata

The Primary SHALL download metadata for all targets and perform a full verification on it as specified in Section 5.4.4.2.

5.4.2.4. Download and verify images

The Primary SHALL download and verify images for itself and for all of its associated Secondaries. Images SHALL be verified by checking that the hash of the image file matches the hash specified in the Director's Targets metadata for that image.

There could be several different filenames that all refer to the same image binary, as described in Section 5.2.7. If the Primary has received multiple hashes for

a given image binary via the Targets role (see Section 5.2.3.1) then it SHALL verify every hash for this image even though the image is identified by a single hash as part of its filename.

5.4.2.5. Send latest time to Secondaries

The Primary SHOULD send the time to each ECU.

5.4.2.6. Send metadata to Secondaries

The Primary SHALL make available to each of its associated Secondaries all new metadata required for verification on that Secondary. For full verification Secondaries, this includes the metadata for all four roles from both repositories, plus any delegated Targets metadata files the Secondary will recurse through to find the proper delegation. For partial verification Secondaries, this could include fewer metadata files; at a minimum, it includes only the Targets metadata file from the Director repository.

The Primary SHOULD determine the minimal set of metadata files to send to each Secondary by performing delegation resolution as described in Section 5.4.4.2.

Each Secondary SHALL store the latest copy of all metadata required for its own verification.

5.4.2.7. Send images to Secondaries

The Primary SHALL send the latest image to each of its associated Secondaries that have sufficient storage to receive it.

For Secondaries without sufficient storage to store a copy of the image, the Primary SHOULD wait for a request from the Secondary to stream the new image file to it. The Secondary will send the request once it has verified the metadata sent in the previous step.

5.4.3. Installing images on Primary or Secondary ECUs

An ECU SHALL perform the following steps when attempting to install a new image:

1. Verify latest attested time. This is optional if the ECU does not have the capacity to verify a time message (Section 5.4.3.1)
2. Verify metadata (Section 5.4.3.2)
3. Download latest image (Section 5.4.3.3)

4. Verify image (Section 5.4.3.4)
5. Install image (Section 5.4.3.5)
6. Create and send version report (Section 5.4.3.6)

5.4.3.1. Load and verify the latest attested time

IF the ECU has the capability to verify a time message, the ECU SHOULD load and verify the current time, or the most recent securely attested time.

5.4.3.2. Verify metadata

The ECU SHALL verify the latest downloaded metadata (Section 5.4.4) using either full or partial verification. If the metadata verification fails for any reason, the ECU SHALL jump to the final step (Section 5.4.3.6).

5.4.3.3. Download latest image

If the ECU has limited secondary storage, i.e., insufficient buffer storage to temporarily store the latest image before installing it, it SHALL download the latest image from the Primary. (If the ECU has sufficient secondary storage, it will already have the latest image in its secondary storage as specified in Section 5.4.2.7, and SHALL skip to the next step.) The ECU SHOULD first create a backup of its previous working image and store it elsewhere (e.g., the Primary).

The filename used to identify the latest known image (i.e., the file to request from the Primary) SHALL be determined as follows:

1. Load the Targets metadata file from the Director repository.
2. Find the Targets metadata associated with this ECU identifier.
3. Construct the Image filename using the rule in Section 5.2.7, or use the download URL specified in the Director metadata.
4. If there is no Targets metadata about this image, abort the update cycle and report that there is no such image. Additionally, in the case of failure, the ECU SHALL retain its previous Targets metadata instead of using the new Targets metadata. Otherwise, download the image (up to the number of bytes specified in the Targets metadata) and verify it according to Section 5.4.3.4.

When the Primary responds to the download request, the ECU SHALL overwrite its current image with the downloaded image from the Primary.

If any part of this step fails, the ECU SHALL jump to the final step (Section 5.4.3.6).

5.4.3.4. Verify image

The ECU SHALL verify that the latest image matches the latest metadata as follows:

1. Load the latest Targets metadata file from the Director.
2. Find the Targets metadata associated with this ECU identifier.
3. Check that the hardware identifier in the metadata matches the ECU's hardware identifier.
4. Check that the image filename is valid for this ECU. This could be a comparison against a wildcard path, which restricts the ECUs to which a delegation will apply.
5. Check that the release counter of the image in the previous metadata, if it exists, is less than or equal to the release counter in the latest metadata.
6. If the image is encrypted, decrypt the image with a decryption key to be chosen as follows:
   - If the ECU key is a symmetric key, the ECU SHALL use the ECU key for image decryption.
   - If the ECU key is asymmetric, the ECU SHALL check the Targets metadata for an encrypted symmetric key. If such a key is found, the ECU SHALL decrypt the symmetric key using its ECU key, and use the decrypted symmetric key for image decryption.
   - If the ECU key is asymmetric and there is no symmetric key in the Targets metadata, the ECU SHALL use its ECU key for image decryption.
7. Check that all hashes listed in the metadata match the corresponding hashes of the image.

If the ECU has enough secondary storage capacity to store the image, the checks SHOULD be performed on the image in secondary storage before it is installed.

When checking hashes, the ECU SHALL additionally check that the length of the image matches the length listed in the metadata.

NOTE: See [DEPLOY] for guidance on how to deal with Secondary ECU failures for ECUs that have limited secondary storage.

If any step fails, the ECU SHALL jump to the final step (Section 5.4.3.6).

5.4.3.5. Install image

The ECU SHALL attempt to install the update. This installation SHOULD occur at a time when all pre-conditions are met. These pre-conditions could include ensuring the vehicle is in a safe environment for an installation (e.g., the vehicle is parked when updating a specific ECU). Other pre-conditions could include ensuring the ECU has a backup of its current image and metadata in case the current installation fails.

5.4.3.6. Create and send version report

The ECU SHALL create a version report as described in Section 5.4.2.1.2, and send it to the Primary (or simply save it to disk, if the ECU is a Primary). The Primary SHOULD write the version reports it receives to disk and associate them with the Secondaries that sent them.

5.4.4. Metadata verification procedures

A Primary ECU SHALL perform full verification of metadata. A Secondary ECU SHOULD perform full verification of metadata. If a Secondary cannot perform full verification, it SHALL, at the very least, perform partial verification.

If a step in the following workflows does not succeed (e.g., the update is aborted because a new metadata file was not signed), an ECU SHOULD still be able to update again in the future. Errors raised during the update process SHOULD NOT leave ECUs in an unrecoverable state.

5.4.4.1. Partial verification

In order to perform partial verification, an ECU SHALL perform the following steps:

1. Load and verify the current time or the most recent securely attested time.
2. Download and check the Targets metadata file from the Director repository, following the procedure in Section 5.4.4.6.

Note that this verification procedure is the smallest set of Uptane checks permissible for an Uptane Secondary ECU. An ECU can additionally implement more metadata checks.

For example, an ECU could also fetch and verify Root metadata from the Director (following the procedure in Section 5.4.4.3) before checking Targets metadata. Performing this additional check would provide the ECU with a secure way to receive and validate a rotation of the Director's Targets key.

See [DEPLOY] for more discussion on this topic.

5.4.4.2. Full verification

Full verification of metadata means that the ECU checks that the Targets metadata about images from the Director repository matches the Targets metadata about the same images from the Image repository. This provides resilience to a key compromise in the system.

Full verification SHALL be performed by Primary ECUs and SHOULD be performed by Secondary ECUs. In the following instructions, whenever an ECU

is directed to download metadata, it applies only to Primary ECUs.

Before starting full verification, the repository mapping metadata SHALL be consulted to determine where to download metadata from. This procedure assumes the basic Uptane case: there are only two repositories (Director and Image), and all image paths are required to be signed by both repositories. If a more complex repository layout is being used, refer to [DEPLOY] for guidance on how to determine where metadata could be downloaded from.

In order to perform full verification, an ECU SHALL perform the following steps:

1. Load and verify the current time or the most recent securely attested time.
2. Download and check the Root metadata file from the Director repository, following the procedure in Section 5.4.4.3.
3. Download and check the Timestamp metadata file from the Director repository, following the procedure in Section 5.4.4.4.
4. Check the previously downloaded Snapshot metadata file from the Directory repository (if available). If the hashes and version number of that file match the hashes and version number listed in the new Timestamp metadata, there are no new updates and the verification process SHALL be stopped and considered complete. Otherwise, download and check the Snapshot metadata file from the Director repository, following the procedure in Section 5.4.4.5.
5. Download and check the Targets metadata file from the Director repository, following the procedure in Section 5.4.4.6.
6. If the Targets metadata from the Directory repository indicates that there are no new targets that are not already currently installed, the verification process SHOULD be stopped and considered complete. Optionally, implementors can order vehicles to check image repo root metadata when desirable, even in the absence of an update. Otherwise, download and check the Root metadata file from the Image repository, following the procedure in Section 5.4.4.3.
7. Download and check the Timestamp metadata file from the Image repository, following the procedure in Section 5.4.4.4.
8. Check the previously downloaded Snapshot metadata file from the Image repository (if available). If the hashes and version number of that file match the hashes and version number listed in the new Timestamp metadata, the ECU SHALL skip to the last step. Otherwise, download and check the Snapshot metadata file from the Image repository, following the procedure in Section 5.4.4.5.
9. Download and check the top-level Targets metadata file from the Image repository, following the procedure in Section 5.4.4.6.
10. Verify that Targets metadata from the Director and Image repositories match. A Primary ECU SHALL perform this check on metadata for all images listed in the Targets metadata file from the Director repository downloaded in step 6. A Secondary ECU can elect to perform this check only on the metadata for the image it will install. (That is, the image

metadata from the Director that contains the ECU identifier of the current ECU.) To check that the metadata for an image matches, complete the following procedure:

(a) Locate and download a Targets metadata file from the Image repository that contains an image with exactly the same filename listed in the Director metadata, following the procedure in Section 5.4.4.7.

(b) Check that the Targets metadata from the Image repository matches the Targets metadata from the Director repository:

i. Check that the non-custom metadata (i.e., length and hashes) of the unencrypted or encrypted image are the same in both sets of metadata. Note: the Primary is responsible for validating encrypted images and associated metadata. The target ECU (Primary or Secondary) is responsible for validating the unencrypted image and associated metadata.

ii. Check that all SHALL match custom metadata (e.g., hardware identifier and release counter) are the same in both sets of metadata.

iii. Check that the release counter, if one is used, in the previous Targets metadata file is less than or equal to the release counter in this Targets metadata file.

If any step fails, the ECU SHALL return an error code indicating the failure. If a check for a specific type of security attack fails (i.e., rollback, freeze, arbitrary software, etc.), the ECU SHOULD return an error code that indicates the type of attack.

5.4.4.3. How to check Root metadata

To properly check Root metadata, an ECU SHOULD:

1. Load the previous Root metadata file.
2. Update to the latest Root metadata file.
   (a) Let N denote the version number of the latest Root metadata file (which at first could be the same as the previous Root metadata file).
   (b) Try downloading a new version N+1 of the Root metadata file, up to some X number of bytes. The value for X is set by the implementer. For example, X could be tens of kilobytes. The filename used to download the Root metadata file is of the fixed form VERSION_NUMBER.FILENAME.EXT (e.g., 42.root.json). If this file is not available, the current Root metadata file is the latest; continue with step 3.
   (c) Version N+1 of the Root metadata file SHALL have been signed by the following: (1) a threshold of unique keys specified in the latest Root metadata file (version N), and (2) a threshold of unique keys specified in the new Root metadata file being validated (version N+1). If version N+1 is not signed as required, discard it, abort the update

cycle, and report the signature failure. On the next update cycle, begin at version N of the Root metadata file. (Checks for an arbitrary software attack.)

(d) The version number of the latest Root metadata file (version N) SHALL be less than or equal to the version number of the new Root metadata file (version N+1). Effectively, this means checking that the version number signed in the new Root metadata file is indeed N+1. If the version of the new Root metadata file is less than the latest metadata file, discard it, abort the update cycle, and report the rollback attack. On the next update cycle, begin at step 1 and version N of the Root metadata file. (Checks for a rollback attack.)

(e) Set the latest Root metadata file to the new Root metadata file.

(f) Repeat steps 2.1 to 2.6.

3. Check that the current (or latest securely attested) time is lower than the expiration timestamp in the latest Root metadata file. (Checks for a freeze attack.)
4. If the Timestamp and/or Snapshot keys have been rotated, delete the previous Timestamp and Snapshot metadata files. (Checks for recovery from fast-forward attacks [MERCURY].)

5.4.4.4. How to check Timestamp metadata

To properly check Timestamp metadata, an ECU SHOULD:

1. Download up to Y number of bytes. The value for Y is set by the implementer. For example, Y could be tens of kilobytes. The filename used to download the Timestamp metadata file is of the fixed form FILENAME.EXT (e.g., timestamp.json).
2. Check that it has been signed by the threshold of unique keys specified in the latest Root metadata file. If the new Timestamp metadata file is not properly signed, discard it, abort the update cycle, and report the signature failure. (Checks for an arbitrary software attack.)
3. Check that the version number of the previous Timestamp metadata file, if any, is less than or equal to the version number of this Timestamp metadata file. If the new Timestamp metadata file is older than the trusted Timestamp metadata file, discard it, abort the update cycle, and report the potential rollback attack. (Checks for a rollback attack.)
4. Check that the current (or latest securely attested) time is lower than the expiration timestamp in this Timestamp metadata file. If the new Timestamp metadata file has expired, discard it, abort the update cycle, and report the potential freeze attack. (Checks for a freeze attack.)

5.4.4.5. How to check Snapshot metadata

To properly check Snapshot metadata, an ECU SHOULD:

1. Download up to the number of bytes specified in the Timestamp metadata file, constructing the metadata filename as defined in Section 5.2.7.
2. The hashes and version number of the new Snapshot metadata file SHALL match the hashes and version number listed in the Timestamp metadata. If the hashes and version number do not match, discard the new Snapshot metadata, abort the update cycle, and report the failure. (Checks for a mix-and-match attack.)
3. Check that it has been signed by the threshold of unique keys specified in the latest Root metadata file. If the new Snapshot metadata file is not signed as required, discard it, abort the update cycle, and report the signature failure. (Checks for an arbitrary software attack.)
4. Check that the version number of the previous Snapshot metadata file, if any, is less than or equal to the version number of this Snapshot metadata file. If this Snapshot metadata file is older than the previous Snapshot metadata file, discard it, abort the update cycle, and report the potential rollback attack. (Checks for a rollback attack.)
5. Check that the version number listed by the previous Snapshot metadata file for each Targets metadata file is less than or equal to its version number in this Snapshot metadata file. If this condition is not met, discard the new Snapshot metadata file, abort the update cycle, and report the failure. (Checks for a rollback attack.)
6. Check that each Targets metadata filename listed in the previous Snapshot metadata file is also listed in this Snapshot metadata file. If this condition is not met, discard the new Snapshot metadata file, abort the update cycle, and report the failure. (Checks for a rollback attack.)
7. Check that the current (or latest securely attested) time is lower than the expiration timestamp in this Snapshot metadata file. If the new Snapshot metadata file is expired, discard it, abort the update cycle, and report the potential freeze attack. (Checks for a freeze attack.)

5.4.4.6. How to check Targets metadata

To properly check Targets metadata, an ECU SHOULD:

1. Download up to Z number of bytes, constructing the metadata filename as defined in Section 5.2.7. The value for Z is set by the implementer. For example, Z could be tens of kilobytes.
2. The version number of the new Targets metadata file SHALL match the version number listed in the latest Snapshot metadata. If the version number does not match, discard it, abort the update cycle, and report the failure. (Checks for a mix-and-match attack.) This step can be skipped when checking Targets metadata on a partial verification ECU, as these ECUs might not have Snapshot metadata.
3. Check that the Targets metadata has been signed by the threshold of unique keys specified in the relevant metadata file. (Checks for an arbitrary

software attack.):
   (a) If checking top-level Targets metadata, the threshold of keys is specified in the Root metadata.
   (b) If checking delegated Targets metadata, the threshold of keys is specified in the Targets metadata file that delegated authority to this role.
4. Check that the version number of the previous Targets metadata file, if any, is less than or equal to the version number of this Targets metadata file. (Checks for a rollback attack.)
5. Check that the current (or latest securely attested) time is lower than the expiration timestamp in this Targets metadata file. (Checks for a freeze attack.)
6. If checking Targets metadata from the Director repository, verify that there are no delegations.
7. If checking Targets metadata from the Director repository, check that no ECU identifier is represented more than once.
8. If checking Targets metadata from the Director repository, and the ECU performing the verification is the Primary ECU, check that all listed ECU identifiers correspond to ECUs that are actually present in the vehicle.

5.4.4.7. How to resolve delegations

To properly check Targets metadata for an image, an ECU SHALL locate the metadata file(s) for the role (or roles) that have the authority to sign the image. This metadata might be located in the top-level Targets metadata, but it could also be delegated to another role or to multiple roles. Therefore, all delegations SHALL be resolved using the following recursive procedure, beginning with the top-level Targets metadata file.

1. Download and check the current metadata file, following the procedure in Section 5.4.4.6. If the file cannot be loaded, or if any verification step fails, abort the delegation resolution, and indicate that image metadata cannot be found because of a missing or invalid role.
2. If the current metadata file contains signed metadata about the image, end the delegation resolution and return the metadata to be checked.
3. If the current metadata file was reached via a terminating delegation and does not contain signed metadata about the image, abort the delegation resolution for this image and return an error indicating that image metadata could not be found.
4. Search the list of delegations, in listed order. For each delegation:
   (a) Check if the delegation applies to the image being processed. For the delegation to apply, it SHALL include the hardware identifier of the target, and the target name SHALL match one of the delegation's image paths. If either of these tests fail, move on to the next delegation in the list.

(b) If the delegation is a multi-role delegation, follow the procedure described in Section 5.4.4.8. If the multi-role delegation is terminating and no valid image metadata is found, abort the delegation resolution and return an error indicating that image metadata could not be found.
(c) If the delegation is a normal delegation, perform delegation resolution, starting at step 1. Note that this could recurse an arbitrary number of levels deep. If a delegation that applies to the image is found but no image metadata is found in the delegated roles or any of its sub-delegations, simply continue on with the next delegation in the list. The search is only completed or aborted if image metadata or a terminating delegation that applies to the image is found.

5. If the end of the list of delegations in the top-level metadata is reached without finding valid image metadata, return an error indicating that image metadata could not be found.

5.4.4.8. Multi-role delegations

It is possible to delegate signing authority to multiple delegated roles as described in [TAP-3]. Each multi-role delegation effectively contains a list of ordinary delegations, plus a threshold of those roles that SHALL be in agreement about the non-custom metadata for the image. All multi-role delegations SHALL be resolved using the following procedure. Note that there could be sub-delegations inside multi-role delegations.

1. For each of the roles in the delegation, find and load the image metadata following the procedure in Section 5.4.4.7.
2. Inspect the non-custom part of the metadata loaded in step 1:
   (a) Locate all sets of roles that have agreeing (i.e., identical) non-custom metadata and "SHALL match" custom metadata. Discard any set of roles with a size smaller than the threshold of roles that SHALL be in agreement for this delegation.
   (b) Check for a conflict. A conflict exists if there is more than one agreeing set of roles, yet each set has different metadata. If a conflict is found, choose and return the metadata from the set of roles that include the earliest role in the multi-delegation list.
   (c) If there is no conflict, check if there is any single set of roles with matching non-custom metadata. If there is, choose and return the metadata from this set.
   (d) If no agreeing set can be found that meets the agreement threshold, return an error indicating that image metadata could not be found.

6. References

6.1. Normative References

[**NFC**]	Davis, M. and M. Duerst, "Unicode Standard Annex #15: Unicode Normalization Forms", Oc
[**RFC2119**]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119
[**RFC5198**]	Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, DOI 10.
[**TAP-3**]	Kuppusamy, T., Awwad, S., Cordell, E., Diaz, V., Moshenko, J. and J. Cappos, "The Update
[**TAP-4**]	Kuppusamy, T., Awwad, S., Cordell, E., Diaz, V., Moshenko, J. and J. Cappos, "The Update
[**TUF-spec**]	Samuel, J., Mathewson, N., Condra, G., Diaz, V., Kuppusamy, T., Awwad, S., Tobias, S., Wri

6.2. Informative References

[**CR-OTA**]	Barry, K., "Automakers Embrace Over-the-Air Updates, but Can We Trust Digital C
[**DEPLOY**]	Members of the Uptane Community, "Uptane Deployment Best Practices", n.d..
[**IN-TOTO**]	"in-toto: A framework to secure the integrity of software supply chains", October 201
[**MERCURY**]	Kuppusamy, T., Diaz, V. and J. Cappos, "Mercury: Bandwidth-Effective Prevention
[**PEP-458**]	Kuppusamy, T., Diaz, V., Moore, M., Pühringer, L., Locke, J., DeLong, L. and J. Ca
[**UPTANEESCAR**]	Kuppusamy, T., Brown, A., Awwad, S., McCoy, D., Bielawski, R., Mott, C., Lauzon,
[**USATODAY**]	O'Donnell, B., "Your average car is a lot more code-driven than you think", June 201

Author's Address

Uptane Community Members of the Uptane Community Joint Development Foundation Projects, LLC, Uptane Series (c/o Prof. Justin Cappos) 6 MetroTech Brooklyn, NY 11201 USA EMail: uptane-standards@googlegroups.com