DEPRECATED AS OF 12/1/19.
Please go to the Deployment pages at
https://uptane.github.io/deployment-considerations/index.html

Uptane Deployment Considerations (v2019.03.02)

Note: A blue highlighted title indicates that the text has been moved into a Markdown file. If the main section title is blue, it means the entire section has been moved. If any sub sections are in blue it means only parts of the section have been moved.

Disclaimer

This is a living document, in the sense that is likely to be continually updated in the near future. Therefore, if you start a deployment based off the current state of the document, please be advised that you may need to update some parts of your deployment as these changes occur. We are releasing this document now so that you may begin to familiarize yourself with the larger pieces of deployment. We hope to make the document stable in the near future, after which point, any significant revisions should be far and few in between.

The views and conclusions contained herein are the authors' and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Department of Homeland Security (DHS) or the US government.

Introduction

This instruction manual guides OEMs and suppliers on how to deploy and use Uptane in production. Unlike the Implementation Specification, which describes the way to build an Uptane-compliant client, or the design document, which explains the rationale behind Uptane, the high level goal of this document is to explain how to setup, operate, integrate, and adapt Uptane to work in a variety of environments. As a result, this document mainly deals with five types of issues:

First, there are maintenance operations that would be performed on a regular schedule (Section A). These operations include updating metadata and images, and preparing new vehicles to use Uptane.

Second, there are operations that would be performed only once to setup Uptane (Section B). These operations include setting up the time server, as well as director and image repositories.

Third, there are operations that would be performed in exceptional cases (Section C). These operations include rolling back updates, replacing ECUs, or recovering from a key compromise.

Fourth, our prescriptions are based on our assumptions about deployment configurations that are likely to be used by OEMs and suppliers (Section D). However, Uptane is designed to be flexible, and can be configured differently to meet your own custom requirements. These

requirements include using Uptane with other protocols (e.g., UDS, OMA-DM), or managing updates on a fleet of vehicles.

Fifth, we discuss techniques that can be used to increase security while using Uptane (Section E), but which are not required for compliance.

Our recommendations in this document are based on extended discussions with OEMs and suppliers on the (invitation-only) Uptane Discussion Forum and in our workshops. We appreciate their participation and contributions.

We assume that the readers of this document have familiarized themselves with the Implementation Specification.

The keywords "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Whenever you see text in berry red, please be advised that this text describes an example of a particular way to implement a feature, and is included to make the discussion more concrete. However, an OEM is free to use other techniques to implement the same feature, as long as the resulting behavior is the same.

A. Regular maintenance operations

In this section, we discuss how to perform regular maintenance operations, such as updating metadata and images, or preparing an ECU to run Uptane. Since these operations are performed on a regular basis, it is important to ensure they be performed in a systematic manner so that software updates are delivered securely to ECUs.

A.1 Updating metadata and images (Copy in this section was moved to Markdown Normal Ops file)

One of these maintenance operations is to update metadata and images. In order to ensure the secure delivery of updates, an OEM SHOULD perform the following steps whenever a new update is delivered. First, the OEM verifies the authenticity and integrity of new images delivered by its suppliers. Second, the OEM tests whether the images work as intended, before releasing them to end-user vehicles.

A.1.1 What suppliers should do

In order to prevent updates from being tampered with by man-in-the-middle attackers, images MUST be delivered from the tier-1 supplier to the OEM in a manner that provides an extremely high degree of confidence in the timeliness and authenticity of the files provided. This may entail any manner of technical, physical, and / or personnel controls.

An OEM and its suppliers MAY use any transport mechanism in order to deliver these files.

For example, an OEM MAY maintain a private web portal where metadata and / or images from suppliers can be uploaded. This private server MAY be managed by either the OEM or the tier-1 supplier, and SHOULD require authentication, so that only certain users are allowed to read and / or write certain files.

Alternatively, the OEM and its suppliers MAY use email, or courier mail.

If the supplier signs its own images, then it delivers all of its metadata, including delegations, and associated images. Otherwise, if the OEM signs images on behalf of the supplier, then the supplier needs to update ONLY images, and the OEM is responsible for producing signed metadata. Regardless of whether it is the OEM or suppliers that produce signed metadata, the release counters associated with images SHOULD be incremented, so that attackers who compromise the director repository are not able to rollback to obsolete images (see Section E.1).

Regardless of the transport mechanism used to deliver images, the most important thing is for the OEM to somehow ensure that the images are authentic, and have not been tampered with by man-in-the-middle attackers. The OEM SHOULD verify these images using some out-of-band mechanism, so that the authenticity and integrity of these images can be double-checked.

For example, to obtain a higher degree of assurance, and for additional validation, the OEM MAY also require the supplier's update team to send a PGP / GPG signed email to the OEM's security team listing the cryptographic hashes of the new files.

Alternatively, the OEM MAY require that updates be transmitted via DVD delivered by bonded and insured couriers. To validate the provided files, the OEM and a known contact at the supplier MAY have a video call where the cryptographic hashes of the metadata and / or images are provided by the supplier, and confirmed by the OEM.

An OEM SHOULD perform this verification even if a trusted transport mechanism is used, because the OEM cannot be sure whether the transport mechanism has been compromised. If the suppliers have signed metadata, then the OEM SHOULD verify metadata and images by checking version numbers, expiration timestamps, delegations, signatures, and hashes, so that it can be sure that the metadata matches the images.

A.1.2 What the OEM should do

After a tier-1 supplier has delivered new images to the OEM, and the OEM has somehow verified the authenticity and integrity of these images, the OEM SHOULD test new metadata and images before releasing them, in order to ensure that the images work as intended on end-user vehicles. To do so, It SHOULD use the following steps.

First, the OEM SHOULD add these metadata and images to the image repository. It SHOULD also add information about these images to the inventory database, including any dependencies and conflicts between images for different ECUs. Both of these steps are done, to make the new metadata and images available to vehicles.

Optionally, if images are encrypted on demand per ECU, then the OEM SHOULD ensure that the director repository somehow has access to the original, unencrypted images, so that automated processes running the director repository are able to encrypt them in the first place.

It does not matter how the original, unencrypted images are stored on the director repository. For example, they MAY be stored unencrypted, or they MAY be encrypted using a master key that is known by the automated processes. See Section A.2.1 for more details.

Second, the OEM SHOULD test the updated metadata and images on reserved vehicles, before releasing them to all vehicles in circulation, so that it can verify whether these images work as intended. To do so, it MAY instruct the director repository to first install the updated images on these reserved vehicles.

Finally, the OEM SHOULD update the inventory database, so that the director repository is able to instruct appropriate ECUs on all affected vehicles on how to install these updated images.

A.2 How an ECU should be prepared to support Uptane (moved to the ECU markdown file)

In order to enable ECUs to securely perform over-the-air software updates (SOTA), another operation that would be performed regularly is preparing new ECUs to support Uptane. The following pieces of code and data SHOULD exist on every ECU when a new vehicle is assembled, so that the ECU is able to perform either full or partial verification, and thus securely update software.

Every ECU contains its software at the time it is manufactured. See Section 8 of the Implementation Specification for more details about what this software SHOULD include in order to perform either full or partial verification.

In order to run Uptane, every ECU MUST also include certain data. This data includes:

- The latest copy of metadata at the time of ECU manufacture.
    - For partial verification secondaries, this is only the root and targets role metadata files from the director repository.
    - For primaries and full verification secondaries, this is a complete set of metadata files from both repositories. In order to enable full verification, it MUST also include the map file containing the mapping of all images to both the image and director repositories (see Sections 2.5 and 3.8 of the Implementation Specification).
- The public keys of the time server.
- The latest downloaded time signed by the time server, so that Uptane can verify the time when the ECU is updated over-the-air for the first time, and is thus not susceptible to freeze attacks from the very beginning.
- The *ECU key*, or a private key belonging to this ECU, used to sign ECU version manifests, and decrypt images. An OEM is free to use either symmetric or asymmetric ECU keys: the advantages and disadvantages of each option is discussed shortly (Section A.2.1). There will be a separate key for signing the ECU manifest and decrypting images. Both the encryption and signature keys are unique to the ECU and will be collectively referred to as the ECU keys.

Finally, the inventory database SHALL be used to record relevant details about the ECUs on this vehicle, so that the director repository is able to update them. See Section B.2.2.6 for more details.

A.2.1 Using symmetric or asymmetric ECU keys

The OEM MAY use either symmetric or asymmetric ECU private keys. However, the OEM SHOULD be careful in choosing one over the other, because each choice has a different security / performance trade-off.

If symmetric ECU keys are used, then the benefit is faster decryption of images, and signing of the vehicle version manifest. However, the downside is that, since the director repository MUST have access to symmetric ECU keys to be able to encrypt images, a compromise of this repository means that attackers may be able to steal these keys, and thus impersonate affected ECUs in the future. (See Section C.4.1 for how the director repository may detect this impersonation.) This downside of using symmetric keys may greatly complicate recovering from an ECU key compromise (see Section C.4.1). Asymmetric ECU keys do not have this downside, because only public keys are stored on the inventory database and, thus, attackers cannot impersonate affected ECUs in the future.

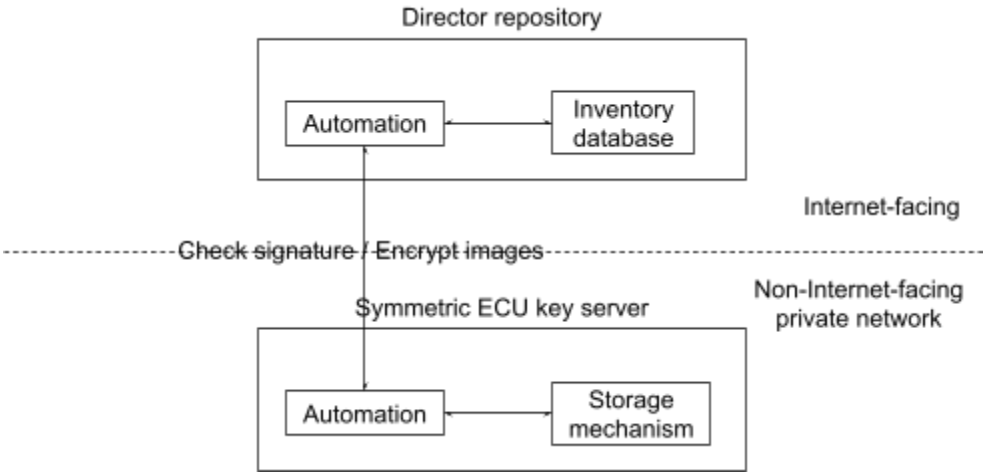

Figure A.2.1a: An arrangement that an OEM SHOULD use when using symmetric ECU keys. See the text for more details.

If the OEM uses symmetric ECU keys, it SHOULD use the arrangement illustrated in Figure A.2.1a in order to significantly reduce the risk of attackers stealing these keys if the directory repository is compromised. Instead of storing these ECU keys in the inventory database on the directory repository, the OEM SHOULD store all of them on a separate *symmetric ECU key server* that is not connected to the Internet. By doing this, compromising the director repository does not immediately yield these keys.

The symmetric ECU key server is allowed to perform only two operations. The first operation simply checks whether signatures on a vehicle version manifest are correct. The second operation would encrypt images for an ECU, given a list of filenames to unencrypted images, and the ECU identifier. In order for the symmetric ECU key server to be able to encrypt

images, the OEM SHOULD periodically populate its storage mechanism with unencrypted images using an out-of-band channel (e.g., DVDs, USB sticks). Both operations SHOULD accept only fixed-length inputs, and both operations SHOULD have been heavily fuzzed and reviewed before use in production, to make it harder for attackers to exploit security flaws. A unique symmetric key SHOULD be used per ECU, so that a key compromise is limited only to that ECU.

In order to prevent loss of confidential information when the symmetric ECU key server is compromised, the OEM MAY require that none of the original images be stored unencrypted on the server. Instead, the OEM MAY encrypt the original images using a master key. This master key is also accessible to the symmetric ECU key server, perhaps using a Hardware Security Module (HSM), so that the master key is not leaked. In any case, the server is able to decrypt original images, and then encrypt them per ECU using its symmetric key.

In summary, using symmetric keys makes the common case (i.e., decryption or digital signatures) faster, but the rare case (i.e., recovering from a repository compromise) more expensive. An alternative solution is to use asymmetric keys, so that a repository compromise does not provide attackers access to private ECU keys. However, the trade-off is now the opposite. The common case (i.e., installation of new updates) is slower, but the rare case less expensive (because a repository compromise does not require updating ECU keys on all vehicles).

A.2.2 Using full or partial verification

In order to ensure the safety of the vehicle using over-the-air software updates, an ECU SHOULD perform either full or partial verification, depending on the type of ECU.

Primaries MUST perform full verification, because their safety is critical to the overall safety of the vehicle. Safety-critical secondaries, whose compromise can jeopardize the safety of the vehicle, SHOULD also perform full verification. All other secondaries SHOULD perform partial verification, so that man-in-the-middle attackers cannot execute arbitrary software attacks on these secondaries. It is up to the OEM to decide which secondaries are safety-critical. Due to the threat of arbitrary software attacks, it is strongly RECOMMENDED that any secondary that performs neither full nor partial verification SHOULD NOT be updated over-the-air.

An implementation is Uptane-compliant if: (1) all primaries perform full verification, (2) all secondaries that are updated over-the-air perform either full or partial verification, and (3) all ECUs that do not perform either full or partial verification cannot be updated over-the-air.

See Section 8 of the Implementation Specification for more details about full or partial verification.

A.3 Backup and garbage collection on the image repository (in Normal Operations Markdown files)

Finally, the OEM SHOULD regularly perform backup and garbage collection of the metadata and images on the image repository. This is done so that the OEM is able to safely recover from a repository compromise, and ensure that the repository has enough storage

space. To do so, an OEM MAY use either the following steps, or its own corporate backup and garbage collection policy.

First, an automated process SHOULD store every file on the image repository, as well as its cryptographic hash on a separate, offline system. This automated process SHOULD also store a copy of the inventory database from the director repository on this offline system. This allows administrators to detect and recover from a repository compromise.

Second, the automated process SHOULD remove expired metadata from the image repository, in order to reclaim storage space. If the OEM is interested in supporting delta updates for vehicles that have not updated for a long time, then the automated process SHOULD NOT remove images associated with expired metadata, because these images MAY be needed in order to compute delta images (see Section D.2).

B. Setting up Uptane for the first time (Unless noted otherwise, content here was moved to the Repositories Markdown file)

In this section, we discuss the one-time operations that an OEM and its suppliers MUST perform when they set up Uptane for the first time. In particular, they SHOULD correctly configure the director and image repositories, as well as the time server, so that the impact of a repository / server compromise is limited to as few ECUs as possible.

B.1 What suppliers should do

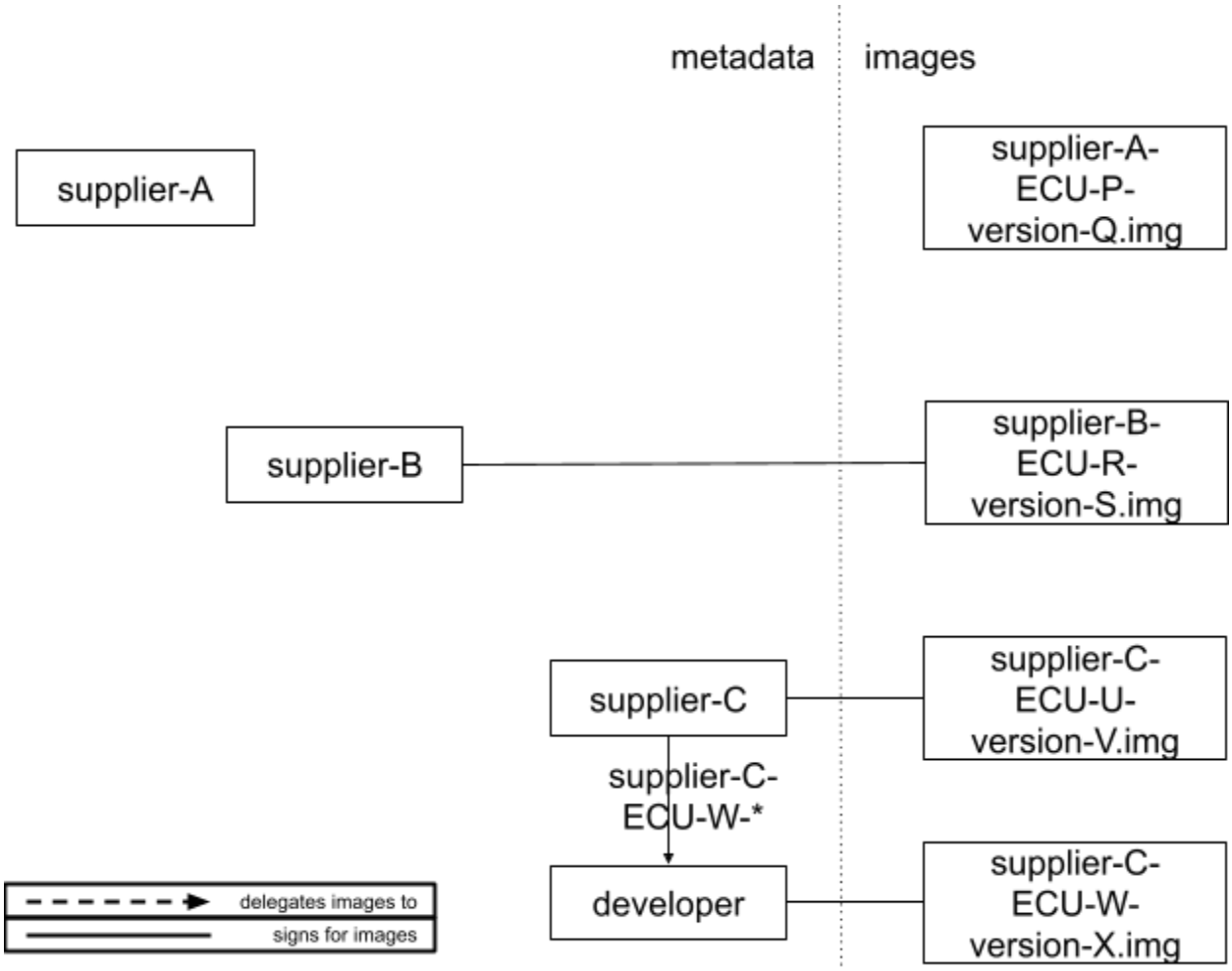

Figure B.1a: Suppliers are free to ask the OEM to sign images on its behalf (supplier A), or sign them itself (supplier B). In the latter case, it MAY also delegate some or all of this responsibility to others (supplier C).

Either the OEM or a tier-1 supplier SHOULD sign for images for ECUs produced by that supplier, so that ECUs never install unsigned images, and thus open themselves to arbitrary software attacks. An OEM would decide whether a tier-1 supplier SHOULD sign its own images. If the OEM signs images on behalf of the supplier, then the supplier MUST only deliver update images to the OEM (Section A.1.1). Otherwise, if the supplier signs its own images, it MUST first set up roles and metadata using the following steps:

1. Generate a number of offline keys used to sign its metadata. In order to provide compromise-resilience, these keys MUST NOT be accessible from the image repository. The supplier MUST take great care to secure these keys, so that a key compromise affects only some, but not all, of its ECUs. The supplier MUST use the threshold number of keys chosen by the OEM. (See Section B.2.3.4 for examples.)

2. Optionally, delegate images to members of its organization (such as its developers), or to tier-2 suppliers (who MAY further delegate to tier-3 suppliers). Delegatees SHOULD recursively follow these same steps.
3. Set an expiration timestamp on its metadata using a duration prescribed by the OEM (Section B.2.3.5).
4. Register its public keys with the OEM using some out-of-band mechanism (e.g., telephone calls, or certified mail).
5. Sign its metadata using the digital signature scheme chosen by the OEM (Section B.2.4.1).
6. Send all metadata, including delegations, and associated images to the OEM (Section A.1.1).

A tier-1 supplier and its delegatees MAY use the Uptane repository and supplier tools to produce these signed metadata.

B.2 What the OEM should do

The OEM MUST set up and configure the time server, as well as the director and image repositories. By using both the director and image repositories, the OEM is able to provide both compromise-resilience and on-demand customization of vehicles. However, as we noted earlier, the OEM MUST configure these repositories and server in such a way that the impact of a repository / server compromise is limited to as few ECUs as possible. The OEM MAY use its own private infrastructure, or cloud computing, to host these backend services.

B.2.1 Time server (This copy has not been moved to Markdown as I was unsure if it required changes)

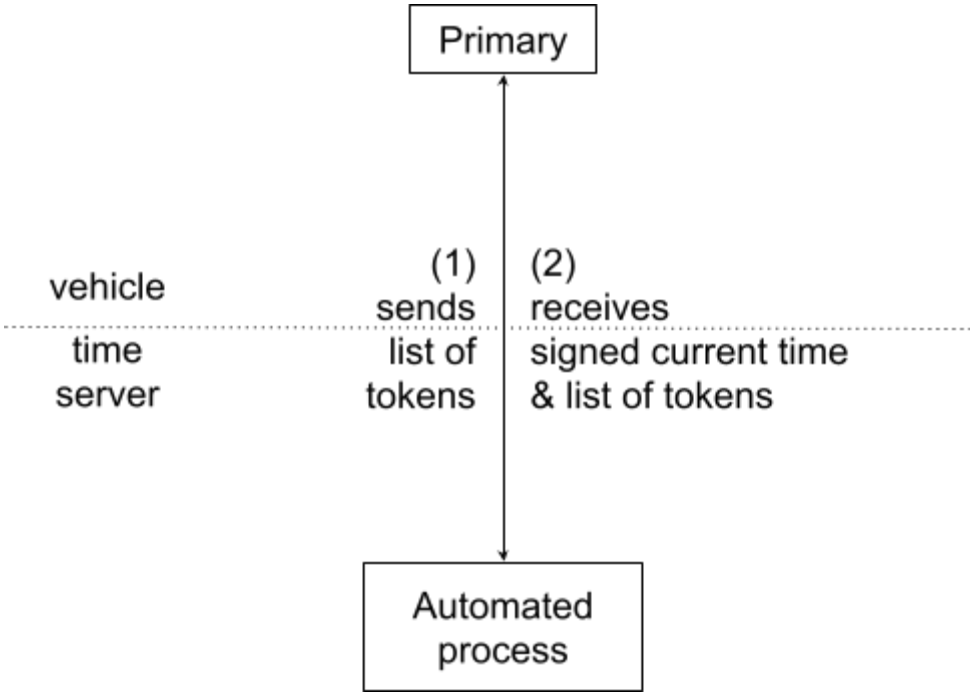

Figure B.2.1a: How primaries interact with the time server.

In order to prevent or limit freeze attacks, the OEM will build the time server following Section 7 of the Implementation Specification. This is because the time server is used by ECUs to tell whether metadata, and thus images, have expired. As illustrated in Figure B.2.1a, this time server is controlled by an automated process. There are only two steps involved when a primary requests the current time from the time server.

First, the primary sends an automated process on the time server a list of tokens.

Second, the automated process signs this list of tokens together with the current time, and sends this to the primary for verification. An OEM MAY use one or more keys to sign the current time.

For examples of how these messages MAY be encoded between the primary and time server, see Section 7 of the Implementation Specification.

An OEM is free to define as it desires the *public API* used by primaries to download the current time from the time server. The API MAY require authentication, depending on the OEM's requirements.

B.2.2 Director repository

In order to provide on-demand customization of vehicles, the OEM MUST also build the director repository, following Section 6 of the Implementation Specification as well as the upcoming prescriptions. Unlike the image repository, the director repository: (1) is managed by

automated processes, (2) uses online keys to sign targets metadata, (3) does not delegate images, (4) generally provides different metadata to different primaries, (5) MAY encrypt images per ECU, and (6) produces new metadata on every request by primaries.

B.2.2.1 Steps to initialize the repository

In order to initialize the repository, an OEM SHOULD perform the following steps:

1. Set up the storage mechanism. The details depend on the choice of storage mechanism (see Section B.2.4.2). For example, this may entail setting up a ZFS filesystem.
2. Set up the transport protocol. The details depend on the choice of protocol (see B.2.4.3). For example, this may entail setting up an HTTP server with SSL/TLS enabled.
3. Set up the private and public APIs to interact over the chosen transport protocol.
4. Set up the timestamp, snapshot, root, and targets roles.
5. Copy all unencrypted images from the image repository. (See Section A.2.1 for how these images may be stored on the director repository.)
6. Initialize the inventory database with the information necessary for the director repository to perform dependency resolution, or encrypt images per ECU. This information includes: (1) metadata about all available images for all ECUs on all vehicles, (2) dependencies and conflicts between images (Section A.1.1), and (3) ECU keys.
7. Set up and run the automated process that communicates with primaries.

The automated process MAY use the repository tools from our Reference Implementation to generate new metadata.

B.2.2.2 Roles

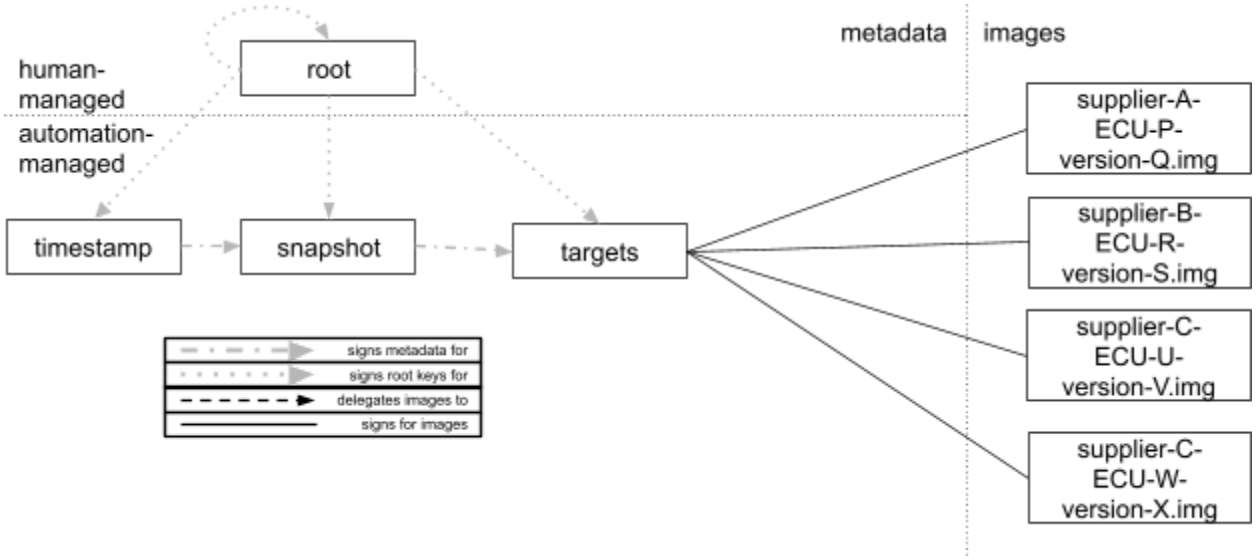

Figure B.2.2.2a: A proposed configuration of roles on the director repository.

Unlike the image repository, the director repository does not delegate images. Therefore, the director repository SHOULD contain only the root, timestamp, snapshot, and targets roles,

as illustrated in Figure B.2.2.2a. In the rest of this section, we will discuss how metadata for each of these roles are produced.

B.2.2.3 Type and placement of keys (moved to Key Management Markdown file)

Repository administrators SHOULD use offline keys to sign the root metadata, so that attackers cannot tamper with this file after a repository compromise.

The timestamp, snapshot, and targets metadata SHOULD be signed using online keys, so that an automated process can instantly generate fresh metadata.

B.2.2.4 Number of keys (moved to Key Management Markdown file)

Since the root role has the highest impact when its keys are compromised, it SHOULD use a sufficiently large threshold number of keys, so that a single key compromise is insufficient to sign its metadata file. Each key MAY belong to a repository administrator. For example, if there are 8 administrators, then at least 5 keys SHOULD be required to sign the root metadata file, so that a quorum is required to trust the metadata.

The timestamp, snapshot, and targets roles MAY each use a single key, because using more keys does not add to security, since these keys are online, and attackers who compromise the repository can always use these online keys.

B.2.2.5 Metadata expiration times

Since the root role keys are expected to be revoked and replaced relatively rarely, its metadata file MAY expire after a relatively long time, such as one year.

The timestamp, snapshot, and targets metadata files SHOULD expire relatively quickly, such as in a day, because they are used to indicate whether updated images are available.

An example number of keys for each role are given in Table B.2.2.5a.

Role	Duration of time until expiration
root	1 year
timestamp	1 day
snapshot	
targets	

Table B.2.2.5a: An example of the duration of time until the metadata for a role expires.

B.2.2.6 The inventory database (moved to Key Management Markdown file)

In order for the director repository to perform its various functions, it MAY also contain an *inventory database* as described in the Implementation Specification that allows it to read and write information about vehicles and ECUs, such as:

1. Unique vehicle identifiers (e.g., VIN).

2. Unique ECU identifiers (e.g., serial numbers).
3. Which ECUs are primaries.
4. The ECU keys used to verify vehicle version manifests, or encrypt images per ECU, as well as associated information, such as the cryptographic algorithm.
5. Metadata about all available images for all ECUs on all vehicles, as well as dependencies and conflicts between images for different ECUs. This information will be used for dependency resolution.

An OEM is free to implement the inventory database however it desires, similar to the storage mechanism for files (Section B.2.4.2).

B.2.2.7 Private API to update images and the inventory database

An OEM SHOULD define a *private API* to the director repository, so that it is able to: (1) upload images, and (2) update the inventory database. This API is private in the sense that only the OEM should be able to perform these actions. The OEM MAY define this API as it wishes.

This API SHOULD require authentication, so that each user is allowed to access only certain information. The OEM is free to use any authentication method, as long as it is suitably strong (e.g., client certificates, or a username coupled with a password or an API key encrypted over TLS, two-factor authentication).

In order to allow automated processes on the director repository to perform their respective functions, without also allowing attackers who compromise the repository to tamper with the inventory database, it is strongly RECOMMENDED that these processes SHOULD be able to read any record in the database, and write new records, but never update or delete existing records.

B.2.2.8 Public API to send updates

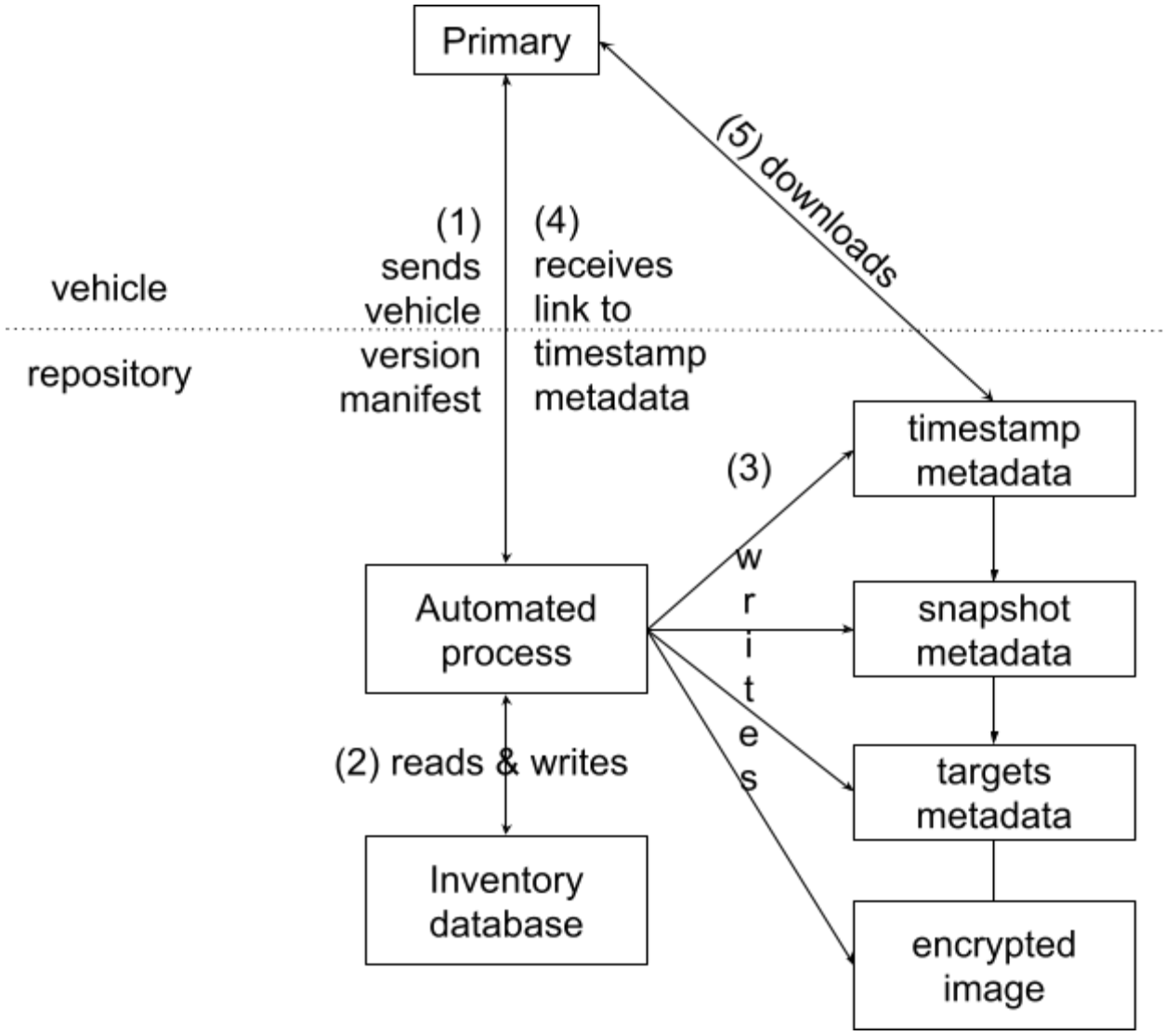

Figure B.2.2.8a: How primaries would interact with the director repository.

An OEM SHOULD define a *public API* to the director repository, so that it is able to send updates to vehicles. The OEM MAY define this API as it wishes.

The OEM MAY use either a push or pull model to send updates, depending on its requirements (Section B.2.2.8.1).

As illustrated in Figure B.2.2.8a, regardless of whether updates are pushed or pulled, in order to send updates at all, this API SHOULD allow a primary to: send a vehicle version manifest (step 1), receive a link to a timestamp metadata file in return (step 4), and download associated files (step 5). The API MAY require authentication, depending on the OEM's requirements. The entire series of steps is elaborated in Section B.2.2.8.2.

B.2.2.8.1 To push or pull updates

Uptane supports both the pushing or pulling of updates to primaries, and leaves the choice to the OEM. The difference between both models is mostly about whether a running vehicle can be told to immediately download an update (via a push), or can wait until a pull occurs.

Either way, the OEM can control how often updates are pushed to or pulled by vehicles. In the push model, the OEM can push an update to a vehicle whenever it likes, as long as the vehicle is online. In the pull model, the OEM can configure the frequency at which primaries pull updates. In most realistic cases, there will be little practical difference between the two models. It is true that if a pull-based OEM configured pulls to only occur once a month, then vehicle updates might be delayed by up to a month, but this would be a counterproductive action. Similarly, a push-based OEM might fail to re-attempt to contact off-line vehicles, but again, this would be a counterproductive action.

Either way, there is no significant difference in resistance to denial-of-service (DoS) attacks or flash crowds. In the push model, a vehicle can control how often updates are pushed to it, so that vehicles can withstand DoS attacks even if the repository has been compromised. In the pull model, the repository can similarly stipulate when vehicles SHOULD download updates, and how often, so that the repository, too, can withstand DoS attacks.

B.2.2.8.2 Sending an update

Regardless of whether updates are pushed or pulled, there are five steps involved in sending an update from the director repository to a primary (see Figure B.2.2.8a).

First, the primary sends an automated process on the director repository its latest vehicle version manifest.

Second, the automated process performs dependency resolution. It reads from the inventory database associated information, such as ECU identifiers and keys, about this vehicle. It checks that the signatures on the manifest are correct, and adds the manifest to the inventory database. Then, using the given manifest, it computes which images SHOULD be installed next by these ECUs. It SHOULD record the results of this computation on the inventory database, so that there is a record of what was chosen for installation. If there is an error at any part of this step (e.g., because signatures are incorrect, or the set of updates installed on the vehicle is unusual), then it SHOULD also record this, so that the OEM can be alerted to a potential risk. Repository administrators MAY then take manual steps to correct the problem (e.g., instructing the vehicle owner to visit the nearest dealership).

Third, using the results of the dependency resolution, the automated process signs fresh timestamp, snapshot, and targets metadata about the images that SHOULD be installed next by these ECUs. Optionally, if the OEM requires it, it MAY encrypt images per ECU, and write them to its storage mechanism. If there are no images to be installed or updated, then the targets metadata SHOULD contain an empty set of targets.

Fourth, the automated process returns to the primary a link to the timestamp metadata file.

Fifth, the primary downloads metadata and images using the link to this timestamp metadata file.

Since the automated process is continually producing new metadata files (and, possibly, encrypted images), these files SHOULD be deleted as soon as primaries have consumed them, so that storage space can be reclaimed. This MAY be done by simply tracking whether primaries have successfully downloaded these files within a reasonable amount of time.

B.2.3 Image repository

Finally, in order to provide compromise-resilience, the OEM will build the image repository following Section 5 of the Implementation Specification, and the upcoming prescriptions. Unlike the director repository, the image repository: (1) is managed by human administrators, (2) uses offline keys to sign targets metadata, (3) MAY delegate images to suppliers, (4) provides the same metadata to all primaries, (5) does not encrypt images per ECU, and (6) updates metadata and images relatively infrequently (e.g., every two weeks, or monthly).

B.2.3.1 Steps to initialize the repository

In order to initialize the repository, an OEM SHOULD perform the following steps:

1. Set up the storage mechanism. The details depend on the choice of storage mechanism (see Section B.2.4.2). For example, this may entail setting up a ZFS filesystem.
2. Set up the transport protocol. The details depend on the choice of protocol (see B.2.4.3). For example, this may entail setting up an HTTP server with SSL/TLS enabled.
3. Set up the private and public APIs to interact over the chosen transport protocol.
4. Set up the timestamp, snapshot, root, and targets roles.
5. Sign delegations from the targets role to all tier-1 supplier roles. The public keys of tier-1 suppliers SHOULD be verified using some out-of-band mechanism (e.g., telephone calls, certified mail), so that the OEM can double-check their authenticity and integrity.
6. Upload metadata and images from all delegated targets roles (including tier-1 suppliers). Verify the metadata and images, and add them to the storage mechanism.

An OEM and its suppliers MAY use the repository and supplier tools from our Reference Implementation to produce new metadata.

B.2.3.2 Roles

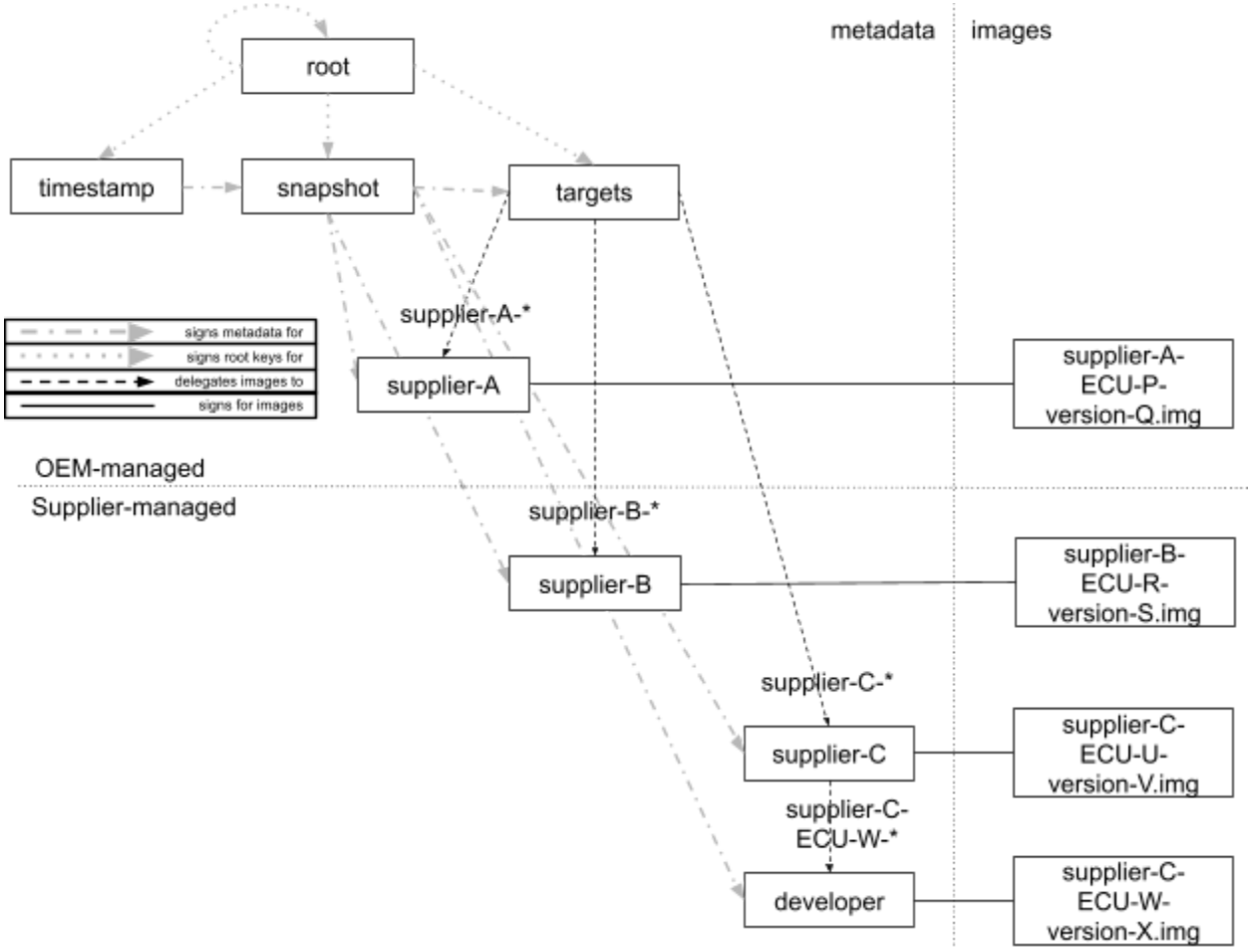

Figure B.2.3.2a: A proposed configuration of roles on the image repository.

Using delegations allows the OEM to: (1) control which roles sign for which images, (2) control precisely which targets metadata vehicles need to download, and (3) distribute, revoke, and replace public keys used to verify targets metadata, and hence, images. In order to set up delegations, an OEM and its suppliers MAY use the configuration of roles illustrated in Figure B.2.3.2a. There are two important points.

First, the OEM maintains the root, timestamp, snapshot, and targets roles. The targets role would delegate images to their respective tier-1 suppliers.

Second, there SHOULD be a delegated targets role for every tier-1 supplier, so that the OEM can: (1) limit the impact of a key compromise, and (2) control precisely which targets metadata vehicles need to download. The metadata for each tier-1 supplier MAY be signed by the OEM (e.g., supplier A), or the supplier itself (e.g., suppliers B and C). In turn, a tier-1 supplier MAY delegate images to members of its organization (such as supplier C, who has delegated a subset of its images to one of its developers), or its tier-2 suppliers (who MAY delegate further to tier-3 suppliers).

Every delegation SHOULD be prefixed with the unique name of a tier-1 supplier, so that the filenames of images do not conflict with each other. Other than this constraint, a tier-1 supplier is free to name its images however it likes. For example, it MAY use the convention "supplier-X-ECU-Y-version-Z.img" to denote an image produced by supplier X, for ECU model Y, and with a version number Z.

B.2.3.3 Type and placement of keys (moved to Key Management Markdown file)

The OEM maintains the keys to the root, timestamp, snapshot, and targets roles.

If a tier-1 suppliers signs its own images, then the supplier maintains its offline keys. Otherwise, the OEM would sign on behalf of the supplier, and thus maintain its keys. Other delegated targets roles, such as developers for a tier-1 supplier, or tier-2 and tier-3 suppliers, are expected to maintain their own keys.

There are two options for signing the timestamp and snapshot metadata, each with the opposite trade-off from the other. In the first option, the OEM uses online keys to sign timestamp and snapshot metadata. The upside is that automated processes can automatically renew the timestamp and snapshot metadata to indicate that there are new targets metadata and / or images available. The downside is that attackers who have compromised a supplier's key, and has access to the image repository (perhaps through a web portal), can cause the image repository to instantly publish malicious images. If they additionally compromise the director repository, then they can execute arbitrary software attacks, because the director repository can be used to select these malicious images on the image repository for installation. Another, less severe downside is that attackers can also execute mix-and-match attacks.

In the second option, the OEM uses offline keys to sign timestamp and snapshot metadata. The upside is that in the aforementioned scenario, attackers cannot immediately cause the image repository to publish malicious images. The downside deals with expiration of timestamp and snapshot. If the timestamp and snapshot metadata expire relatively quickly, then it is more cumbersome to use offline keys to renew their signatures. However, if a longer expiration time is used, then a man-in-the-middle attackers has more time with which to execute freeze attacks, hence defeating the purpose of the timestamp role.

The keys to all other roles on the image repository SHOULD be kept offline. This is because these metadata are expected to be updated relatively infrequently, and so that a repository compromise does not immediately affect full verification ECUs. It does not matter where an offline key is stored (e.g., in a Hardware Security Module, YubiKey, or a USB stick in a safe deposit box), as long as the key is not accessible from the repository. Each key SHOULD be kept separately from others, so that a compromise of one does not affect them all.

B.2.3.4 Number of keys (moved to Key Management Markdown file)

Each role MAY use as many keys as is desired. However, the greater the impact when the keys for a role are compromised, then the greater the number of keys that it SHOULD use. Also, a *threshold* number of keys SHOULD be used, so that a single key compromise is generally insufficient to sign new metadata. To further increase compromise-resilience, each key

SHOULD be unique across all roles. An example number of keys for each role are given in Table B.2.3.4a.

Since the root role has the highest impact when its keys are compromised, it SHOULD use a sufficiently large threshold number of keys. Each key MAY belong to a repository administrator. For example, if there are 8 administrators, then at least 5 keys SHOULD be required to sign the root metadata file, so that a quorum is required trust the metadata.

Since the targets role also a high impact when its keys are compromised, it SHOULD also use a sufficiently large threshold number of keys. For example, 3 out of 4 keys MAY be required to sign the targets metadata file.

Since the timestamp and snapshot roles have a relatively low impact when its keys are compromised, each role MAY use a small threshold number of keys. For example, each role MAY use 1 out of 2 keys to sign its metadata file.

Finally, each delegated targets role SHOULD use at least 1 out of 2 keys to sign its metadata file, so that one key is available in case the other is lost. It is RECOMMENDED that the higher the number of ECUs that can be compromised when a delegated targets role is compromised, then the higher the threshold number of keys that SHOULD be used to sign the role metadata.

Role	Threshold number of keys
root	(5, 8)
timestamp	(1, 2)
snapshot	(1, 2)
targets	(3, 4)
Delegated targets roles (e.g., suppliers)	(1, 2)

Table B.2.3.4a: An example number of keys that MAY be used by each role. Each role uses a threshold of (n, m) keys, where *n* out of *m* signatures are required to trust the signed metadata.

B.2.3.5 Metadata expiration times (moved to Key Management Markdown file)

In order to prevent or limit freeze attacks, the Implementation Specification requires that all metadata files have expiration times. If ECUs know the time, then attackers are unable to indefinitely replay outdated metadata, and hence, images. In general, the expiration date for a metadata file depends on how often it is updated. The more often that it is updated, then the faster it SHOULD expire, so that attackers cannot execute freeze attacks. Even if it is not updated frequently, it SHOULD expire after a bounded period of time, so that stolen or lost keys can be revoked and replaced. Table B.2.3.5a lists an example of expiration times for metadata files on the image repository.

Since the root role keys are expected to be revoked and replaced relatively rarely, its metadata file MAY expire after a relatively long time, such as one year.

The expiration times for timestamp and snapshot metadata depend on: (1) whether online or offline keys are used to sign these metadata, and (2) how often targets metadata and / or images are updated. If online keys are used, or metadata and / or images are updated frequently, then the expiration times should be smaller (e.g., 1 day). Otherwise, the expiration times should be larger (e.g., 1 month). Note, the length of time until expiration is the amount of time a man-in-the-middle attackers can execute freeze attacks. Hence the time should not be too large.

The targets metadata files MAY expire relatively slowly, such as in two weeks or a month, because the OEM is expected to release these files only periodically, after having tested the new metadata files as well as images.

Role	Duration of time until expiration
root	1 year
timestamp	1—30 days
snapshot	
targets	2—4 weeks
Delegated targets roles (e.g., suppliers)	

Table B.2.3.5a: An example of the duration of time until the metadata for a role expires.

B.2.3.6 Private API to upload files

An OEM MUST define a *private API* to the image repository, so that it is able to upload metadata and images. This API is private in the sense that only the OEM should be able to perform these actions. The OEM MAY define this API as it wishes.

This API SHOULD require authentication, so that only authorized users are allowed to write to files. The OEM is free to use any authentication method, as long as it is suitably strong (e.g., client certificates, a username coupled with a password or an API key encrypted over TLS, two-factor authentication).

B.2.3.7 Public API to download files

An OEM MUST define a *public API* to the image repository, so that primaries are able to use this API in order to download metadata and images. The OEM MAY define this API as it wishes.

This API MAY require authentication, depending on the OEM's requirements, so that only authenticated primaries are allowed to download updates. The OEM is free to use any authentication method. The choice of authentication method affects only how certain the OEM can be that it is communicating with authentic primaries, and does not affect how resilient ECUs are to a compromise of the image repository.

B.2.4 Common features (profile)

The OEM MAY use any digital signature scheme, storage mechanism, and transport protocol for the time server as well as the director and image repositories, because Uptane is designed to be agnostic with respect to these details.

B.2.4.1 Digital signature scheme

An OEM MAY use any digital signature scheme that employs asymmetric cryptography in order to sign and verify metadata.

For example, it MAY use the RSA or Ed25519 cryptosystem.

B.2.4.2 Storage mechanism used to contain files

An OEM MAY use any durable storage mechanism to read and write metadata and images, as long as each file can be accessed using a unique name.

For example, it MAY use a filesystem such as btrfs, ext4, NTFS, or ZFS, a key-value database such as Amazon S3, Google Cloud Storage, Microsoft Azure Storage, MongoDB, or Redis, or a relational database such as Microsoft SQL Server, Postgres, or Oracle MySQL.

B.2.4.3 Transport protocol used to send or receive files

An OEM MAY use any transport protocol to send and receive metadata and images to and from the repository.

For example, it MAY use FTP, SFTP, HTTP, or HTTPS.

The OEM MAY use a transport protocol such as SFTP or HTTPS that encrypts metadata and images, so that man-in-the-middle attackers in between the repository and primaries cannot execute eavesdrop attacks. However, this is entirely optional, and up to the discretion of the OEM.

C. Exceptional operations (Section moved to Exceptional Markdown file, except for C.4 to Key Management file)

In this section, we discuss operations that generally are performed only in exceptional cases. These operations should be done carefully, as they may have security implications for software updates.

C.1 Rolling back software updates

Sometimes, it may be necessary to roll back updates. This is because, for example, if the latest updates are less reliable than previous ones, then the OEM may wish to rollback to the previous updates.

By default, Uptane does not allow updates to be rolled back. There are two mechanisms that are used to achieve this. First, Uptane will reject any new metadata file with a version

number lower than what is contained in the previous metadata file. Second, Uptane will reject any new image associated with a release counter that is lower than the release counter of the previous image in the previous targets metadata file. The first mechanism prevents an attacker from replaying an old metadata file. The second mechanism prevents an attacker who compromises the director repository from being able to choose old versions of images, despite being able to sign new metadata. See Figure C.1a for an example.

Figure C.1a: Uptane prevents rollback attacks by rejecting older: (1) metadata files, and / or (2) images.

There are at least two ways to solve this problem, each with different advantages and disadvantages.

Figure C.1b: Uptane allows the installation of different images with the same release counter as what is currently installed.

In the first option, an OEM MAY choose never to increment the release counters of images (see Figure C.1b). Uptane will accept any new image associated with a release counter that is *equal* to the release counter of the previous image in the previous targets metadata file. Therefore, if release counters are never incremented, then all images would have the same release counters. In this situation, an ECU would accept the installation of any compatible image referred in new targets metadata (see Section E.1).

The advantage to this method is that it is simple. It allows the OEM to easily install interchangeable versions of the same image. In the example in Figure C.1b, “foo.img” may simply be a version of “bar.img” containing diagnostic functions. Therefore, the OEM may install either “bar.img” or “foo.img’ on the same ECU. The disadvantage to this method is that it allows attackers who compromise the director repository to execute rollback attacks, because obsolete images may also be installed. Therefore, this method SHOULD NOT be used.

Figure C.1c: Uptane forbids the installation of images with a lower release counter than what is currently installed.

In the second option, an OEM MUST increment the release counter of an image whenever it is critical that an ECU not install images with lower release counters. In the example in Figure C.1c, if an ECU installs “foo.img”, then it cannot install “bar.img”. This is done to prevent the installation of compatible images with lower release counters that have known security vulnerabilities which have been fixed in newer images.

The advantage to this method is that it prevents attackers who compromise only the director repository from being able to execute rollback attacks. However, there are two disadvantages. First, the release counters for images have to be maintained even if role B now signs for images previously signed by role A. This is because release counters are always compared to previous targets metadata files. Second, it is more cumbersome to roll back updates, or deliberately cause ECUs to install older images, because offline keys are used to increment the release counters of these older images in the new targets metadata for the image repository. However, this method SHOULD be preferred, because it is more secure. See Section E.2 for more techniques that can be used to limit rollback attacks when the director repository is compromised.

C.2 Adding, updating, or removing ECUs on a vehicle

Sometimes, it may be necessary for a dealership or mechanic to add, update, or remove ECUs on a vehicle. This may be done in order to support custom configurations of vehicles.

In order to support this use case, one way is for a dealership or mechanic to use an out-of-band communications channel (e.g., perhaps a private, authenticated web site) to

communicate with an OEM about the hardware updates to the vehicle. The dealership or mechanic would then identify the vehicle using its identifier (e.g., VIN), and tell the OEM about the ECUs added to or removed from the vehicle.

Note also that Uptane does not prescribe a protocol for primaries to discover whether ECUs have been added to, updated on, or removed from a vehicle. This is because it is an orthogonal problem to software update security. The advantage of this approach is that an OEM is free to solve this problem using existing solutions that it may already have in place.

The OEM should then decide how to respond to the new information. The OEM can verify the new information by requiring the vehicle to produce a new vehicle version manifest that corresponds to the new hardware. If this is a rare enough use case, then human intervention MAY always be required in order to update the hardware on a vehicle.

C.3 Adding or removing a supplier

Due to changes in business relationships, it may be the case that the OEM may need to add or remove a tier-1 supplier on its repositories.

To add a tier-1 supplier, the OEM SHOULD use the following steps. First, if the supplier signs its own images, then the OEM SHALL add a delegation to the supplier on the image repository following the steps described in Section B.1. Second, the supplier SHALL deliver metadata and / or images to the OEM following the steps in Section A.1.1. Finally, the OEM SHALL add the metadata and images to its repositories, possibly test them, and then release them to affected vehicle following the steps in Section A.1.2.

To safely remove a tier-1 supplier, the OEM SHOULD use the following steps. First, it SHOULD delete the corresponding delegation from the targets role on the image repository, as well as all metadata and images belonging to that supplier, so that their metadata and images are no longer trusted. Second, it SHOULD also delete information about the supplier from the director repository, such as its images as well as its dependencies and conflicts, so that the director repository no longer chooses these images for installation. In order to continue to update vehicles with ECUs originally maintained by this supplier, the OEM SHOULD replace this supplier with another delegation, either maintained by itself or another tier-1 supplier.

Tier-1 suppliers are free to manage delegations to members within its own organizations, or tier-2 suppliers (who may delegate, in turn, to tier-3 suppliers), without involving the OEM.

C.4 Key compromise (This was moved to the .md file on key management, unless noted.)

An OEM and its suppliers SHOULD be prepared to handle a key compromise, because it is an unlikely, not an impossible, event. If the OEM and its suppliers use the recommended number and type of keys in Section B, then this should be a rare event. When it happens, OEM and suppliers could use the following recovery procedures.

C.4.1 ECU keys

If ECU keys are compromised, then the OEM SHOULD manually recall vehicles to replace these keys. This is the safest course of action, because since attackers have compromised these keys, an OEM cannot be sure whether it is remotely replacing keys controlled by attackers or the intended ECUs.

An OEM MAY use the director repository and its inventory database to infer whether ECU keys have been compromised. Since the inventory database is used to record vehicle version manifests, which lists what ECUs on a vehicle have installed over time, the OEM MAY use this information to detect abnormal patterns of installation that may have been caused by an ECU key compromise. However, note that this method is not perfect, because if attackers control ECU keys, then they can also use these keys to send fraudulent ECU version manifests.

C.4.2 Time server (Not moved yet pending resolution of whether Standards decisions have changed this text)

If the time server has been compromised, then the same mechanism used to update images on ECUs can also be used to revoke and replace the public keys of the time server. This is possible, because an image can be used to ship code and / or data (Section D.1).

The OEM would begin by distributing to its tier-1 suppliers the new public keys of the time server. Suppliers would respond by producing new images that use these new public keys instead of the old ones. These images, as well as associated metadata, would be uploaded to the image and directory repositories, where they will be distributed to ECUs for installation.

Note that we do not use the root role to revoke and replace the time server key. This is because Uptane also supports partial verification secondaries. These secondaries do not update any metadata file besides the target metadata file from the director repository. Thus, if the root role were used to revoke and replace the time server key, these secondaries would fail to notice the update.

C.4.3 Director repository

Since the director repository MUST keep at least some software signing keys online, a compromise of this repository can lead to some security attacks, such as mix-and-match attacks (see the Design Overview). Thus, the OEM SHOULD take great care to protect this repository, and reduce its attack surface as much as possible. For example, this MAY be done, in part, by using a firewall. However, if the repository has been compromised, then the following procedure SHOULD be performed, in order to recover ECUs from the compromise.

First, the OEM SHOULD use the root role to revoke and replace the keys to the timestamp, snapshot, and targets roles, because only the root role can replace these keys. Following the type and placement of keys prescribed for the director repository in Section B.2.2.3, we assume that attackers have compromised the online keys to the timestamp, snapshot, and targets role, but not the offline keys to the root role.

Second, the OEM SHOULD manually recall all vehicles in order to replace these keys. This MAY be done by requiring vehicle owners to visit the nearest dealership. A manual recall SHOULD be done even though it could replace these keys for full verification ECUs over-the-air by publishing new metadata. This is because: (1) the OEM SHOULD perform a safety inspection of vehicles, in case of security attacks, and (2) partial verification secondaries are not designed to handle key revocation and replacement over-the-air. In order to update keys for partial verification secondaries, the OEM should overwrite their copies of the root metadata file, perhaps using new images (Section D.1). After having inspected the vehicle, the OEM SHOULD replace and update metadata and images on all ECUs, so that: (1) these images are known to be safe, and (2) partial verification secondaries have replaced keys for the director repository.

C.4.4 Image repository

Following the type and placement of keys prescribed for the image repository in Section B.2.3.3, a key compromise of this repository should be an unlikely event. However, a key compromise here is a much more serious affair, because attackers can now tamper with images without being detected, and thus execute arbitrary software attacks (see the Design Overview). There are two cases for handling a key compromise, depending on whether the key is managed by tier-1 suppliers or its delegatees, or the OEM.

C.4.4.1 Supplier-managed keys

In the first case, a tier-1 supplier or one of its delegatees has faced a key compromise. In this case, the supplier, and its affected delegatees (if any), SHOULD revoke and replace keys. They SHOULD update metadata, including delegations, and images, and send them to the OEM (Section A.1.1).

The OEM SHOULD then manually recall only affected vehicles that run software maintained by this supplier in order to replace metadata and images. This MAY be done by requiring vehicle owners to visit the nearest dealership. A manual recall SHOULD be done, because without trusted hardware (such as TPM), it is difficult to ensure that compromised ECUs can be remotely and securely updated. After having inspected the vehicle, the OEM SHOULD replace and update metadata and images on all ECUs, so that these images are known to be safe.

C.4.4.2 OEM-managed keys

In the second case, the OEM has faced a key compromise. This case is potentially far more serious than the first one, because attackers may be able to execute attacks on all vehicles, depending on exactly which keys have been compromised. If attackers have compromised the keys to the timestamp and snapshot roles, or the targets role, or the root role, then the OEM SHOULD use the following recovery procedure.

First, the OEM SHOULD use the root role to revoke and replace keys for all affected roles. Second, it SHOULD restore all metadata and images on the image repository to a known good state using an offline backup (Section A.3). Third, the OEM SHOULD manually recall all vehicles in order to replace metadata and images. A manual recall SHOULD be done, because

without trusted hardware (such as TPM), it is difficult to ensure that compromised ECUs can be remotely and securely updated.

D. Customizing Uptane for special requirements (to Markdown Customizations file)

In this section, we discuss how OEMs and suppliers may customize Uptane to meet special requirements.

D.1 Updating code and / or data (note this section had not been transferred to Markdown along with the others. It was added on 5/6)

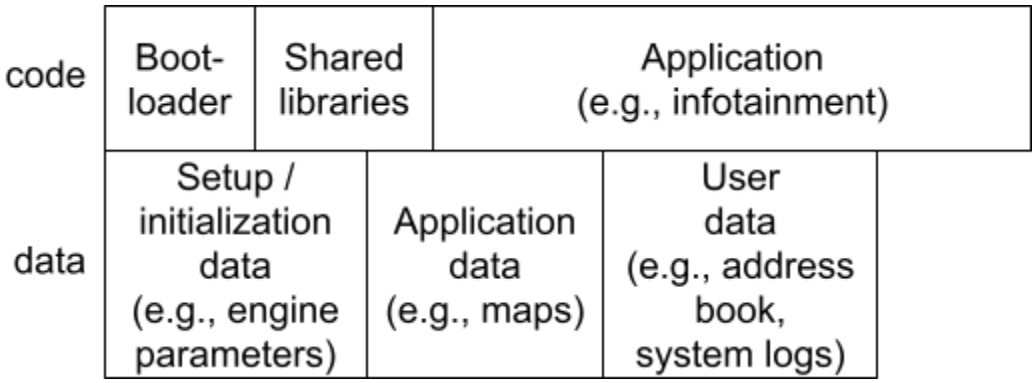

Figure D.1a: An example of how code and / or data may constitute an image.

An image need not necessarily update all code and data on an ECU. The OEM and its suppliers MAY use an image to arbitrarily update code and data on an ECU. For example, an image MAY be used to update only some code but no data, all code and no data, no code and some data, or no code and all data, or all code and data.

Examples of code include the bootloader, shared libraries, and the application (which provides the actual functions of the ECU).

Examples of data include setup or initialization data (such as engine parameters), application data (such as maps), and user data (such as address book, or system logs).

D.2 Using delta updates to deliver images

In order to save bandwidth costs, Uptane allows an OEM to deliver updates as *delta images*. A delta image update contains only the code and / or data that differs from the previous image installed by the ECU. In order to use delta images, the OEM SHOULD make the following changes.

The OEM SHOULD add two types of information 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. This is done so that ECUs know how to apply and verify the

delta image. The director repository SHOULD also be modified to produce delta images, because Uptane does not require it to compute deltas by default. The director repository can use the vehicle version manifest and dependency resolution to determine the differences between the previous and latest images. If desired, then the director repository MAY encrypt the delta image.

As these images are produced on demand by the director repository, primaries SHOULD download all delta and / or encrypted images only from that source. After full verification of metadata, primaries SHOULD also check whether delta images match the targets metadata from the director repository, just as they check whether encrypted images match the targets metadata from the director repository when using non-delta images.

Finally, in order to install a delta image, an ECU SHOULD take one of the actions described in Table D.2a, depending on whether or not the delta image has been encrypted, and if the ECU has additional storage. Note that the OEM MAY use stream ciphers in order to enable on-the-fly decryption by ECUs that have no additional storage space. In this case, the ECU would decrypt the delta image as it is downloaded, then follow the remainder of the steps in the third box below.

Is the delta image encrypted?	**Is there additional storage?**	**Action**
No	**No**	Download the delta image from the primary. Build the latest image by applying the delta image directly to the previous working image. If the latest image does not match the metadata about the full unencrypted image stored on the director repository, abort the update and take the necessary steps to recover or reinstall the previous working image (e.g., by downloading a backup of the previous working image from the primary).
No	**Yes**	Keep the previous working image. Build the latest image by copying the previous image, and applying the delta image. If the latest image does not match the metadata about the full, unencrypted image stored on the director repository, recover the previous working image from additional storage.
Yes	**No**	Download the delta image from the primary, and keep it on volatile memory. Decrypt the delta image in-place on volatile memory.

		Build the latest image by applying the delta image on volatile memory onto the previous working image on non-volatile memory. If the latest image does not match the metadata about the full, unencrypted image stored on the director repository, abort the update and take the necessary steps to recover or reinstall the previous working image (e.g., by downloading a backup of the previous working image from the primary).
Yes	**Yes**	Keep the previous working image. Decrypt the delta image in-place. Build the latest image by copying the previous image, and applying the delta image. If the latest image does not match the metadata about the full, unencrypted image stored on the director repository, recover the previous working image from additional storage.

Table D.2a: The actions an ECU SHOULD take to install a delta image, depending on whether the image has also been encrypted, and whether the ECU has additional storage. See the text for more details.

Dynamic delta updates vs precomputed delta updates

There are two options when computing delta updates. Delta updates can be computed dynamically for each ECU during the installation process (dynamic delta updates), or possible delta images can be precomputed before installation begins (precomputed delta updates).

Dynamic delta updates reduce the amount of data sent on each update, while allowing for fine grained control of what version is placed on each ECU. By using the custom field of the targets metadata, as described in section D.9, the director can be configured to specify a particular version of software for every ECU. Dynamic delta updates allow the director to do file granular resource tracking, which can save bandwidth by only transmitting the delta of the image.

To send dynamic delta updates, the director would compute the delta as described earlier in this section. The computed images would be made available to the primary ECU using the non-standard direction process described in section D.9.

One drawback of dynamic delta updates is that if many ECUs are updating from the same version, computing the delta of each would result in duplicate computation that could be time consuming or take a lot of memory. A possible solution to this is to use precomputed delta updates.

To send precomputed delta updates the director precomputes various probable diffs and makes these available as images. The director then specifies which precomputed delta image to

send to each ECU by using the custom field of targets metadata, as described in section D.9. Precomputing the delta images has the added advantage of allowing these images to be stored on the image repository, which offers additional security against a director compromise.

D.3 Using Uptane with other protocols

Implementers MAY use Uptane in conjunction with other protocols already being used to send updates to the vehicle.

For example, implementers MAY use SSL / TLS to encrypt the connection between primaries and the image and director repositories as well as the time server.

For example, implementers MAY use OMA Device Management (OMA-DM) to send Uptane metadata, images, and other messages to primaries.

For example, implementers MAY Unified Diagnostic Services (UDS) to transport Uptane metadata, images, and other messages between primaries and secondaries.

Any system being used to transport images to ECUs needs to modified only to also transport Uptane metadata, and other messages. Note that Uptane does not require network traffic between the director and image repositories and primaries, as well as between primaries and secondaries, to be authenticated.

However, in order for an implementation to be Uptane-compliant, no ECU can cause another ECU to install an image without performing either full or partial verification of metadata. This is done in order to prevent attackers from being able to bypass Uptane, and thus execute arbitrary software attacks that way. Thus, in an Uptane-compliant implementation, an ECU performs either full or partial verification of metadata and images before installing any image (see Section 8 of the Implementation Specification), regardless of how the metadata and images were transmitted to the ECU.

D.4 Multiple primaries in a vehicle

We expect that the most common deployment configuration of Uptane on vehicles would feature one primary per vehicle. However, we observe that there are cases where it may be useful to have multiple, active primaries in a vehicle. For example, this setup provides redundancy when some, but not all, primaries could fail permanently. The OEM MAY use this setup to design a failover system where one primary takes over when another fails. If so, then the OEM SHOULD take note of the following considerations, in order to prevent safety issues.

It is highly RECOMMENDED that there is a single, active primary in any given vehicle. This is because using multiple, active primaries to update secondaries can lead to problems in consistency, especially when different primaries try to update the same secondaries. If an implementation is not careful, race conditions could cause secondaries to install an inconsistent set of updates (for example, some ECUs would have installed some updates from one primary,

whereas other ECUs would have installed some updates from another primary). This can cause ECUs to fail to interoperate.

If multiple primaries are active in the vehicle at the same time, then each primary SHOULD control a mutually exclusive set of secondaries, so that each secondary is controlled by one primary.

D.5 Atomic installation of a bundle of images

An OEM may wish to require atomic installation of a bundle of images, which means that either all updates in a bundle are installed, or, in case of failure, none of them are installed. Uptane does not provide a way to guarantee atomic installation of bundles, because this problem is out of its scope. It is challenging for ECUs to atomically install a bundle in the face of arbitrary failure: if just one ECU fails to install its update in the bundle for any reason (such as hardware failure), then the guarantee is lost. Furthermore, different OEMs and suppliers already have established ways of solving this problem. Nevertheless, we discuss several different solutions for those who are interested in guidance.

The simplest solution is to use the vehicle version manifest to report to the director repository the failure to atomically install a bundle, and not to retry installation. After receiving the report, it is up to the OEM to decide how to respond. For example, the OEM MAY require the owner of the vehicle to diagnose the failure at the nearest dealership or authorized mechanic.

Another simple solution is for the primary and / or director to retry a bundle installation until it succeeds (bounded by a maximum number of retries). This solution does not require ECUs to perform a rollback in case a bundle is not fully installed. This is an advantage, because ECUs without additional storage cannot perform a rollback to undo a partial bundle installation.

If all ECUs do have additional storage, and can perform a rollback, then the OEM may use a two-phase commit protocol. We assume that a gateway ECU would act as the coordinator, which ensures that updates are installed atomically. This technique should ensure atomic installation as long as: (1) the gateway ECU behaves correctly and has not been compromised, and (2) the gateway ECU does not fail permanently. It is considerably less complicated than Byzantine-fault tolerant protocols, which may have a higher computation / communication overhead. Nevertheless, note that this technique does not provide other security guarantees: for example, the gateway ECU may show different bundles to different secondaries at the same time.

D.6 Fleet management

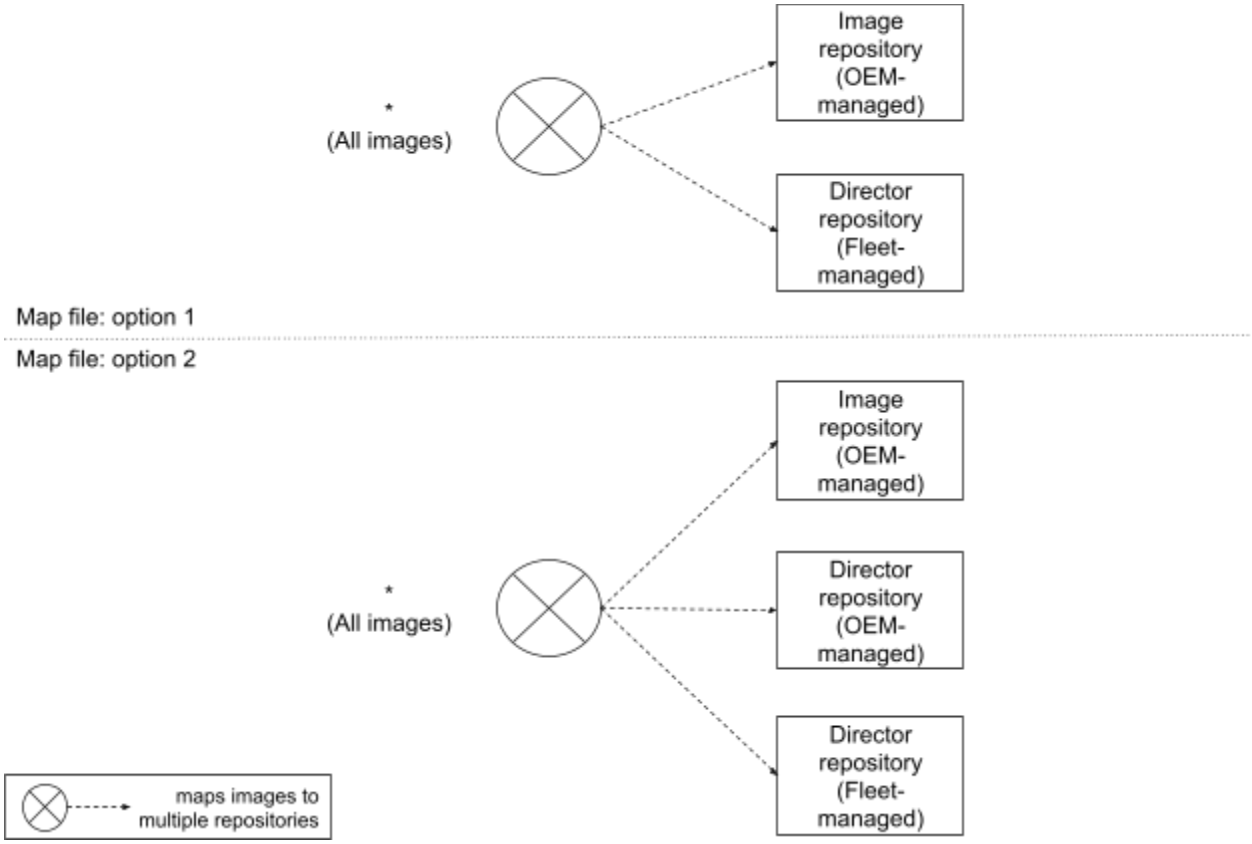

Figure D.6a: Two options for fleet management with Uptane.

Some parties, such a vehicle rental company, or the military, may wish to exercise control on how their own fleet of vehicles are updated. We discuss two options for implementing fleet management using Uptane, illustrated by Figure D.6a. Both options use the map file (Section 2.5 of the Implementation Specification). Choosing between these options depends on whether the fleet manager wishes to have either complete control, or better compromise-resilience.

In the first option, which we expect to be the common case, a fleet manager would configure the map file on ECUs, such that primaries and full verification secondaries would trust an image if and only if it has been signed by both the image repository managed by the OEM, and the director repository managed by the fleet instead of the OEM. Partial verification secondaries would trust an image if and only if it has been signed by this director repository. The upside of this option is that the fleet manager, instead of the OEM, has complete control over which updates are installed on its vehicles. The downside of this option is that if the directory repository controlled by the fleet manager is compromised, then attackers can execute mix-and-match attacks.

In the second option, a fleet manager would configure the map file on ECUs, such that primaries and full verification secondaries would trust an image if and only if it has been signed by the image repository, managed by the OEM, the director repository managed by OEM, and the director repository managed by the fleet. The upside of this option is that attackers cannot execute mix-and-match attacks if they have compromised only the director repository managed by either the OEM or the fleet. The downside of this option is that updates cannot be installed on vehicles unless both the OEM and fleet agree on which images should be installed together. This agreement may require both director repositories to communicate using an out-of-band channel. Using this option also means that partial verification secondaries should be configured to trust the director repository managed by either the OEM or the fleet, but not both, since these secondaries can afford to check for only one signature.

Another option for fleet management is discussed in the next subsection.

D.7 User customization of updates

In its default implementation, Uptane allows only the OEM to fully control which updates are installed on which ECUs on which vehicles. Thus, no third party, such as a dealership, mechanic, fleet manager, or the end-user, has an input as to which updates should be installed. There are very good reasons, such as legal considerations, for enforcing this constraint. However, this capability exists to the point that the OEM wishes to make it available. We discuss two options for doing so.

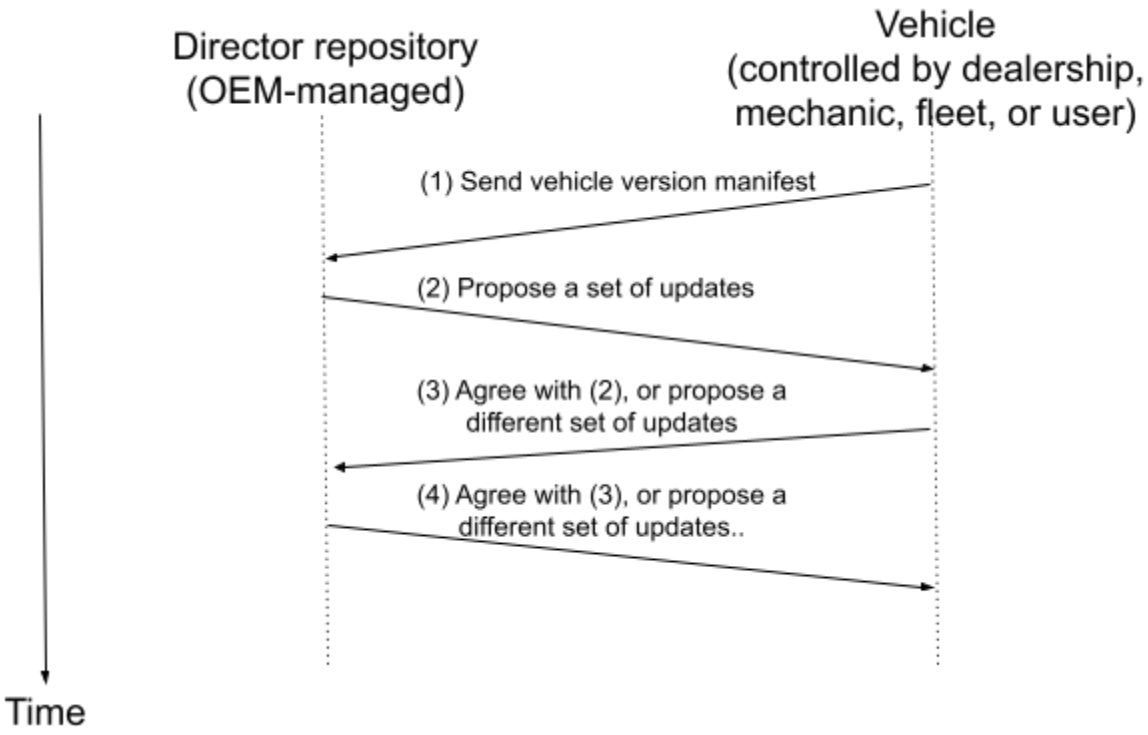

Figure D.7a: An OEM MAY allow a third party to negotiate which updates are installed.

In the first option, an OEM MAY elect to receive input from a third party as to which updates should be installed. A process for how this may be done is illustrated in Figure D.7a. In the first step, the vehicle would submit to the director repository controlled by the OEM its vehicle version manifest, which lists which updates are currently installed. In the second step, the director repository would perform dependency resolution using this manifest, and propose a set of updates. In the third step, the third party would either agree with the OEM, or propose a different set of updates. This step SHOULD be authenticated (e.g., using client certificates, or username and password encrypted over TLS), so that only authorized third parties are allowed to negotiate with the OEM. In fourth step, the OEM would either agree with the third party, or propose a different set of updates. The third and fourth steps MAY be repeated, for a maximum number of retries, until both the OEM and the third party agree as to which updates should be installed.

In the second option, the third party MAY choose to override the root of trust for ECUs, provided that the OEM makes this possible. Specifically, the third party may overwrite the map and root metadata file on ECUs, so that updates are trusted and installed from repositories managed by the third party, instead of the OEM. The OEM may infer whether a vehicle has done so, by monitoring from its inventory database whether the vehicle has recently updated from its repositories. The OEM MAY choose not to make this option available to third parties by, for example, using a Hardware Security Module (HSM) to store Uptane code and data, so that third parties cannot override the root of trust.

D.8 Accommodating ECUs without filesystems

Currently, the Implementation Specification is written with the implicit assumptions that: (1) ECUs are able to parse the string filenames of metadata and images, and that (2) ECUs may have filesystems to read and write these files. However, not all ECUs, especially partial verification secondaries, may fit these assumptions. There are two important observations.

First, filenames needs not be strings. Even if there is no explicit notion of “files” on an ECU, it is important for distinct pieces of metadata and images to have distinct names. This is so that primaries can perform full verification on behalf of secondaries, which entails comparing the metadata for different images for different secondaries. Either strings or numbers may be used to refer to distinct metadata and images, as long as different “files” have different “file” names or numbers. The image and director repositories can continue to use file systems, and may also use either strings or numbers to represent “file” names.

Second, ECUs need not have a filesystem in order to use Uptane. It is only important that ECUs are able to recognize distinct metadata and images, using either strings or numbers as “file” names or numbers, and that they allocate different parts of storage to different “files.”

D.9 ECU Accessing Non-Standard Directions

Most inputs to ECUs are delivered as signed targets files and are stored on the image directory. These signed targets files are then sent to the ECU by the director. However, there

may be some cases where the inputs required for a particular customization cannot be configured to follow this standard signing process. The inputs may have to vary from the standard signing process because the input is not known in advance. Or, perhaps these instructions need to be customized for each vehicle. Examples of such inputs could be a command line option that turns on a feature in certain ECUs, a configuration sent by a director repository to an ECU, or a director doing a dynamic customization for an ECU. We can collectively call all these non-standard inputs “dynamic directions.” Uptane allows ECUs to access dynamic directions in two different ways, each having particular advantages for different use cases.

Accessing dynamic directions through signed images from the director repository

The first option for providing dynamic directions is to slightly modify the standard delivery procedure described above. A signed image would still be sent to the ECU from the director repository, but with one main difference. Though this file will be signed by the director, timestamp, and metadata roles, it would not be stored on—or validated by—the image repository. As the image repository is controlled by offline keys, it can not validate a file created dynamically by the director.

However, even though the image repository can not sign the file, some security protections are still in place. The ECU would still have rollback protection for a file sent this way as a release counter will still be included in the metadata and would be incremented for each new version. If additional validation is needed, the file could be put on multiple repositories created for this purpose. These repositories could behave similar to the director repository, but would all have separate keys to allow for additional security. The primary ECU will be aware of these extra repositories so it can check for consistency by downloading the image from all repositories, and comparing them.

Adding Dynamic Directions to the Custom Field of Targets Metadata

Another way to provide dynamic directions is to use the custom field of the targets metadata file. This field provides the option to include custom inputs to individual ECUs. Using the custom field is an especially good option for managing small variations in the existing image. For example, a compilation flag to enable a navigation feature might be set on some ECUs, but not on others. The custom field could contain dynamic directions, and additional subfields would help determine for which ECUs the direction is intended. In the flag example above, the director can put the ECU id and the flag into the custom field so the flag will be used during the installation process only on that particular ECU. This custom field can then be included in the targets metadata received by all ECUs. The intended ECU would be able to check for this flag and use it during an installation or update to enable the navigation system.

Using this method of providing dynamic directions offers several ways to secure the system. The targets metadata is created by the director, validated using the timestamp, and stored in the image repository. As an added protection against a compromise of the director role, the custom field could be included in the targets metadata file on the image repository as well. This option works best if there is only a small set of known customizations. However, if there are multiple customization possibilities, the better option would be to store the targets

metadata without the custom field in the image repository, and keep the custom field configured separately and signed by the director.

Picking an option (Efficiency vs Security)

In choosing whether to send dynamic variations from the director repository or through the custom field of targets metadata, one needs to consider two factors: how quickly the dynamic direction needs to be received, and how security-sensitive the receiving ECU may be. If dynamic directions are sent using the custom field of targets metadata, these directions will be downloaded by all ECUs on the car. So, if the direction has a large file size, it could significantly slow down delivery of the metadata. Sending files from the director repository that are signed only for a specific ECU could be a bit more efficient. Yet if the ECUs are connected on a single bus, there is no way to avoid this large quantity of data going to all units.

In terms of security, dynamic direction transmission through the custom field of targets metadata seems to have an advantage, as including directions in the custom field of targets metadata gives the sender the option of storing these directions on the image repository. This would offer protection against a compromised director repository. In contrast, files sent from the director to a specific ECU are only signed by the director. Therefore, should the director be compromised, images signed only by the director could be changed without the ECU knowing.

It is important to consider these tradeoffs when deciding how to send dynamic directions. If dynamic directions are for a security critical ECU, these directions should be sent using the custom field of targets metadata and stored on the image repository. If the dynamic directions are large or there are efficiency concerns, the directions should be sent as signed images from the director repository.

E. Additional security considerations (moved to security considerations markdown file)

In this section, we discuss techniques that can be used to increase security while using Uptane.

E.1 Controlling which images are installed on ECUs

In order to prevent attackers who compromise the director repository from causing one type of ECUs to install the wrong images intended for another type, an OEM and / or its suppliers SHOULD include certain information about images in the targets metadata.

Suppose that attackers have compromised the director repository. If release counters are used (Section C.1), and further mitigating steps are taken (Section E.2), they cannot rollback software updates. Furthermore, without additional key compromise, they cannot cause arbitrary software attacks on primaries and full verification secondaries. However, if an implementation is not careful, attackers can cause ECUs of one hardware type to install images intended for another hardware type. To use an analogy, this is similar to causing Linkedsys routers to install images intended for GetNear routers.

ECU identifiers (e.g., serial numbers) specified in the targets metadata signed by the director repository do not solve this problem, because: (1) they are used by the director repository only to instruct which ECU should install which image, and (2) they are not specified in the targets metadata signed on the image repository, because it is impractical to list all ECU identifiers that pertain to an image.

In order to avoid this problem, the targets metadata about unencrypted images on the image repository SHOULD always include the `TargetsModule.Custom.hardwareIdentifier` attribute (see Section 3.4.1 of the Implementation Specification). A hardware identifier allows an OEM and / or its suppliers to succinctly capture an entire class of ECUs without listing their ECU identifiers one by one. Note that the OEM and / or its suppliers SHALL ensure that hardware identifiers are unique across different hardware types of ECUs, so that attackers who compromise the director repository cannot cause ECUs of one type to install images intended for another type.

Before an ECU installs a new image, it SHOULD always check the hardware type of the image (see Section 8.2 of the Implementation Specification), so that attackers cannot cause ECUs to install randomly chosen images that have not been intended for it.

E.2 Preventing rollback attacks when the director repository is compromised

In Section C.1, we discussed how an OEM and / or its suppliers SHOULD use release counters in order to prevent attackers who compromise the director repository from executing rollback attacks. To further limit these attacks in this scenario, the OEM and / or its suppliers SHOULD also use the following recommendations.

First, they SHOULD diligently remove obsolete images from new versions of targets metadata files uploaded to the image repository. This is done in order to prevent attackers who compromise the director repository from being able to choose these obsolete images for installation. This method has the downside that it complicates updating vehicles that require intermediate updates in order to install the latest updates. For example, an ECU has previously installed image A, and C is the latest image it should install. However, the ECU should first install image B before it installs C, and B has already been removed from the targets metadata on the image repository in order to prevent / limit rollback attacks. Thus, the OEM and / or its suppliers SHOULD balance between the requirement to prevent / limit rollback attacks by removing obsolete images from the targets metadata, and the requirement to support updating ECUs that need to first install these obsolete images before updating to the latest images.

Second, they SHOULD decrease the expiration timestamps in the targets metadata uploaded to the image repository. This is done so that these targets metadata expire more quickly, thus preventing attackers who compromise the director repository from being able to choose these obsolete images. This method has the downside that these targets metadata now need to be updated more quickly. This will not cause accidental freeze attacks as long as an ECU can update both the time from the time server, and metadata from the image repository. Accidental freeze attacks can happen if, for some reason, the ECU is able to update metadata,

but not the time. In this unlikely event, the ECU is able to continue working with the previously installed image, but it would be unable to update to the latest image. The director repository can detect this unlikely event using the vehicle version manifest. In this case, the OEM MAY require the owner of the vehicle to diagnose the problem at the nearest dealership or authorized mechanic.

E.3 Transmitting metadata over vehicular networks

An implementation of Uptane MAY have a primary unicast metadata to secondaries, which means that the primary would send metadata separately to each secondary. However, this has a downside: due to network disruptions, different ECUs may end up seeing different versions of metadata released by repositories at different times.

In order to mitigate this problem, it is RECOMMENDED that a primary use a broadcast network such as CAN, CAN FD, or Ethernet to transmit metadata to all of its secondaries at the same time. Primaries MAY use the `ECUModule.MetadataFiles` message to do so (see Section 8.1 of the Implementation Specification). Note that this still does not guarantee that different ECUs do not end up seeing different versions of metadata released by repositories at different times. This is because network traffic may still get disrupted between primaries and secondaries, especially if they are connected through intermediaries such as gateways. Nevertheless, it should not be worse, and should be better, than unicasting.

If an update is intended to be applied to a gateway itself, it should be updated either before or after (but not while) applying updates to ECUs on the other side of the gateway so as to avoid disruption.

E.4 Checking dependencies and conflicts between installed images

The *dependencies* of an image A for an ECU is a set of other images that MUST also be installed on other ECUs in order for A to work on this ECU. The *conflicts* of an image A for this ECU is a set of other images that MUST not be installed on other ECUs in order for A to work correctly on this ECU. *Dependency resolution* is the process of finding which versions of the latest images and their dependencies can be installed without conflicts.

There are three options for which entities should check dependencies and conflicts:

1. **Only ECUs check dependencies and conflicts.** This information should be included in the targets metadata on the image repository, and should not add substantially to bandwidth costs. The upside is that, without offline keys, attackers cannot cause ECUs to fail to satisfy dependencies and prevent conflicts. The downside is that it can add to computational costs, because dependency resolution is generally an NP-hard problem. However, it is possible to control the computational costs if some constraints are imposed [1][2].
2. **Only the director repository checks dependencies and conflicts.** This is currently the default on Uptane. The upside is that the computational costs are pushed to a

powerful server. The downside is that attackers who compromise the director repository can tamper with dependency resolution.

3. **Both ECUs and the director repository check dependencies and conflicts.** To save computational costs, and avoid having each ECU perform dependency resolution, only primaries and full verification secondaries may be required to double-check the dependency resolution performed by the director repository. Note that this is *not* NP-hard problem, because these ECUs simply need to check that there is no conflict between the director and image repositories. The trade-off is that when primaries are compromised, secondaries have to depend on the director repository.

E.5 Managing dependencies and conflicts between installed images

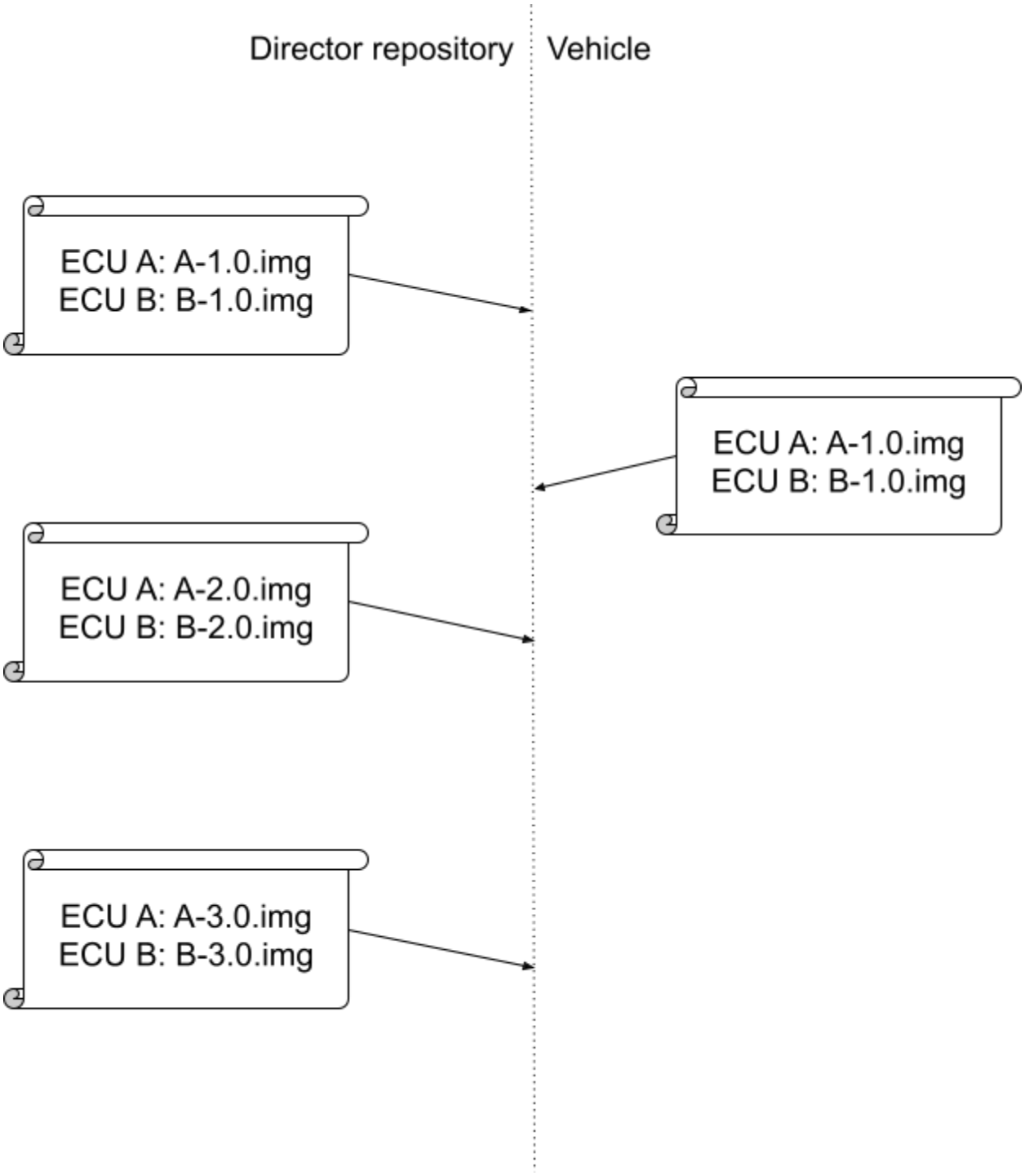

Figure E.5a: A series of hypothetical exchanges between a director repository and a vehicle.

Generally speaking, the director repository SHOULD NOT issue a new bundle that may conflict with images known with complete certainty to have been installed on the vehicle, as determined by the last vehicle version manifest. This is because, due to partial bundle installation “attacks”, any bundle sent after the last vehicle version manifest may have been only partly installed by ECUs. If the director repository is not careful to handle this issue, then the vehicle may end up installing conflicting images, causing ECUs to fail to interoperate.

Consider the series of messages exchanged between a director repository and a vehicle in Figure E5.a. In the first bundle of updates, the director repository instructs ECUs A and B to install the images A-1.0.img and B-1.0.img respectively. Later, the vehicle sends a vehicle version manifest stating that these ECUs have now installed these images.

In the second bundle of updates, the director repository instructs these ECUs to install the images A-2.0 img and B-2.0.img respectively. However, for some unknown reason, the vehicle does not send a new vehicle version manifest in response.

In the third bundle of updates, the director repository instructs these ECUs to install the images A-3.0.img and B-3.0.img. However, the problem is that it has not received a new vehicle version manifest from the vehicle in the meantime, stating that both ECUs have installed the second bundle.

Furthermore, the director repository knows that B-1.0 and C-3.0 conflict with each other. The only thing the director repository can be certain of is that B has installed either B-1.0 or B-2.0, and C has installed either C-1.0 or C-2.0. Thus, the director repository SHOULD NOT send the third bundle to the vehicle, because B-1.0 from the first bundle may still be installed, which would conflict with C-3.0 from the third bundle.

Therefore, the director repository SHOULD NOT issue the third bundle, until it has received a vehicle version manifest from the vehicle that confirms that ECUs B and C have installed the second bundle, which is known to contain images that do not conflict with the third bundle.

In conclusion, the director repository SHOULD NOT issue a new bundle until it has received confirmation, using the vehicle version manifest, that no image known to have been installed conflicts with the new images in the new bundle.

If the director repository is not able to update a vehicle for any reason, then it SHOULD raise the issue to the OEM.

E.6 Decoding ASN.1 messages

If an OEM chooses to use ASN.1 to encode and decode metadata and other messages, then it SHOULD be careful in decoding ASN.1 messages. This is because improper decoding of ASN.1 messages may lead to arbitrary code execution or denial-of-service attacks. For example, see CVE-2016-2108 and attacks on a well-known ASN.1 compiler.

In order to avoid these problems, OEMs and suppliers SHOULD use ASN.1 decoders that have been comprehensively tested via unit tests and fuzzing, whenever possible.

Furthermore, following best practices, we recommend that the DER encoding is used instead of BER and CER, because DER provides a unique encoding of values.

E.7 Balancing storage performance and security on ECUs

Many ECUs use EEPROM which, practically speaking, can be written to for a limited number of times. This limits how often these ECUs can be updated.

In order to analyse this problem, let us recap what new information should be downloaded in every software update cycle (Section 8.1 of the Implementation Specification):

1. The primary writes and sends the latest vehicle version manifest to the director repository.
2. All secondaries write and send to the primaries fresh tokens for the time server.
3. All ECUs download, verify, and write the latest downloaded time from the time server.
4. All ECUs download, verify, and write metadata from the director and / or image repositories.
5. At some point, ECUs download, verify, and write images.
6. At some point, ECUs install new images. Then, they sign, and write the latest ECU version manifests.

Let us make two important observations.

First, it is not necessary to continually refresh the time from the time server, separately from a software update cycle. This is because: (1) the time may not be successfully updated from the time server, (2) an ECU MUST be able to boot to a valid image even if its metadata has expired, and (3) it is necessary to check only that the metadata for the latest downloaded updates has not expired.

Care must be taken by implementers not to update time information too frequently. Writing time information once per day, will cause flash with 10K write lifetime to wear out within about 27 years. If valid time server metadata is always written to the same block (which is unlikely since the old metadata is likely to be retained before the new metadata validated), this may cause unacceptable wear. Implementers should reason about the lifetime of devices and their likely update patterns when using technologies with limited writes.

However, note that there is a trade-off between frequently updating the time from the time server (and thus, exhausting EEPROM), and the efficacy of preventing freeze attacks by a compromised director repository. If it is important to more frequently update the time, and thus better prevent freeze attack, despite using EEPROM, note that there are ways to make more efficient use of EEPROM. For example, the ECU may write data to EEPROM in a circular fashion, so that the EEPROM is exhausted less quickly.

Second, it is not necessary for ECUs to write and sign an ECU version manifest upon every boot or reboot cycle. At a minimum, an ECU should write and sign a new ECU version manifest only upon the successful verification and installation of a new image.

E.8 Expected computational resources consumed by Uptane

When deploying any system, it is important to think about the costs involved. Those can roughly be partitioned into computational, network (bandwidth), and storage. To understand

these costs, this section gives a rough sense of how those costs vary in based upon the deployment scenario. These numbers are not authoritative, but are meant to give a rough sense of order of magnitude costs.

A primary will end up retrieving and verifying any updated metadata from the repositories it communicates with. This usually means that an image repository and a director repository will be contacted. Whenever an image is added to the image repository, a primary will download a new targets, snapshot, and timestamp role file. Rarely, the root file will be updated and may need to also be verified. Verifying these requires checking a signature on each of the files. Whenever the vehicle is requested to install an update, the primary also receives a new piece of targets, snapshot, and timestamp metadata. Rarely, the root file will be updated and may need to also be verified. As before, verification requires a signature check. A primary also must compute the secure hash of all images it will serve to ECUs. The previous, known good version of all metadata files must be retained. It is also wise to retain any images until secondaries have confirmed installation.

A full verification secondary is nearly identical in cost to a primary. The biggest difference is that it has no need to store, retrieve, or verify an image that is not destined for it. Other costs are fundamentally the same.

A partial verification secondary merely retrieves targets metadata when it changes and images it will install. This requires one signature check and one secure hash operation per software installation.

Note also that time server costs are typically one signature verification per ECU per time period of update (e.g., daily). This cost varies based upon the algorithm and needs to be measured based upon the algorithm.

Acknowledgements

We would like to thank Lois Anne DeLong for her efforts on this document. Our work on Uptane is supported by U.S. Department of Homeland Security grants D15PC00239 and D15PC00302.