Politecnico di Torino

Academic Year 2024/2025

Security Verification and Testing - 02TYAUV

Formal Verification of the Uptane Automotive OTA Framework with ProVerif

---

Project Report

Nobili Luca - s331461
Prof. Riccardo Sisto
PhD Student Simone Bussa

Contents

1 Abstract

Modern vehicles increasingly rely on software to manage critical systems, making secure software updates essential for both safety and functionality. Traditional update mechanisms, originally designed for general-purpose computing, are insufficient for the automotive domain due to the presence of heterogeneous Electronic Control Units (ECUs), intermittent connectivity, and the high impact of failures.

Uptane is a compromise-resilient framework specifically designed to secure over-the-air (OTA) updates in automobiles. It adds strategic features to the state-of-the-art software update framework, The Update Framework (TUF), to address automotive-specific requirements.

This report details a formal verification of core Uptane protocol interactions using ProVerif, an automated cryptographic protocol verifier. The proposed model incorporates necessary abstractions, including a focus on a single update cycle and a simplified representation of key management, to facilitate tractable analysis under the Dolev-Yao attacker model. Verification primarily assesses critical security properties through correspondence queries, particularly focusing on the prevention of arbitrary software installation on ECUs. The study also discusses the model's scope and limitations concerning the verification of other common attack vectors, such as freeze, rollback, and mix-and-match attacks, in light of the chosen abstractions.

2 Introduction

The increasing complexity and connectivity of modern vehicles significantly increases their exposure to cybersecurity threats. Electronic Control Units (ECUs), responsible for critical vehicle functions, are frequently updated over-the-air (OTA) to patch vulnerabilities and enhance performance. However, inadequately secured OTA mechanisms can become vectors for exploitation, potentially leading to catastrophic safety failures, privacy breaches, and significant financial liabilities for manufacturers, thereby endangering passengers and infrastructure.

Uptane addresses these security concerns by providing a robust, resilient software update framework that is specially adapted to the automotive domain. Based on the principles of The Update Framework (TUF), Uptane adapts to vehicle-specific constraints such as intermittent connectivity, various ECU capabilities, and stringent real-time requirements.

Initiated in 2016 through collaboration among researchers from the NYU Tandon School of Engineering, the University of Michigan Transportation Research Institute (UMTRI), the Southwest Research Institute (SWRI), and key industry stakeholders, Uptane has been formally adopted by OEMs and suppliers representing over 70 countries.

Given the high stakes involved in automotive cybersecurity, a rigorous and provably secure update mechanism is paramount. This report presents a formal verification of the Uptane protocol using the ProVerif tool, a widely recognized automated verifier for cryptographic protocols, by examining its security properties against realistic threat scenarios.

3 The Update Framework (TUF): A Foundation for Secure Software Updates

TUF is an open-source specification and set of tools designed to secure software update systems against a wide range of attacks. It operates on the fundamental principle that even if parts of an update system are compromised, the client should still be able to distinguish legitimate updates from malicious ones. Unlike traditional methods that often rely on a single signing key, TUF employs a multi-role, hierarchical approach to distribute trust and responsibilities, significantly enhancing resilience against compromise.

Key Features and Security Properties of TUF

TUF primarily provides a **compromise-resilient framework** for secure software updates. Its key security properties and benefits include:

- **Robust Attack Protection:** Defends against common update attacks such as rollback, freeze, mix-and-match, and attacks involving compromised signing keys.
- **Distributed Trust:** Minimizes the impact of key compromises by spreading signing authority across multiple roles. This is often achieved using **key threshold signatures**, where for a piece of metadata to be considered valid, it must be signed by at least a pre-defined number 'M' of a total 'N' available unique keys for that role (an M-of-N policy). This prevents a single compromised key from being sufficient to authorize actions and allows for operational flexibility.
- **Flexible Delegation:** Enables repository owners to securely delegate update responsibilities for specific components, supporting scalable and distributed ecosystems.

- **Metadata Freshness:** Ensures clients always receive the latest and most accurate metadata, preventing the use of stale or manipulated information.

Key Roles in TUF

TUF defines a set of cryptographic roles, each with specific responsibilities and signing keys, to collectively secure the update process. These top-level roles are:

- **Root Role:** This is the ultimate root of trust for the entire system. It defines and validates the public keys and signature thresholds for all other top-level roles, including itself. Root keys are typically stored offline and are rarely used, making them highly secure. A compromise of the Root key is very difficult to recover from, thus its extreme protection. The Root role is also responsible for managing **key rotation** for all roles, including itself. Key rotation is the practice of replacing old cryptographic keys with new ones, either periodically or in response to a suspected compromise, to limit the potential damage from key exposure over time.
- **Targets Role:** This role is responsible for signing the metadata that lists all the actual software "target" files (e.g., applications, libraries, firmware) available for update. This metadata includes cryptographic hashes and sizes of the target files, ensuring their integrity. The Targets role can also delegate trust to other sub-roles for managing specific subsets of targets.
- **Snapshot Role:** This role provides a consistent view of all current metadata files across the repository. It signs metadata that lists the version numbers and hashes of all `targets.json` files and any delegated targets metadata.
- **Timestamp Role:** This is the most frequently updated role, primarily responsible for signing a small metadata file that contains the hash and version of the latest `snapshot.json` file. It has a short expiration time, allowing clients to quickly detect if the repository is not being updated or if a freeze attack is in progress.

By separating these responsibilities and requiring clients to verify signatures across these different roles, TUF significantly enhances the security and resilience of software update systems.

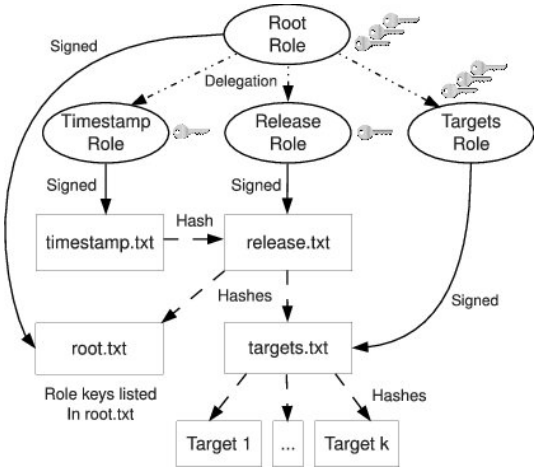

Figure 1: TUF Role Dependencies

4 Uptane: Secure Software Updates for Automotive Systems

Uptane builds upon the robust foundation of TUF, tailoring it specifically for the demanding environment of ground vehicles. Its primary purpose is to empower Original Equipment Manufacturers (OEMs) and suppliers to implement a secure update mechanism that shields connected units within vehicles from cyber threats.

4.1 Core Principles and Automotive-Specific Enhancements

Uptane is fundamentally engineered for resilience, designed to minimize the impact of attacks even if parts of the update system are compromised. It offers comprehensive protection against a wide array of threats, ranging from unauthorized access to update content and denial of service against legitimate updates, to more severe scenarios such as an attacker inducing ECU failures or gaining control over ECUs or the entire vehicle. The framework operates under the realistic assumption that adversaries may possess significant capabilities, including intercepting and

altering network communications, or even compromising server infrastructure (like Director or Image repositories) and in-vehicle ECUs.

To achieve its robust security goals, Uptane meticulously ensures the authenticity, integrity, and freshness of all metadata and software images. This is primarily accomplished through the rigorous application of digital signatures, consistent version control, and explicit timestamping. Furthermore, a cornerstone of Uptane's defense strategy is the architectural separation of responsibilities among different roles and repositories. This distribution of trust significantly fortifies its security posture against sophisticated attacks, including those that might involve compromised signing keys or backend servers.

4.2 Uptane Components

The Uptane ecosystem is characterized by several key roles and entities, distributed between server-side repositories and the in-vehicle client systems.

4.2.1 Repositories and Server-Side Roles

- **Director Repository:** This repository plays a crucial role in authorizing updates. It consults an inventory database (containing vehicle-specific ECU information and software versions) to produce signed metadata that instructs ECUs about images they are to install. This metadata is generated on demand and can be tailored to individual vehicles.
- **Image Repository:** This repository serves as the storage location for the actual software images and their corresponding TUF-style metadata (Targets, Snapshot, Timestamp, Root). It allows OEMs and their suppliers to upload these artifacts, making them accessible for download by vehicles.
- **Root Role :** Operates with the same high-security responsibilities as in TUF. It signs metadata that distributes and revokes the public keys for verifying Root, Timestamp, Snapshot, and Targets role metadata originating from the Image Repository and the Director (for its specific metadata). Its keys are the ultimate source of trust.
- **Timestamp Role :** Signs metadata indicating whether new metadata or images are available on a repository (both Image and Director). This provides freshness guarantees, ensuring clients are aware of the latest state.
- **Snapshot Role :** Signs metadata that lists the versions of all Targets metadata files for a repository, ensuring a consistent view.
- **Targets Role :** Signs metadata describing the actual update images, including their hashes and sizes.
- **Supplier Involvement:** Although suppliers do not constitute a distinct repository role, they are essential within the update ecosystem. Suppliers create and deliver software images along with the related metadata, typically submitting them to the Image Repository through TUF's delegation mechanism.

4.2.2 In-vehicle Components

- **Primary ECU:** A designated ECU within the vehicle, often with more computational power and reliable network connectivity (e.g., the Telematics Control Unit). The Primary is responsible for:
  - Performing **full verification** of all downloaded metadata and images from both the Director and Image repositories.
  - Securely receiving and validating time information.
  - Downloading and installing updates for itself and potentially for Secondary ECUs it communicates with.
  - Aggregating a **vehicle version manifest** (a comprehensive list of software versions on all ECUs it is aware of) and sending it to the Director repository. This manifest allows the Director to make informed decisions about necessary updates.
- **Secondary ECU:** These are other ECUs within the vehicle, which may have limited resources or direct connectivity to the external repositories. Secondary ECUs:
  - Typically receive updates via the Primary ECU.
  - Are expected to perform at least **partial verification** of updates, focusing on the metadata relevant to their own software. The level of verification can vary based on the ECU's capabilities.
  - Report their current software versions to the Primary for inclusion in the vehicle version manifest, **vehicle version report** (ECU manifest).

4.3 Control Flow and Verification Processes

Uptane defines robust verification processes to ensure the end-to-end security of software updates, accommodating the different capabilities of ECUs. [1]

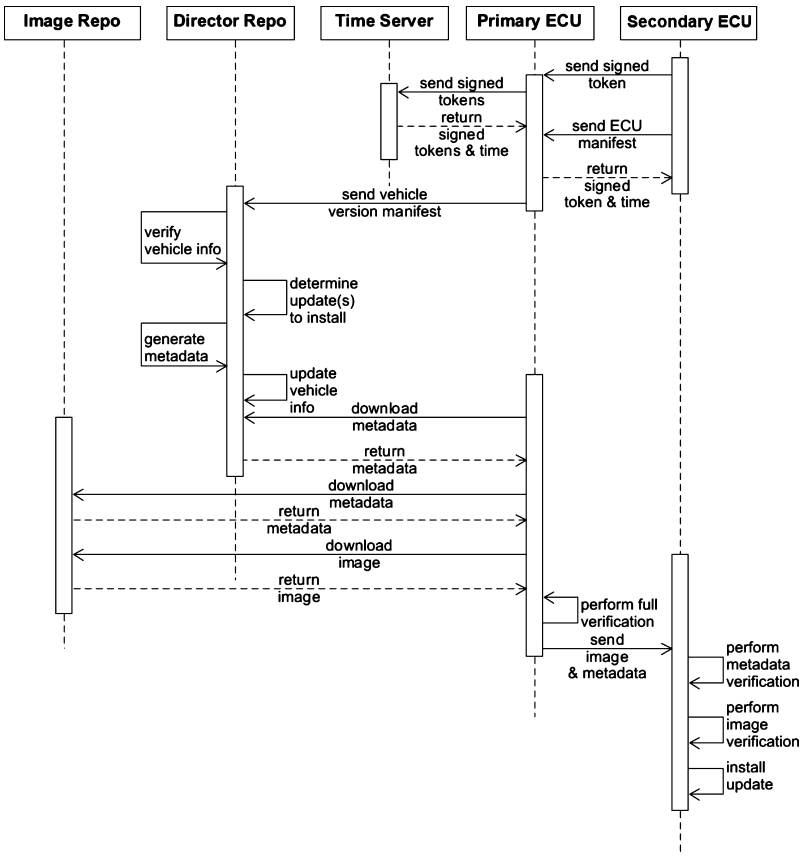

Figure 2: Overview of the Uptane Message Flow for Software Updates.

Figure 2 illustrates several key phases of the Uptane update process:

- **Time Synchronization:** Initially, ECUs ensure they have a secure sense of time. The diagram shows the Primary ECU and Secondary ECU interacting with the Time Server by sending "signed tokens" (which, as detailed in Section 4.3.4, are typically nonces in a challenge-response protocol) and receiving back "signed tokens & time." This step is crucial for validating the freshness of metadata.
- **Vehicle State Reporting:** The Secondary ECU reports its local manifest information ("send ECU manifest") to the Primary ECU. The Primary ECU aggregates this information with its own to form a complete Vehicle Version Manifest, which it then sends to the Director Repository ("send vehicle version manifest").
- **Update Determination by Director Repository:** Upon receiving the Vehicle Version Manifest, the Director Repository verifies the vehicle's information ("verify vehicle info") and determines if any updates are necessary for the ECUs in that vehicle ("determine update(s) to install"). Based on this, the Director generates the appropriate metadata (e.g., new Targets metadata).
- **Primary ECU Update Acquisition:**
  1. The Primary ECU first receives "update vehicle info" from the Director, which includes the Director's instructions and signed metadata about which updates are targeted for the vehicle.
  2. It then proceeds to "download metadata" from both the Director Repository and subsequently the Image Repository. This involves fetching and verifying the chain of trust (Root, Timestamp, Snapshot, and specific Targets metadata) from each repository as detailed in the full verification process.
  3. Once all metadata are verified, the Primary ECU downloads the actual software image ("download image") from the Image Repository.
- **Secondary ECU Update Process:**
  1. The Primary ECU, having performed full verification, forwards the validated image and relevant metadata ("send image & metadata") to the Secondary ECU.

[1] For the full specification see the Uptane Standard https://Uptane.org/docs/2.1.0/standard/Uptane-standard.

2. The diagram indicates the Secondary ECU then proceeds to "perform full verification." Although full verification is the ideal scenario if the Secondary ECU has sufficient resources, it is important to note that the Uptane standard mandates partial verification as the minimum requirement (as detailed in Section 4.3.2). The subsequent steps "perform metadata verification" and "perform image verification" are essential components of either verification level, culminating in "install update" if all checks pass.

The following subsections detail these verification processes and metadata checks.

4.3.1 Full Verification (Performed by Primary ECUs)

Primary ECUs undertake a comprehensive verification procedure for all updates. The general sequence is:

1. **Establish Current Time:** Securely obtain and validate the current time. This is paramount for checking the validity period of all signed metadata, preventing replay attacks with expired (but otherwise valid) metadata.
2. **Verify Director Repository Metadata:** The Primary first interacts with the Director repository to determine what updates are intended for it and the ECUs it manages.
   (a) Download and cryptographically verify the Director's Root metadata file using its pre-trusted Director root keys.
   (b) Download and verify the Director's Timestamp metadata.
   (c) Download and verify the Director's Snapshot metadata.
   (d) Download and verify the Director's Targets metadata. This metadata lists the specific images the Director intends for the ECUs in this vehicle. If this indicates no new updates are needed, the process may conclude for targets listed here.
3. **Verify Image Repository Metadata:** For each actual software image to be downloaded (as specified by the Director's Targets metadata), the Primary verifies the corresponding metadata from the Image Repository.
   (a) Download and verify the Image Repository's Root metadata file using its pre-trusted Image Repo root keys.
   (b) Download and verify the Image Repository's Timestamp metadata.
   (c) Download and verify the Image Repository's Snapshot metadata.
   (d) Download and verify the relevant Image Repository Targets metadata file and potentially delegated targets files) that describe the actual image.
4. **Cross-Verification and Image Download:**
   (a) **Consistency Check:** The Primary ensures that the metadata for an image obtained from the Director's Targets file (e.g., image hash, size) is consistent with the metadata for the same image found in the Image Repository's Targets file. This critical step ensures that the Director's instruction aligns with what the Image Repository provides, preventing certain mix-and-match attacks if one repository is compromised.
   (b) **Image Download and Verification:** If all checks pass, the Primary downloads the actual software image. It then verifies the integrity and authenticity of the image by checking its cryptographic hash and size against the trusted Targets metadata from both Image repository.
5. **Installation** Once an image has been verified, it can be installed.

Full verification by Secondary ECUs is also encouraged if their capabilities permit.

4.3.2 Partial Verification (Performed by Secondary ECUs)

Secondary ECUs, particularly those with limited resources or limited direct connectivity to repository servers, are required to perform at least partial verification for Over-The-Air (OTA) updates. This process ensures that even these ECUs can cryptographically verify the authenticity and integrity of the updates intended for them by the Director.

The Uptane standard specifies the minimal set of steps an ECU SHALL perform for partial verification:

1. **Load and Verify Secure Time:** The ECU must first establish the current time from a secure and verified source. This is crucial for checking the validity period of the metadata and preventing replay attacks. (This process is detailed in Section 4.3.4).
2. **Download and Check Director's Targets Metadata:** The ECU SHALL download and check the Targets metadata file from the Director repository. This check must follow the standard procedures for verifying Target metadata.

Extensibility of Checks: The standard notes that this two-step procedure is the *smallest set* of Uptane checks permissible for a Secondary ECU. Implementers can and often should include additional checks for enhanced security. For example:

- A Secondary ECU could also fetch and verify the Director's Root metadata before checking the Targets metadata. This would provide the Secondary with a secure mechanism to learn about and validate rotations of the Director's signing keys for the Targets role. Without this, if the Director's Targets key is rotated, the Secondary might not be able to verify new Targets metadata.

This partial verification ensures that Secondaries, despite their limitations, still play an active role in validating their updates against authoritative metadata from the Director repository.

4.3.3 Image Installation and Verification Checks (Post-Metadata Verification)

Once the necessary metadata (from Director and/or Image repositories) has been successfully verified by either a Primary (full verification) or Secondary (partial verification), the ECU proceeds to verify the actual software image before installation. This step ensures that the image file itself is authentic, intact, and appropriate for the specific ECU. According to the Uptane Standard, the ECU SHALL perform the following checks:

1. **Load Latest Director Targets Metadata:** The ECU loads the latest valid Targets metadata from the Director repository that pertains to it. This ensures the checks are against the most current authoritative information from the Director.
2. **Identify ECU-Specific Metadata:** Within the Director's Targets metadata, the ECU locates the specific metadata entry associated with its unique ECU identifier.
3. **Hardware Identifier Match:** The ECU verifies that the hardware identifier specified in this metadata entry matches its own actual hardware identifier. This prevents an image intended for a different hardware variant from being installed.
4. **Release Counter Check:** The ECU compares the release counter of the image (as specified in the newly verified Targets metadata) against the release counter from the previously installed image's metadata (if such metadata exists and is accessible). The new release counter MUST be greater than or equal to the previous one to prevent rollbacks to versions with known vulnerabilities, unless explicitly authorized by other mechanisms (e.g., creating a new image with an incremented release counter that happens to be an older software version).
5. **Image Decryption (optional):** If the image is encrypted, the ECU decrypts it using an appropriate key.
6. **Cryptographic Hash Verification:** The ECU calculates the cryptographic hash(es) of the image file. These calculated hashes MUST exactly match all corresponding hashes listed for that image in the verified Targets metadata. This ensures that the image has not been corrupted or altered.

The installation of the new software image by the ECU will proceed only once all required checks have been successfully validated.

4.3.4 Timeserver Integration

A secure sense of time is fundamental to Uptane's security model, as all TUF and Uptane metadata files contain expiration timestamps. Typically, ECUs do not possess reliable internal clocks, making them vulnerable to accepting outdated or compromised metadata. Therefore, ECUs require integration with an external, trusted timeserver to securely synchronize and validate the current time, preventing replay or freeze attacks leveraging expired metadata.

- **Mechanism for Secure Time Attestation:** The Uptane standard mandates that ECUs use the "current time or the most recent securely attested time." While the standard allows for various implementations, one detailed mechanism for achieving this, particularly involving a timeserver, can be described as follows:
    1. **Nonce Generation and Collection by Primary:** Each Secondary ECU generates a fresh, unpredictable number (a nonce), signs and sends it to its designated Primary ECU. The Primary ECU also generate its own nonce.
    2. **Request to Timeserver:** The Primary ECU collects all these nonces and forwards them in a request to a trusted timeserver.
    3. **Timeserver's Signed Response:** For each nonce received, the timeserver individually signs a message containing the current time along with the original nonce. It then sends these signed (time, nonce) pairs back to the Primary ECU.

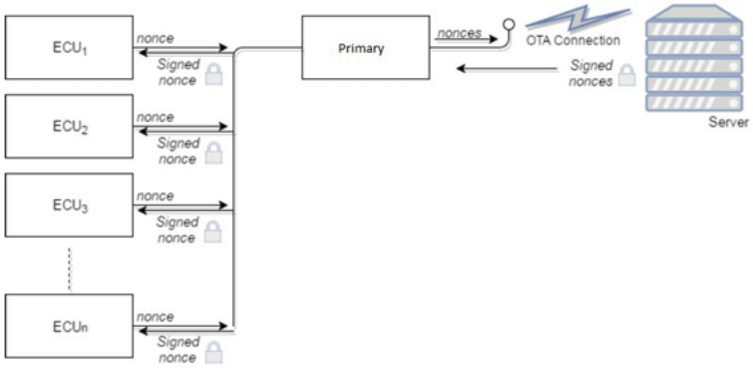

Figure 3: Mechanism for Secure Time Attestation

4. **Primary's Verification and Time Dissemination:** The Primary ECU performs several crucial verification steps on the responses from the timeserver:
   - It verifies the timeserver's signature on each (time, nonce) pair using the timeserver's authentic public key.
   - It check that the nonce sent is also included among the received nonces.
   - It checks that the timestamp provided by the timeserver is consistent across all the signed responses for the current batch of nonces. This ensures a coherent view of time.

   Once these verifications are successful, the Primary ECU considers the time authoritative and securely disseminates this verified current time to its Secondary ECUs. The Secondary ECUs, having sent the nonce, can trust this time as it's tied to their challenge.

This nonce-based challenge-response protocol helps ensure that the time is not only authentic (correctly signed by the timeserver) but also fresh (as it's a response to a recent, unique challenge).

- **Impact on Freshness:** The overall freshness of the update system, primarily guaranteed by the Timestamp role metadata, critically depends on each client's ability to accurately know the current time. Without secure time, an attacker could potentially replay old metadata.

4.4 Uptane Metadata and Checks

Uptane utilizes TUF's metadata structure, with specific applications for both the Director and Image repositories. Clients perform rigorous checks on these files. The checks described below are generally applicable, with minor variations depending on whether the metadata originates from the Image Repository or the Director.

4.4.1 Root Metadata

- **Purpose:** Distributes and revokes public keys for all top-level roles (Root, Timestamp, Snapshot, Targets) of a given repository. It is the ultimate anchor of trust.
- **Key Content:** Public keys for all trusted roles, version number, expiration date, and signatures from Root role keys.
- **Checks Performed by Clients:**
  1. Load the previously trusted Root metadata file (if one exists).
  2. Download the new Root metadata file (up to a defined maximum size).
  3. **Signature Verification for Update:** The new Root metadata (version N + 1) MUST be signed by:
     - A threshold of unique keys specified in the currently trusted Root metadata (version N).
     - A threshold of unique keys specified in the new Root metadata (version N + 1) itself (ensuring consistency for future updates and graceful key rotation).
  4. **Version Check:** The version number of the new Root metadata MUST be greater than or equal to the version of the currently trusted one. (Incrementing version is typical; equality might occur in specific re-signing scenarios).
  5. **Expiration Check:** The current securely attested time MUST be earlier than the expiration timestamp in the new Root metadata.
  6. **Key Rotation Handling:** If the new Root metadata indicates changes to Timestamp or Snapshot keys, the client MUST delete any locally cached Timestamp and Snapshot metadata files, as they would no longer be verifiable.

4.4.2 Timestamp Metadata

- **Purpose:** Provides freshness for the repository. It indicates if there are any new versions of other metadata files (specifically Snapshot) by signing over the hash and version of the latest Snapshot metadata.
- **Key Content:** Version number, expiration date, cryptographic hash(es) and version number of the current Snapshot metadata file, and signature(s) from Timestamp role keys.
- **Checks Performed by Clients:**
  1. Download the Timestamp metadata (up to a defined maximum size).
  2. **Signature Verification:** Verify signatures using the Timestamp role keys loaded from the trusted Root metadata.
  3. **Version Check:** The version number of this Timestamp metadata MUST be greater than or equal to any previously seen Timestamp version from this repository.
  4. **Expiration Check:** The current securely attested time MUST be earlier than the expiration timestamp in this Timestamp metadata.

4.4.3 Snapshot Metadata

- **Purpose:** Provides a consistent view of all Targets metadata files (and their versions/hashes) available on the repository at a specific point in time. This prevents mix-and-match attacks involving different versions of Targets files.
- **Key Content:** Version number, expiration date, a list of all Targets metadata filenames (including delegated targets) along with their respective version numbers. Signed by Snapshot role keys.
- **Checks Performed by Clients:**
  1. Download the Snapshot metadata (up to the size specified in the trusted Timestamp metadata, if provided, otherwise a general maximum).
  2. **Timestamp Consistency:** The cryptographic hash and version number of this downloaded Snapshot metadata MUST match the hash and version listed in the verified Timestamp metadata.
  3. **Signature Verification:** Verify signatures using the Snapshot role keys loaded from the trusted Root metadata.
  4. **Version Check:** The version number of this Snapshot metadata MUST be greater than or equal to any previously seen Snapshot version from this repository.
  5. **Targets Version Consistency:** For each Targets metadata file listed in this new Snapshot metadata, its listed version number MUST be greater than or equal to the version listed for that same Targets file in any previously verified Snapshot metadata (if available). This prevents rollback of individual Targets files.

4.4.4 Targets Metadata

- **Purpose (Image Repository):** Describes the actual software images available for download. Contains cryptographic hashes, file sizes, and potentially custom metadata for each image. Can delegate trust for subsets of images to other Targets roles (e.g., signed by suppliers).
- **Purpose (Director Repository):** Specifies which images (identified by their hashes) a particular vehicle or ECU should install. This metadata is an instruction, not a description of downloadable files from the Director itself.
- **Key Content:** Version number, expiration date, a list of target files (with hashes, sizes for Image Repo; primarily hashes for Director Repo), and potentially custom metadata or delegations. Signed by Targets role keys (or delegated Targets keys).
- **Checks Performed by Clients:**
  1. **Snapshot Consistency:** The version number of each Targets metadata file being processed MUST match what is listed in the trusted Snapshot metadata.
  2. **Signature Verification:** Verify signatures using the Targets role keys loaded from the trusted Root metadata (for top-level Targets) or from the delegating parent Targets metadata (for delegated Targets).
  3. **Expiration Check:** The current securely attested time MUST be earlier than the expiration timestamp.
  4. **Version Check:** Ensure the version number is not a rollback if local state allows this check.

5. **Image Integrity (after download, for Image Repo Targets):** When an actual software image is downloaded, its cryptographic hash and size MUST exactly match the values specified in the trusted Targets metadata from the Image Repository.
6. **Director-Image Repo Cross-Verification (by Primary):** As mentioned in Full Verification, the Primary ensures that the description of an image (e.g., its hash) in the Director's Targets metadata matches the description of that same image in the Image Repository's Targets metadata.

5 Formal Verification in ProVerif

ProVerif is a sophisticated automated verification tool specifically designed for analyzing cryptographic protocols to ensure essential security properties, including secrecy, authentication, privacy, traceability, and verifiability. Its broad support covers a comprehensive range of cryptographic primitives, such as symmetric and asymmetric encryption, digital signatures, hash functions, and other relevant primitives.

A distinguishing feature of ProVerif is its capacity to evaluate protocols that involve an unbounded number of sessions and message spaces, a capability particularly relevant when dealing with complex, large-scale distributed systems. When the analysis identifies potential vulnerabilities, ProVerif attempts to reconstruct attack traces, providing explicit demonstrations of possible protocol executions that violate the intended security properties.

The input for ProVerif analysis consists of carefully structured scripts, typically including the following components:

- **Declarations of operations**: Definitions of cryptographic primitives used in the protocol, such as encryption algorithms, digital signatures, and hash functions.
- **Process macros**: Reusable, modular process definitions that simplify the description of repetitive or common protocol actions.
- **Main process**: A central definition capturing the behavior and interactions of all protocol participants.
- **Queries**: Formal statements specifying the security properties ProVerif must verify, such as the confidentiality of secrets or the authenticity of messages.

ProVerif scripts are expressed in an extended form of the typed pi calculus, a formal language adept at modeling concurrent processes communicating over open, potentially compromised channels like the Internet. Within this calculus, adversaries are represented through the Dolev-Yao threat model, where attackers fully control communication channels and are empowered to intercept, modify, delete, and inject messages. However, attackers' cryptographic operations are limited strictly to those actions enabled by the keys explicitly in their possession.

Furthermore, ProVerif operates under the assumption of perfect cryptography, implying that cryptographic primitives cannot be broken computationally, and attackers must adhere to the defined cryptographic operations.

Despite its powerful capabilities for cryptographic protocol verification, ProVerif is not inherently the perfect or complete solution for formally verifying a comprehensive and multifaceted standard like Uptane. Several inherent characteristics of Uptane, when juxtaposed with ProVerif's operational model and analytical strengths, present significant challenges:

- **Statefulness Management:** Uptane is deeply stateful. Its security relies on the consistent management of version numbers, image installation states across multiple ECUs, nonce tracking for freshness, and complex rollback prevention mechanisms. ProVerif can model some state, but Uptane's security relies on complex state that is spread out and must last over time. Accurately capturing this state in ProVerif is very difficult. Rich state can lead to non-terminating analyses or necessitate abstractions that might oversimplify critical behaviors.
- **Verification of Fine-Grained Policies and System-Level Properties:** Uptane enforces detailed security policies, such as consistency checks between different metadata sources, dependency resolution for updates, and strict rules for key management and rotation. While ProVerif excels at verifying properties like secrecy and authentication of exchanged messages, expressing and verifying these complex, system-wide policy adherence and some aspects of freshness or integrity beyond individual message exchanges can be non-trivial.
- **Abstraction Limitations:** Our formal model of Uptane in ProVerif, as detailed in Sections 5.2.4 ("Model Abstractions and Scope in Setup") and 5.3 ("High-Level Description of Modeled Uptane Processes"), necessarily incorporates several abstractions to achieve analytical tractability. These include simplifications such as the "single update cycle focus," a streamlined vehicle topology, and symbolic representations of time and version progression. While these modeling choices enable focused analysis of core cryptographic exchanges, they can also mean that certain real-world details or complexities of the full Uptane standard are not fully captured. Consequently, this may limit the model's ability to identify specific types of attacks that exploit very subtle state interactions, precise timing dependencies, or system-level behaviors that fall outside the scope of these deliberate abstractions.

Therefore, while ProVerif is invaluable for analyzing the cryptographic integrity and authenticity of specific protocol exchanges within Uptane, a complete, top-to-bottom formal verification of the entire standard using ProVerif alone is highly challenging and would likely require significant, carefully justified abstractions.

The subsequent sections of this report will detail the methodology adopted to model key aspects of the Uptane standard within ProVerif. This involves a focused approach, targeting specific critical security properties and interactions, and employing necessary abstractions to make the analysis tractable while still providing meaningful security assurances for the chosen scope.

5.1 Declaration of Operations

To analyze Uptane's security with ProVerif, the presented model begins by defining the basic elements needed. These include important data types, the cryptographic functions used, the communication channels for messages, and other key operations of the protocol.

5.1.1 Communication Channels

At the heart of any distributed protocol like Uptane are the pathways for message exchange. In our ProVerif model, we define several public communication channels. It's crucial to understand that "public" here implies they operate under the Dolev-Yao threat model: an adversary has full control to observe, intercept, modify, or inject any messages traversing these links. This pessimistic assumption allows for a robust security analysis. The channels defined are:

```
free cDirector: channel.      (* Primary ECU <-> Director Repository *)
free cVehicle: channel.       (* Primary ECU <-> Secondary ECU *)
free cImageRepo: channel.     (* Primary ECU <-> Image Repository *)
free cTime: channel.          (* Primary ECU <-> Time Server *)
```

These channels directly map to specific Uptane interactions:

- The `cDirector` channel models the vital communication link where a vehicle's Primary ECU reports its status (vehicle version manifest) to the Director Repository and, in turn, receives tailored update instructions.
- For in-vehicle communication, `cVehicle` represents the network segment, as described by Uptane, between the Primary ECU and various Secondary ECUs. This is the pathway for Secondary ECUs to send their local manifests to the Primary, and for the Primary to distribute validated software images and their accompanying metadata to the Secondaries.
- Interaction with the Image Repository, the source of actual software images and their protective TUF metadata (Root, Timestamp, Snapshot, Targets), is modeled via the `cImageRepo` channel.
- Finally, recognizing Uptane's critical requirement for fresh and trustworthy time, the `cTime` channel facilitates communication between a Primary ECU and a designated Time Server.

5.1.2 Cryptographic Primitives

The security of Uptane is deeply rooted in its use of cryptographic primitives. The presented ProVerif model symbolically represents these primitives, operating under the standard assumption of perfect cryptography.

Signatures: Digital signatures are the cornerstone of Uptane's authenticity and integrity model, protecting all crucial metadata. To reflect this, our ProVerif model incorporates a comprehensive set of functions for key generation, the act of signing, and the verification process. These directly correspond to how Uptane roles and ECUs manage and use asymmetric cryptographic keys.

```
fun ok(): result. (* Represents a successful signature verification *)

fun sign(bitstring, skey): bitstring. (* Models a signature *)

reduc forall m:bitstring, y:keymat;
      getmess(sign(m,sk(y))) = m.

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

Hash Function: Uptane relies extensively on cryptographic hashes to ensure the integrity of software images and to create a chain of trust within its metadata structure. This is captured in our model by a symbolic hash function:

```
fun h(bitstring): bitstring.
```

Public Key Identifier Function: Managing the array of public keys associated with various ECUs or different roles (Root, Targets, Snapshot, Timestamp in both Director and Image Repositories) and ensuring that the correct key is used for verification is a critical aspect of the framework. Our ProVerif model addresses this through a mechanism that derives a unique key identifier (`kID`) from a public key (`pkey`):

```
fun fkey(pkey): kID [private].
reduc forall x:pkey; getkey(fkey(x)) = x.
```

The utility of these functions in the ProVerif model of Uptane is best understood through their application:

- The `fkey` function allows us to take a role's public key, say `x` (which could be the public key of the Director's Targets role), and generate a distinct identifier `kID` for it. The `[private]` designation is significant: it restricts the Dolev-Yao attacker from arbitrarily forging new, valid identifiers for any public keys they might intercept. This reflects the controlled manner in which key identities are established and managed within the Uptane framework, primarily through the securely distributed Root metadata.
- Conversely, the `getkey` reduction rule, `forall x:pkey; getkey(fkey(x)) = x.`, models the process by which a party, upon encountering a key identifier (perhaps embedded in a signed message or listed in the Root metadata's key delegations), can resolve this identifier back to the actual public key `x`. This retrieved key is then used for signature verification.

These key identifier mechanisms are thus indispensable in the ProVerif model of Uptane. They enable ECUs, as well as the Director and Image Repositories when verifying metadata from other sources, to unambiguously determine and apply the correct public keys. This ensures that the authenticity of the various metadata files they exchange can be rigorously checked. Such a mechanism is especially pertinent in reflecting the dynamic nature of Uptane, where key rotation and the use of multiple signing keys for a single role are standard practices to maintain long-term security and operational resilience, ensuring the chain of trust remains intact.

In developing our ProVerif model, however, we have introduced a specific abstraction concerning Uptane's comprehensive key management features. For analytical tractability and to focus on core signature validation pathways, each Uptane role within the model is represented by a single static cryptographic key pair. This simplification is reasoned within the context of ProVerif's symbolic analysis: the Dolev-Yao attacker operates under the assumption of perfect cryptography and does not gain knowledge of private keys unless they are explicitly compromised or leaked through a protocol flaw. Therefore, representing each role with a single, uncompromised key allows us to effectively verify whether an entity possesses the legitimate authority (i.e., the correct secret key) to issue signed metadata. While this abstraction does not capture the full operational complexity of M-of-N threshold policies or the specific temporal aspects of key revocation and replacement in a rotation schedule, it sufficiently models the fundamental requirement that ECUs must verify signatures using keys that are vouched for by a trusted authority.

A direct consequence of this modeling choice is the streamlined, yet critical, function of the Root role within our ProVerif representation of Uptane. In this abstracted model, the Root role is principally responsible for securely providing the ECU with the authoritative public key identifiers for each of the other top-level Uptane roles. This aligns with the Root's position as the ultimate source of trust in the Uptane architecture, ensuring that ECUs have a secure basis for validating all subsequent metadata from other roles, even within our simplified key management scheme.

5.2 Main Process and Model Setup

The main process in a ProVerif script serves as the orchestrator, meticulously setting the stage for the protocol's simulated execution. It begins by generating the cryptographic key pairs that are fundamental to the security of all participating Uptane entities. Following this, it establishes a baseline of shared information, representing the system's known state immediately prior to the specific update cycle being modeled. Finally, and most importantly, it instantiates the concurrent processes that embody the distinct roles and behaviors of the Uptane standard, allowing their interactions to be formally analyzed.

5.2.1 Initial Key Generation

The security guarantees of Uptane are deeply anchored in the correct use of cryptographic keys. Consequently, the main process initiates its execution by generating fresh key material (`keymat`) for every entity involved in the update protocol. From this raw material, corresponding public (`pk`) and private (`sk`) key pairs are derived.

In this setup, distinct key pairs are generated for:

- The **Primary ECU** (`kp1`) and the **Secondary ECU** (`kp2`).
- The various TUF-defined roles within both the **Director Repository** (Root `kRootD`, Timestamp `ktimestampD`, Snapshot `ksnapshotD`, and Targets `ktargetD`) and the **Image Repository** (Root `kRootI`, Timestamp `ktimestampI`, Snapshot `ksnapshotI`, and Targets `ktargetI`). This reflects the previously discussed abstraction where each role's signing capability is represented by a single key pair.

- The **Time Server** (`kTimeServer`), which uses its key to sign time attestations.

5.2.2 Establishing Shared Context and Initial State

Beyond cryptographic keys, the Uptane protocol operates on a body of shared knowledge. This includes vehicle-specific identifiers, details about ECU hardware, and the status of software versions installed prior to the current update. The main process models this by declaring fresh bitstrings for such data, effectively representing information that is known to the relevant participants at the commencement of the simulated update.

For instance: This initial state data encompasses ECU-specific details like identifiers (e.g., `id1`, `id2`), vehicle information (e.g., `VIN`), current software states including release counters (e.g., `ReleasCounter1`, `info1`), versions of previously encountered metadata from both repositories (e.g., `targetVD`, `rootVI`), and an initial `currentTime` value.

This collection of pre-existing data is then conveniently bundled into a `group` tuple, facilitating its propagation as a cohesive unit of shared knowledge to the various Uptane processes.

5.2.3 Instantiation of Concurrent Uptane Entities

With the cryptographic groundwork laid and the initial state of the system defined, the main process culminates in the instantiation of the active Uptane entities. Each entity is represented as a distinct ProVerif process. The crucial `!` prefix before each process call signifies that an unbounded number of instances of that process can execute in parallel. This effectively models the potential for multiple, simultaneous sessions or interactions within the Uptane ecosystem.

The instantiated processes, each initialized with its relevant keys, identifiers, and shared state information, are:

- `p1`: The Primary ECU process.
- `p2`: The Secondary ECU process.
- `pD`: The Director Repository process.
- `pI`: The Image Repository process.
- `pT`: The Time Server process.

5.2.4 Model Abstractions and Scope in Setup

It is important to acknowledge certain abstractions embedded within this main process definition, which are deliberately introduced to maintain a manageable model complexity suitable for formal analysis with ProVerif:

- **Simplified Vehicle Topology:** Our Uptane model features a streamlined vehicle configuration, consisting of a single Primary ECU that communicates with only one Secondary ECU. This necessary simplification means that more complex in-vehicle network scenarios, such as a Primary ECU broadcasting updates to, or collecting manifests from, multiple Secondary ECUs concurrently, are outside the direct scope of this particular model.
- **Single Update Cycle Focus:** The model is designed to simulate the mechanics of a single generic update event. This implies that it does not model the very first ("bootstrap") update a vehicle might receive, as evidenced by its reliance on pre-existing `ReleaseCounter` values and past metadata versions. Nor does it attempt to model an indefinite sequence of updates over an extended period.
- **Omission of Delegation Mechanism:** The Uptane (and TUF) feature of *delegation* for Targets metadata has not been incorporated into the model. This means that scenarios involving the Targets role delegating signing authority for specific software components to other entities (e.g., tier-1 suppliers signing for their respective firmware, or complex chains of delegations) are not represented. The model implicitly assumes that the top-level Targets roles on the Image and Director repositories are directly responsible for signing all relevant metadata for the software images being updated.

These abstractions were consciously adopted to enhance the simplicity and tractability of the ProVerif model. While they allow for a rigorous verification of key security properties within the defined boundaries, they also naturally shape the spectrum of threat scenarios that can be effectively investigated and the nature of the security queries that can be posed to the ProVerif tool.

5.3 High-Level Description of Modeled Uptane Processes

The core logic of the Uptane protocol is captured in our ProVerif model through several distinct, concurrently executing processes. Each process embodies the behavior of a key entity within the Uptane architecture, performing its designated roles in the software update cycle.

Modeling Time and Expiration

In our ProVerif model, **time is abstracted as a `bitstring`**. The progression of time is represented through sequential applications of a hash function. For instance, if the current moment is represented by a bitstring `currentTime`, the next discrete time step is modeled as `h(currentTime)`, the one after that as `h(h(currentTime))`, and so on.

This abstraction directly influences how metadata expiration is checked. When a process verifies if a piece of metadata is still valid, it compares the metadata's expiration timestamp against this evolving notion of time. In our model, a typical check is structured as:

```
if h(time) = expirationM || h(h(time)) = expirationM then (* ...metadata is valid... *)
```

This logic implies that metadata is considered valid if its expiration timestamp matches either the next immediate time step (`h(time)`) or the one following (`h(h(time))`). Consequently, the maximum validity window for any metadata, once issued, is effectively constrained to two such symbolic "time units" beyond the current time when the check is performed. If the expiration timestamp does not fall within this narrow window, the metadata is treated as expired.

Modeling Versions and Release Counters

A similar hashing-based abstraction is applied to **versions of metadata and release counters of software images**. These are also declared as `bitstring`s. When a new version or release counter is generated, or when an entity checks for an update, the model represents the "next" version or counter as the hash of the current one. For example, if `targetVD` is the bitstring representing the current version of a Director's Targets metadata, its immediate successor version is represented as `h(targetVD)`.

While this abstraction streamlines version management by focusing on direct, single-step progression, it limits the model's ability to validate more complex scenarios. Specifically, it can confirm that a new version is the cryptographic hash of its immediate predecessor, but it lacks inherent mechanisms to detect arbitrary version jumps or enforce sequencing rules across multiple steps.

Rationale and Implications of Abstractions

These two abstractions, for time and versions / counters were adopted primarily to maintain a lightweight ProVerif model, enhancing the tractability of formal analysis. By simplifying these potentially state-heavy aspects, we can focus on the core cryptographic exchanges and logical flows of Uptane.

However, it is crucial to acknowledge that these simplifications introduce certain limitations. The constrained time window and the single-step version increment mechanism mean that the model may not capture all the nuances of long-term temporal attacks or sophisticated version manipulation scenarios. These abstractions will therefore have implications for the scope of security properties that can be queried and the range of threat models that can be exhaustively explored in the subsequent analysis sections.

We now proceed to a high-level overview of each of the main Uptane processes: `pT` (Time Server), `p2` (Secondary ECU), `p1` (Primary ECU), `pD` (Director Repository), and `pI` (Image Repository) within this modeled environment.

5.3.1 The `pT` Process (Time Server)

The Time Server process, `pT`, is responsible for providing a secure and fresh notion of time.

- It begins by awaiting a time request on the `cTime` channel. This request, originating from the Primary ECU, is a bundled and signed message containing distinct tokens, one from the Primary itself and one relayed from the Secondary ECU, each individually signed by their respective ECUs and then collectively signed by the Primary before transmission to the Time Server.
- Upon receiving this request, `pT` meticulously verifies the layered signatures: first the overall signature from the Primary, then the individual signatures on each enclosed token. This ensures the authenticity and authorization of the request.
- If verification is successful, the Time Server generates a new, current timestamp (modeled as an update to its notion of `currentTime`).
- Finally, it constructs a response containing this new time. Specifically, it creates two separate messages, each pairing the new time with one of the original tokens, and signs each of these pairs with its own secret key. These two signed messages are then bundled together, and this bundle itself is signed by the Time Server before being sent back to the Primary ECU over the `cTime` channel. This layered response allows the Primary to securely distribute the attested time to the relevant ECU.

5.3.2 The p2 Process (Secondary ECU)

The Secondary ECU process, p2, models a typical resource-constrained ECU within the vehicle that relies on the Primary for updates and performs partial verification.

- **Phase 1: Reporting and Initial Time Synchronization Setup**
  - It first constructs its own ECU Version Report , detailing its current software state, its own ID, and a fresh nonce. This report is signed with its private key sk(kp2) and sent to the Primary ECU via the in-vehicle channel cVehicle.
  - Concurrently, it generates a unique token for the time synchronization process, signs this token, and also sends it to the Primary ECU.
  - It then waits to receive a signed time attestation, which is forwarded by the Primary ECU. Upon receipt, p2 verifies this message using the Time Server's public key and checks that the token in the response matches its original token, ensuring the time is fresh and authentic for its context.
- **Phase 2: Update Reception and Partial Verification**
  - After successful time synchronization, p2 awaits update instructions and the actual software image from the Primary ECU. This arrives as a package containing the Director Repository's signed Targets metadata (relevant to p2) and the corresponding software image.
  - It then performs the partial verification steps as mandated by the Uptane standard. These checks, which include signature validation, freshness checks, ECU/hardware identifier matching, release counter verification, and image hash comparison against the Director's Targets metadata, are detailed in Section 4.3.3.
  - If all these checks, as outlined previously, are successful, the Secondary ECU considers the update valid and "installs" the new image.

5.3.3 The p1 Process (Primary ECU)

The Primary ECU process, p1, acts as the central orchestrator within the vehicle for the Uptane update process, performing full verification and managing communication with external repositories and internal Secondary ECUs.

- **Phase 1: Manifest Aggregation and Time Request Initiation**
  - p1 generates its own ECU Version Report, including its software state, its ID and a nonce and signing it with its private keysk(kp1).
  - It receives the signed ECU Version Report and the signed time token from the Secondary ECU (p2) via the cVehicle channel and verifies both signatures.
  - It then constructs the Vehicle Version Manifest (VVM), which includes the vehicle's VIN (Vehicle Identification Number), its own ID, its signed report, and the verified signed report from secondary ECU. This comprehensive VVM is signed and transmitted to the Director Repository over cDirector.
  - For time synchronization, p1 creates a bundle containing its own signed time token and the verified signed time token from p2. This entire bundle is then signed and sent to the Time Server (pT) via cTime.
- **Phase 2: Time Reception and Validation**
  - Primary ECU awaits the response from the Time Server. Upon receiving the bundled signed message, it verifies the correctens of it as detailed in the Section.
- **Phase 3: Metadata/Image Acquisition and Full Verification**
  - **Director Interaction:** p1 fetches the chain of TUF metadata from the Director Repository. This involves:
    1. Receiving and verifying the Director's Root metadata (using its pre-trusted Director Root public key. This step also involves extracting the public keys for the Director's Timestamp, Snapshot, and Targets roles.
    2. Sequentially receiving and verifying the Director's Timestamp, Snapshot, and Targets metadata.
  - **Image Repository Interaction:** A similar full verification sequence is performed for the Image Repository:
    1. Receiving and verifying the Image Repository's Root metadata and extracting its roles' public keys.
    2. Sequentially receiving and verifying the Image Repository's Timestamp, Snapshot, and relevant Targets metadata, applying all standard TUF checks.

- **Image Download and Cross-Verification:** Based on the verified metadata, `p1` then downloads the actual software images (one for itself, `Image1`, and one for the Secondary, `Image2`) from the Image Repository. Crucially, it performs a cross-repository check: it ensures that the image details (e.g., hashes, release counters for `id1` and `id2`) specified in the Director's Targets metadata are consistent with those found in the Image Repository's Targets metadata. It also verifies that the hashes of the downloaded images match the hashes in the trusted Image Repository Targets metadata and that in the Target metadata of the Director.

- **Phase 4: Update Installation and Forwarding to Secondary**
  - If all comprehensive verification steps pass successfully, `p1` proceeds to "install" its own new firmware (`Image1`).
  - It then forwards the relevant portion of the validated time response from the Time Server (the signed (time, token) pair for `p2`) to the Secondary ECU.
  - Finally, it sends the validated Director's Targets metadata (specifically the part relevant to `p2`) and the corresponding validated software image (`Image2`) to the Secondary ECU, to perform its update.

5.3.4 The `pD` Process (Director Repository)

The Director Repository process, `pD`, is responsible for authorizing updates by determining which software versions should be installed on which ECUs, based on the vehicle's reported state.

- **Phase 1: VVM Reception and Validation**
  - `pD` receives a Vehicle Version Manifest (VVM) from a Primary ECU (`p1`) over the `cDirector` channel.
  - It performs a thorough validation of the VVM: it checks the overall signature from `p1`, and then unpacks and verifies the signatures on the individual ECU Version Reports (from `p1` and `p2`) contained within. It also confirms that the reported current software state (e.g., `infver1`, `infver2`) and identifiers (`id1`, `id2`, `VIN`) in the VVM align with its expectations or database records (abstracted by comparing with the initial `group` data).

- **Phase 2: Metadata Generation and Dispatch**
  - Based on the validated VVM, `pD` determines the appropriate updates. In the model, this involves generating new metadata for new, distinct software images (`image1` for `p1` and `image2` for `p2` are represented by private free variables. They are declared in this way because the attacker cannot know the new update until it is explicitly sent over the network and are common to both Image and Director repositories).
  - It then constructs a complete set of fresh TUF metadata for this update directive.
  - Each of these metadata files is signed using the secret key of its respective role (`sk(kRootD)`, `sk(ktimestampD)`, etc.).
  - Finally, `pD` transmits this full bundle of signed Director metadata (Root, Timestamp, Snapshot, and Targets) to the Primary ECU via the `cDirector` channel.

5.3.5 The `pI` Process (Image Repository)

The Image Repository process, `pI`, serves as the source for TUF-protected software images and their descriptive metadata. In this model, it acts somewhat autonomously to prepare and offer these artifacts.

- **Metadata and Image Publication**:
  - The `pI` process models the creation and signing of a new set of TUF metadata for the software images it hosts.
  - All generated metadata files are signed with the Image Repository's respective role keys (`sk(kRootI)`, `sk(ktimestampI)`, etc.).
  - `pI` then makes these signed metadata files, along with the actual software images (`image1`, `image2`), available for download by outputting them onto the `cImageRepo` channel, from where the Primary ECU can fetch them.

These process descriptions, while high-level, outline the modeled behavior of each Uptane entity, forming the basis for the subsequent security analysis using ProVerif's verification capabilities.

6 Uptane Threat Model

Ensuring the security of software updates in modern vehicles is paramount, as the consequences of a compromised update mechanism can be severe. To design robust defenses, it's essential to first understand the motivations that drive attackers, the capabilities they might possess, and the specific methods they could employ to undermine the update process. This section outlines these critical aspects of the threat landscape.

6.1 Attacker Motivations and Goals

Attackers targeting automotive software update systems may have a variety of objectives, which can escalate in terms of potential impact:

- **Intelligence Gathering and Information Theft:** A primary goal could be to illicitly access the contents of software updates. This might be to discover confidential information or intellectual property embedded in the firmware, to reverse-engineer the software for competitive analysis, or to compare different firmware versions to identify recently patched security vulnerabilities, thereby pinpointing exploitable weaknesses in unpatched vehicles.
- **Denial of Service and Disruption of Updates:** Attackers may seek to prevent vehicles from receiving or successfully installing legitimate software updates. The aim here is to hinder the remediation of known software issues or security vulnerabilities, potentially leaving vehicles in an unsafe or degraded state.
- **Interference with Vehicle Functionality:** A more malicious intent involves attempts to cause one or more Electronic Control Units (ECUs) within the vehicle to fail or behave in unintended ways. Such interference could deny the use of the vehicle entirely or disable specific functions.
- **Achieving Control over ECUs or the Entire Vehicle:** The most critical threat involves an attacker gaining unauthorized control over the operation of one or more ECUs, or even the vehicle as a whole. This could allow manipulation of vehicle performance or behavior, posing significant safety risks.

6.2 Assumed Attacker Capabilities

To achieve the goals mentioned above, attackers may leverage a range of capabilities, reflecting different levels of access and sophistication:

- **Network Traffic Manipulation:** A common capability is the ability to intercept, monitor, modify, replay, or inject network traffic. This is often referred to as a man-in-the-middle (MitM) position and can be established in several ways:
  - *External to the vehicle:* For instance, by controlling a cellular network, Wi-Fi access point, or any part of the internet infrastructure used to distribute updates from backend repositories to the vehicle.
  - *Internal to the vehicle:* By gaining access to the vehicle's internal communication buses (such as CAN, LIN, or Ethernet).
- **Compromise of Server-Side Infrastructure:** Attackers might compromise the backend servers that host software updates and metadata. This could include either the Director repository (which typically manages update policies and ECU-specific instructions) or the Image repository (which stores the actual software images), along with any online signing keys accessible on those compromised servers.
- **Compromise of In-Vehicle ECUs:** An attacker might manage to compromise specific ECUs within the vehicle.
- **Access to Historical Update Information:** Attackers are often assumed to have the ability to obtain copies of previously released software updates and metadata. They could achieve this by legitimately acting as a vehicle requesting updates over time or by finding publicly accessible archives. This historical data can be invaluable for planning certain attacks, like freeze or rollback attacks.

6.3 Common Attack Vectors and Model Coverage

Given the attacker goals and capabilities outlined previously several distinct attack vectors can be identified against automotive software update systems. This subsection discusses these strategies, clarifying how each has been addressed or considered within the scope and abstractions of our ProVerif model. Our primary verification effort, detailed below, focuses on preventing arbitrary software installation, a critical security property that also encompasses the ultimate goals of several other attack types. Other attacks are discussed in terms of the ProVerif model's inherent capabilities or specific abstractions employed.

6.3.1 Verification Against Arbitrary Software Installation and Its Variants

The most severe threat involves an attacker succeeding in making an ECU install and execute unauthorized software. Our ProVerif model primarily targets this through correspondence queries. We also consider Rollback and certain Mix-and-Match attacks as variants whose end goal can be abstracted as a form of arbitrary or attacker-influenced software installation.

- **Arbitrary Software Attack:** This is the most severe form of attack, where the adversary succeeds in making an ECU install and execute malicious software of the attacker's own design. This implies a complete bypass of the update mechanism's security controls, granting the attacker potential control over the ECU's functions and, depending on the ECU, potentially broader control over vehicle operations.

  Preventing the installation of arbitrary software is a cornerstone of a secure update system. In our ProVerif model, this is primarily verified through **correspondence properties** that assert that if an ECU installs a piece of software, that software must have been legitimately sourced and authorized by the backend repositories. We use specific queries to check this for both the primary (P1) and secondary (P2) ECUs:

  1. **Verification for the Primary ECU (P1):** The following query ensures that any software installed by P1 originates from and is consistent with what the Director and Image Repositories have published:

     ```
     query rc:bitstring, h:bitstring, img:bitstring;
       event(ecu_installed(rc,h,img)) ==>
         ( event(image_publish(rc,h,img)) &&
           event(director_publish(rc,h)) ).
     ```

     - `event(ecu_installed(rc,h,img))`: Signals that the primary ECU (P1) has completed the installation of an image `img` with a specific release counter `rc` and hash `h`.
     - `event(image_publish(rc,h,img))`: Signals that the Image Repository (`pI` process) has legitimately published this exact image `img`, associated with the same release counter `rc` and hash `h`.
     - `event(director_publish(rc,h))`: Signals that the Director repository (`pD` process) has legitimately published metadata authorizing an update for the primary ECU with the given release counter `rc` and image hash `h`.

     If ProVerif proves this query true, it means that P1 will only install software that is an exact match (in terms of content, release counter, and hash) to what was officially provided by the Image Repository and simultaneously authorized by the Director for P1. This directly prevents an attacker from injecting a foreign or unauthorized software image onto an ECU, as any such image would not trigger the corresponding `image_publish` and `director_publish` events with matching parameters.

  2. **Verification for the Secondary ECU (P2):** Similarly, for the secondary ECU (P2), we verify that its installations are authorized by the Director:

     ```
     query rc:bitstring, h:bitstring, img:bitstring;
       event(ecu_installed_2(rc,h,img)) ==>
         ( event(director_publish_2(rc,h)) ).
     ```

     - `event(ecu_installed_2(rc,h,img))`: Signals that the secondary ECU (P2) has completed the installation of an image `img` with release counter `rc` and hash `h`.
     - `event(director_publish_2(rc,h))`: Signals that the Director repository (`pD` process) has legitimately published metadata authorizing an update specifically for the secondary ECU with the given release counter `rc` and image hash `h`.

     If this query is proven true, it ensures that P2 only installs software whose characteristics (release counter `rc` and hash `h`) have been explicitly authorized by the Director for P2. While this query doesn't directly show an `image_publish` event from P2's perspective (as P2 receives its image via P1), the overall security relies on:
     - The Director (`pD`) including the correct hash (e.g., `h(image2)`) in the `target2` metadata it signs and which is conveyed to P2 (triggering the `director_publish_2` event with this hash).
     - The primary ECU (`p1`) performing a full verification of metadata from both Director and Image Repository, ensuring consistency for all images, including the one destined for P2.
     - The secondary ECU (`p2`) then verifying the hash of the image it actually receives from P1 against the hash (`hImageD`) provided in the Director's metadata fragment (`targetD`) it receives.

  By proving these queries, the model demonstrates that neither the primary nor the secondary ECU can be tricked into installing software that hasn't been properly authorized and (for P1 directly, and for P2 via P1's full verification and P2's partial verification) sourced from the legitimate backend infrastructure. This provides a strong guarantee against an attacker successfully executing an arbitrary software attack.

- **Rollback Attack (as a form of Arbitrary Software Installation):** This attack tricks an ECU into installing a software version older than its current one. While our model's versioning checks (expecting `h(current_version)`) are designed to prevent simple replays of outdated metadata, a sophisticated rollback (e.g., forging new metadata with incremented versions but pointing to an old image's hash) would typically require compromised signing keys. The single update cycle modeled limits specific queries for such advanced rollbacks. However, since a successful Rollback Attack results in the ECU installing an attacker-chosen (albeit older) software version, its ultimate outcome is effectively an instance of an Arbitrary Software Attack. The security against this broader category is therefore addressed by the Arbitrary Software Attack queries discussed above.
- **Mix-and-Match Attack:** This involves an attacker with compromised repository keys constructing a new, seemingly valid update bundle with a malicious combination of software images. The ultimate goal of compelling an ECU to install and execute a harmful, attacker-defined software configuration, created through key compromise and forgery of new bundles, can be abstracted to fall under the broader category of an Arbitrary Software Attack. The defenses against, and verification of, such arbitrary software installations have been addressed with the specific queries in our model.

6.3.2 Other Attack Vectors and Modeling Considerations

Several other attack vectors described in the Uptane threat model were considered. Their explicit verification with dedicated queries is influenced either by ProVerif's general analytic scope or by specific abstractions made in our model's design.

Attacks Generally Outside ProVerif's Direct Modeling Scope: ProVerif excels at analyzing cryptographic protocol logic but is not primarily designed for properties related to system performance, complex application-level atomicity, or resource exhaustion. Consequently, detailed modeling and verification for the following attacks were not undertaken:

- **Slow Retrieval Attack:** Degrading communication speed to hinder updates.
- **Denial-of-Service (DoS) on Repositories:** Overwhelming server infrastructure.
- **Endless Data Attack:** Exhausting ECU storage resources.

Attacks Considered in Light of Specific Model Abstractions: The full verification of certain attacks is constrained by the deliberate abstractions made in our ProVerif model to ensure tractability:

- **Eavesdrop Attack:** Within the Uptane standard, encryption of update images is merely an *optional* add-on; the protocol's primary security objectives are authenticity and integrity rather than confidentiality. Accordingly, our *ProVerif* model leaves image encryption disabled, placing secrecy of the images outside the verification scope. To assess the resulting exposure, we issued the following queries against a Dolev–Yao adversary:

  ```
  query attacker(image1).
  query attacker(image2).
  ```

  Both queries succeed, and the complementary checks `not attacker(image1[])` and `not attacker(image2[])` evaluate to **false**. *ProVerif* therefore confirms that, when the optional encryption feature is omitted, an attacker can indeed recover the transmitted update images.
- **Freeze Attack:** This involves replaying old but validly signed metadata. As detailed in our model's description, the ProVerif model's repositories, due to static initialization in replicated processes, only offer a single "next" version. This prevents modeling the core condition of a Freeze Attack where genuinely newer updates are available from repositories but are being actively hidden from the ECU by the attacker.
- **Mixed-Bundles Attack:** This involves different ECUs installing software from different, incompatible (though individually valid) update bundles. Our model's simplified topology (one Primary, one Secondary ECU) and its focus on a single, coherent update campaign limit the ability to model an attacker presenting multiple distinct, simultaneously valid bundles to different ECUs to create conflicts.

7 Further ProVerif Analyses

This section details additional formal verifications with ProVerif, including queries that explore broader protocol correctness beyond the previously defined threat model, and an analysis of the system's behavior under simulated key compromise.

7.1 Exploring Event Reachability

A central goal in formal protocol verification is to determine whether key states or events can be reached during execution. In ProVerif, this is handled via reachability queries, which confirm whether certain protocol steps can successfully complete. For instance, verifying that a component's process concludes correctly can be expressed as:

```
query event(endX()).
```

Here, `endX()` denotes the successful termination of participant `X`'s process (e.g., `endP1()` for the Primary ECU, or `endD()` for the Director Repository).

A positive verification is indicated by:

```
Query not event(endX) is false.
```

This result shows that ProVerif finds it impossible for the protocol to complete without triggering `endX()`, thereby confirming both reachability and necessity of the event.

For all core Uptane components: `pT` (Time Server), `p2` (Secondary ECU), `p1` (Primary ECU), `pD` (Director), and `pI` (Image Repository) such queries were successfully verified. In each case, ProVerif confirmed that the corresponding `endX()` event is always reachable, contingent on the proper execution of all internal checks and interactions.

7.2 Analysis of Key Compromise Scenarios

This subsection further evaluates the robustness of our UPTANE model by examining its behavior under critical key compromise scenarios. The Dolev-Yao attacker is assumed to gain illicit access to signing keys, allowing us to test the protocol's resilience. We will investigate two specific conditions: first, the implications of the attacker compromising the signing keys associated with the Director Repository, and subsequently, a more severe scenario where the attacker obtains the signing keys for both the Director and the Image Repositories.

7.2.1 Compromise of Director Repository Keys

In this scenario, we explore the security implications if an attacker gains access to the signing keys of the Director Repository, while the Image Repository's keys remain secure. The Dolev-Yao attacker in ProVerif is modeled to possess these Director-specific keys. We then re-evaluate the correspondence properties related to arbitrary software installation.

Verification for Primary ECU (P1) The query for the Primary ECU remains true even under this compromise:

```
Query event(ecu_installed(rc,h_1,img)) ==>
    (event(image_publish(rc,h_1,img)) &&
     event(director_publish(rc,h_1))) is true.
```

Implication: The Primary ECU (P1) remains secure against installing arbitrary software. This is because P1 performs a full verification, which includes cross-referencing metadata from the Director Repository with metadata from the (still secure) Image Repository. Even if an attacker uses compromised Director keys to forge malicious Director metadata (e.g., authorizing an `img` with hash `h_1` and release counter `rc`), P1 will detect an inconsistency if this forged metadata does not align with the legitimate metadata provided by the uncompromised Image Repository for that same `img`, `rc`, and `h_1`. Thus, P1 would not install software that isn't also vouched for by the Image Repository.

Verification for Secondary ECU (P2) However, the corresponding query for the Secondary ECU (P2) evaluates to false:

```
Query event(ecu_installed_2(rc,h_1,img)) ==>
    event(director_publish_2(rc,h_1)) is false.
```

Implication: This result indicates a vulnerability for the Secondary ECU under these conditions, which is consistent with the security guarantees of Uptane when a Secondary ECU performs only partial verification. The key insights are:

- **Partial Verification Vulnerability:** P2, in its minimal partial verification mode, primarily validates the authenticity and integrity of metadata from the Director Repository. If the Director's keys are compromised, an attacker can forge legitimate-looking Director metadata (e.g., new Targets metadata) that authorizes a malicious image.

- **Successful Attack Path:** The ProVerif attack trace demonstrates that if the attacker can then deliver this forged Director metadata along with the corresponding malicious software image directly to the Secondary ECU (e.g., by having a presence on the in-vehicle network and bypassing the Primary ECU), P2 will install it.
- **Role of the Primary ECU as a Defense:** If the update path strictly goes through the Primary ECU, P1's full verification (including the cross-check against the uncompromised Image Repository) would likely detect the discrepancy (as explained for P1 above). In such a case, P1 would not forward the malicious update package to P2, thus protecting it. The vulnerability for P2 materializes when this protective path via P1 is circumvented by an attacker with access to the internal vehicle network.

This scenario underscores the importance of the Primary ECU's full verification role and the potential risks for Secondary ECUs performing only partial verification if the Director Repository is compromised and the attacker has a direct channel to the Secondary ECU.

7.2.2 Compromise of Both Director and Image Repository Keys

This scenario examines a more severe threat where the Dolev-Yao attacker is assumed to have compromised the signing keys for *both* the Director Repository and the Image Repository. Under this condition, we assess the impact on the security guarantees for both the Primary and Secondary ECUs.

Verification for Primary ECU (P1) With both sets of repository keys compromised, the correspondence property for the Primary ECU now evaluates to false:

```
Query event(ecu_installed(rc,h_1,img)) ==>
    (event(image_publish(rc,h_1,img)) &&
     event(director_publish(rc,h_1))) is false.
```

Implication: This result signifies that the Primary ECU (P1) is now vulnerable to installing arbitrary software. The reasoning is as follows:

- **Complete Forgery Capability:** With control over both Director and Image Repository signing keys, the attacker can forge a complete and internally consistent set of TUF metadata for both repositories.
- **Bypassing Cross-Verification:** The attacker can create malicious Director Targets metadata authorizing an attacker-chosen image (`img`) with hash `h_1` and release counter `rc`. Simultaneously, he can forge Image Repository Targets metadata that also legitimately describes the same malicious `img` with the identical `h_1` and `rc`. All of these metadata will be validly signed using the compromised keys, and thus will pass full verification.

Verification for Secondary ECU (P2) Similarly, the query for the Secondary ECU also evaluates to false:

```
Query event(ecu_installed_2(rc,h_1,img)) ==>
    event(director_publish_2(rc,h_1)) is false.
```

Note: The compromise of both Director and Image Repository keys described in this section was modeled in ProVerif by granting the attacker access to the **Targets, Snapshot, and Timestamp signing keys**.

It is important to acknowledge that a more catastrophic scenario involves the compromise of the **Root keys**. Possession of Root keys would enable an attacker to authorize their own malicious keys for all other roles, thereby gaining full control over the signing and validation of all metadata. Modeling this Root key compromise in our ProVerif setup, however, posed a significant challenge: it necessitated making the constructor for key identifiers public, allowing the attacker to effectively introduce and validate arbitrary new keys. An attempt to simulate this Root key compromise led to the ProVerif analysis not reaching termination even after five days, at which point the simulation was stopped. Consequently, while the impact of compromising online repository keys (Targets, Snapshot, Timestamp) was successfully analyzed, the full implications of a Root key compromise within this specific ProVerif model remain computationally prohibitive to explore fully. It is anticipated, however, that had this simulation terminated, it would have confirmed a failure of the arbitrary software installation correspondence properties, similar to or exceeding the severity of the outcomes observed in the scenario where all other repository keys were compromised.

7.3 Identified Vulnerability in Uptane v2.1.0: Insufficient Integrity Binding in Snapshot Metadata

While our ProVerif analysis focused on key compromise scenarios, a closer examination of the Uptane Standard v2.1.0 reveals a subtle but significant vulnerability. Specifically, the way Snapshot metadata references Targets

metadata and the corresponding checks ECUs are required to perform may allow an attacker to bypass integrity verification under certain conditions. This subsection outlines the relevant specifications and the security implications of this design.

The Uptane Standard v2.1.0 defines metadata content requirements as follows:

Section 5.2.4. Snapshot metadata:

> The Snapshot metadata lists version numbers and filenames of all Targets metadata files. It protects against mix-and-match attacks if a delegated supplier key is compromised. For each Targets metadata file on the repository, the Snapshot metadata SHALL contain:
>
> - The filename and version number of the Targets metadata file.

Section 5.2.5. Timestamp metadata:

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

A key observation is that Snapshot metadata is only required to include the *version number* and filename of each Targets metadata file. It does not include, nor suggest including, cryptographic hashes of those files. By contrast, the Timestamp metadata does include a hash of the Snapshot metadata, ensuring its integrity.

This design affects ECU-side verification as defined in later sections of the standard:

Section 5.4.4.5. How to check Snapshot metadata:

> 2. The hashes and version number of the new Snapshot metadata file SHALL match those in the Timestamp metadata. If they do not match, the Snapshot metadata must be discarded and the update aborted.

Section 5.4.4.6. How to check Targets metadata:

> 2. The version number of the new Targets metadata file SHALL match the version listed in the latest Snapshot metadata. If not, it must be discarded and the update cycle aborted.

Thus, while the Snapshot metadata is integrity-protected via its hash in the Timestamp metadata, the Targets metadata is only checked for version alignment with the Snapshot. Because Snapshot metadata does not contain hashes of the Targets metadata files, ECUs lack a mechanism to verify their content integrity beyond signatures.

Implication: Potential Vulnerability Path

This verification model enables the following attack scenario:

- An attacker with access to a compromised Targets signing key (e.g., for the Director or Image Repository) can forge a malicious Targets metadata file.
- As long as the forged metadata retains the same version number as that listed in the legitimate Snapshot metadata, an ECU will accept it.
- Since the Snapshot metadata does not include a hash for the Targets file, the ECU cannot detect that the file has been modified.
- The forged metadata passes all standard checks and may lead to installation of unauthorized software.

This demonstrates that the integrity of Targets metadata relies solely on its signature, not on cross-verification through the Snapshot metadata. Although the Snapshot protects against inconsistent versions (i.e., mix-and-match attacks), it does not guard against tampering with the content of a Targets file whose version number remains unchanged.

To validate this finding, we modeled a scenario in ProVerif where only the Targets signing keys for the Director (`sk(ktargetD)`) and Image (`sk(ktargetI)`) repositories were compromised. All other keys including Root, Timestamp, and Snapshot remained secure. The following correspondence queries evaluated to false:

```
Query event(ecu_installed(rc,h_1,img)) ==>
    (event(image_publish(rc,h_1,img)) &&
     event(director_publish(rc,h_1))) is false.

Query event(ecu_installed_2(rc,h_1,img)) ==>
    event(director_publish_2(rc,h_1)) is false.
```

These results confirm that, even without compromising Snapshot or Timestamp keys, an attacker can trick both Primary and Secondary ECUs into installing unauthorized software. This is possible because forged Targets metadata, signed with compromised keys and bearing the correct version number, will pass ECU checks. This outcome mirrors the impact seen in Section 7.2.2 ("Compromise of Both Director and Image Repository Keys"), but is achieved with a far less powerful attacker highlighting the security implications of not including Targets metadata hashes in Snapshot metadata.

8 Conclusion

This report presented a formal verification of key components and interactions in the Uptane over-the-air (OTA) update framework using the ProVerif automated verification tool. A symbolic model was developed to represent the core entities Primary and Secondary ECUs, Director and Image Repositories, and the Time Server focusing on a single update cycle and simplified key management for tractability.

ProVerif analysis confirmed several critical security properties. Under normal (non-compromised) conditions, correspondence queries showed that both Primary and Secondary ECUs only install software published and authorized by the legitimate repositories. Reachability queries verified that all processes reach their intended terminal states, assuming proper execution of internal checks and message exchanges.

Key compromise scenarios revealed important insights. When only the Director Repository's keys are compromised, the Primary ECU remains secure due to its full verification process that cross-checks metadata with the uncompromised Image Repository. In contrast, a Secondary ECU performing only partial verification becomes vulnerable if an attacker can deliver malicious updates directly. When both the Director and Image Repository keys are compromised, both ECUs are susceptible to unauthorized software installation.

A deeper review of the Uptane Standard v2.1.0 revealed a specific vulnerability in how Snapshot metadata references Targets metadata. According to Section 5.2.4, Snapshot metadata is required to list only version numbers—not cryptographic hashes—of Targets metadata. Section 5.4.4.6 further specifies that ECUs validate only the version number during Targets metadata verification. A targeted ProVerif experiment showed that, if only the Targets keys are compromised, an attacker can forge a malicious Targets metadata file with a matching version number that ECUs will accept, leading to unauthorized software installation. This demonstrates that a single compromised key, combined with minimal standard-compliant checks, can undermine system integrity.

While Uptane is intended as a flexible framework, this finding underscores the importance of implementers going beyond the minimum required checks. Incorporating cryptographic hashes of Targets metadata in Snapshot files and enforcing corresponding ECU checks—as aligned with TUF best practices—would mitigate this risk.

Finally, we note that certain classes of attacks—such as persistent Freeze, sophisticated Rollback, and some Mix-and-Match variants—fall outside the scope of our current single-cycle model due to their reliance on state history or multi-cycle interactions.

In conclusion, this study affirms Uptane's strong security posture under typical conditions and highlights the effectiveness of its full verification approach. At the same time, it identifies a previously underexamined edge case in the specification that may allow an attacker with limited capabilities to bypass core protections. These insights reinforce the value of formal verification and careful standard interpretation in securing real-world deployments.