A Comprehensive, Automated Security Analysis of the Uptane Automotive Over-the-Air Update Framework

Robert Lorch
robert-lorch@uiowa.edu
The University of Iowa
Iowa City, Iowa, USA

Daniel Larraz
daniel-larraz@uiowa.edu
The University of Iowa
Iowa City, Iowa, USA

Cesare Tinelli
cesare-tinelli@uiowa.edu
The University of Iowa
Iowa City, Iowa, USA

Omar Chowdhury
omar@cs.stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

ABSTRACT

We present our experience of formally verifying the desired security properties of the Uptane over-the-air (OTA) software update framework against a set of applicable threat models. Uptane is gaining traction in the automobile industry and is widely considered the next de-facto standard for OTA automobile software updates. The security of Uptane is of utmost importance because modern automobiles rely on software for their safety-critical functionalities and, especially, require OTA software updates to add new safety features or patch bugs in existing ones. Design flaws in Uptane can either violate the integrity of the updates to be installed or prevent vehicles from installing new updates, both of which can cause severe safety issues. Previous approaches to protocol verification either fail to capture the necessary features of Uptane or suffer from termination issues due to Uptane's complexity. A key component of our approach lies in the *eager* combination of an infinite-state model checker and a cryptographic protocol verifier, where (in contrast to prior *lazy* approaches) we are able to eliminate a key manual step in the workflow while enabling reasoning over more fine-grained message structures. In addition, our approach utilizes two proven *soundness- and completeness-preserving state-space-reduction optimizations* for computational tractability, as well as a *meta-level analysis technique* that makes it feasible to reason over Uptane's set of optional protocol features. Our approach is able to discover six new vulnerabilities while rediscovering all five known ones. While there have been previous analyses of Uptane's security properties, they either missed design flaws identified by our approach or suffered from coverage and termination issues. The Uptane standards body has positively acknowledged our findings and has suggested updates to the protocol specification documents to address them.

CCS CONCEPTS

• **Security and privacy → Logic and verification**.

RAID 2024, September 30–October 02, 2024, Padua, Italy

ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678927

KEYWORDS

automotive security, model checking, protocol analysis, attacks, vulnerabilities

ACM Reference Format:
Robert Lorch, Daniel Larraz, Cesare Tinelli, and Omar Chowdhury. 2024. A Comprehensive, Automated Security Analysis of the Uptane Automotive Over-the-Air Update Framework. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3678890.3678927

1 INTRODUCTION

Safety-critical functions in modern automotive vehicles such as airbag system activation, engine control, stability control, braking, and more are now controlled by software [49]. In fact, the average new car runs on a staggering 100 million lines of code [50]. Therefore, vehicles must regularly perform software updates to patch bugs and implement new features. Physically taking vehicles to dealers for software updates can be both ineffective and impractical. It is thus preferable for next-generation automotive vehicles to perform these updates over the air (OTA). OTA updates, however, can be vulnerable to adversaries, as demonstrated by the compromising of many major update repositories, including those run by Apache, Debian, Fedora, and GitHub [20, 29, 30, 64]. Further, many major auto manufacturers have suffered security vulnerabilities allowing attackers to tamper with safety-critical functions such as locking/unlocking, engine starting/stopping, and more [17, 19, 65].

Due to these issues, much work has been dedicated to securing OTA updates [3, 5, 11, 12, 15, 22, 31, 34, 36, 41–43, 53, 56, 57, 59]. Deviating from traditional computing devices, software in automotive vehicles runs on electronic control units (ECUs), which have limited computing power, secure remote communication capabilities, and built-in security measures. *An automotive OTA update protocol is thus required to take these limitations into account.*

Uptane [26, 41] is a proposed OTA protocol that seems set to become the industry-standard framework for secure automotive OTA updates. In fact, it is currently being integrated into Automotive Grade Linux (AGL) [25], which has a huge share of the automotive OS market and is used by manufacturers such as Toyota, Honda, Ford, Nissan, Mercedes, and Audi [25]. In addition to AGL, Uptane is supported by commercial software management platforms (*e.g.*,

[1]) with major clients such as Infiniti, Renault, Bosch, and Continental. *Because of this, a large proportion of vehicles on the road will soon be using Uptane.* Design-level protocol issues will become much more problematic after large-scale deployment, as the cost and technical difficulty of fixing protocol design bugs is disproportionately high at later stages of the development cycle. Thankfully, we are currently in a unique window of opportunity to rigorously analyze the protocol for bugs *before* it is widely deployed. *Despite the automotive industry's large reliance on Uptane, its security has not been studied comprehensively in the literature.*

Problem and scope. Uptane claims to protect *safety-critical systems* from adversaries with a wide range of capabilities. Being a framework, Uptane is designed to be compatible with a wide range of automakers and ECU capabilities, meaning that there is considerable freedom for implementers to make design decisions. Since frameworks are specified at a high level of abstraction, design errors can have sweeping consequences. Given this landscape, we answer the following question: *Can we apply formal verification to analyze the Uptane framework with respect to a set of feasible adversaries in order to increase confidence in its promised security properties and discover underlying design issues?* In our formal analysis, we consider adversarial influence both (i) within the vehicle's internal network and (ii) between the vehicle and remote communication partners. *The overall problem can be reduced to a protocol analysis in the symbolic model where some or all of the above communication channels are controlled by a Dolev-Yao [21] adversary.*

Challenges. A comprehensive, fine-grained, terminating, and automated security analysis of Uptane warrants addressing the following challenges: (1) ***long execution traces***: the Uptane protocol induces long execution traces when it is modeled exactly as described in the standard documents, resulting in extremely long execution times for the automated analysis; (2) ***properties referencing concrete payload values***: analyzing Uptane requires reasoning about fine-grained security properties that reference concrete values in message payloads, which is a problem for both cryptographic protocol verifiers and current abstraction-refinement-based techniques; (3) ***large number of problem instances to solve***: when considering Uptane's specific threat models, desired properties, and optional features (*i.e.*, 9 threat models, 8 properties, and 4 optional features), a naive protocol verification approach for *comprehensively* analyzing Uptane requires solving 1152 (= $9 \times 8 \times 2^4$) protocol verification problem instances; (4) ***manual efforts***: although manual efforts needed for formally analyzing a protocol are unavoidable, especially for formalizing the protocol design and collecting desired properties (*i.e.*, inputs to the analysis approach), any protocol analysis approach that requires manual intervention becomes prohibitive when considering 1152 problem instances.

Note that these challenges apply to formal verification in general, not just Uptane. In particular, since automated verification tools are mostly black-box, remedying verification issues (in the form of non-termination, excessive manual effort, etc.) takes expert knowledge, and there is no one-size-fits-all solution. In our analysis of Uptane, we found that none of prior approaches from the literature [32, 33, 37, 38, 41, 48, 51] were sufficient to address the above challenges (see Section 3).

Approach. Our automated analysis approach takes three inputs, namely, (a) a formal protocol model of Uptane, (b) a list of security guarantees formalized as temporal safety properties, and (c) a threat model. It then exhaustively checks whether the design satisfies all the properties with respect to the given threat model. In case of a violation, like any protocol verification approach, it outputs a *counterexample* as an evidence of violation. Our overall approach is aided by the following insights. We stress that *although the insights are demonstrated for the analysis of the Uptane protocol, they are equally applicable to the analysis of other protocols.*

❏ *A highly automated, staged protocol analysis.* Our staged analysis approach follows an *abstraction-refinement* paradigm where a cryptographic protocol verifier (CPV) and a model checker are ***eagerly*** combined. In contrast to prior *lazy combination approaches* [32, 33], our approach enjoys a higher degree of automation due to the use of an infinite-state model checker and a novel encoding of a *replay attacker.* The key idea behind the eager combination strategy is that one can identify relevant CPV lemmas, execute all of them ahead of time, cache the results, and then execute the model checking phase while incorporating the CPV's feedback, without *any* repeated work or manual intervention. This sort of analysis allows us to perform a fine-grained reasoning over message payload values with a very high level of automation. A detailed comparison (see Table 3) with prior approaches is discussed in Section 3.

❏ *Protocol step and input trace compressions.* Our analysis takes advantage of two optimization techniques: *protocol step compression*, which coalesces different protocol steps into a single one, and *input trace compression*, which only considers time steps where something meaningful happens. Although optimizations like these are sometimes used in formal verification for computational tractability, a differentiating factor in our work is that we have *proven* that our optimizations preserve the correctness of the analysis. That is, the optimized analysis will *provably* miss no attacks and never introduce spurious ones in comparison to the unoptimized modeling. (The proof is included in a separate technical document [47].)

❏ *Meta-level protocol analysis.* Our approach also enables a meta-level analysis of optional features, reducing the manual efforts to a feasible level. In particular, it allows us to formulate only 72 protocol verification instances (instead of 1152) by considering all possible combinations of the optional features together, which is a 16× reduction. This meta-level analysis is facilitated by a new feature of certain model checkers called *blame assignment* [45, 46], a technique that determines a minimal set of boolean model parameters that must be true in order to falsify a given property. In our case, the boolean model parameters of interest are the protocol's optional features, where setting one of those parameter to *true* disables the corresponding feature.

Prior Analyses of Uptane. An initial analysis of Uptane was presented by Boureanu [9]. Our analysis is more comprehensive with respect to the set of threat models and desired security properties. Additionally, Boureanu reports that the execution time for proving some of the more difficult lemmas with the Tamarin verifier [51] ranged from 20+ hours to non-terminating. In contrast, our novel insights into automated protocol verification lead to termination in under 10 minutes in a vast majority of cases. The difference between our approach and Boureanu's [9] is discussed in depth in Section 3 and outlined in Table 8 in the appendix.

Other Related Work. Other applications of formal verification to Uptane [37, 38, 48] are discussed in Section 8. More generally, state-of-the-art approaches to formal verification of security protocols include *symbolic model verification* [8, 51], *computational model verification* [4, 7], and an *abstraction-refinement paradigm* [32, 33] that combines symbolic model verification with general-purpose model checking. *Blame assignment* [45, 46] is a meta-level analysis technique which we discussed in the **Approach** paragraph. *Compression* techniques make system models more amenable to formal verification tools by minimizing the length of counterexample traces (e.g., [2] with "step compression").

Evaluation and Findings. The novel aspects of our work allow us to achieve termination in 69 of 72 attempted protocol verification instances (see Table 5) when comprehensively analyzing the Uptane protocol design. We achieve termination on more instances than other approaches, as discussed later. All our models are publicly available at https://github.com/lorchrob/UptaneModels.

Our analysis rediscovered five known attacks, which is an initial validation of the correctness of the approach. In addition, we discovered six *new* vulnerabilities (see Table 6), all confirmed by the standards body, which allow adversaries to prevent vehicles from installing new updates and also cause vehicles to install the wrong updates.

Contributions. In summary, the paper makes the following technical contributions: (*i*) a *formalization* of the Uptane protocol from a natural-language specification containing gaps and ambiguities; (*ii*) six *new security vulnerabilities* in the protocol; (*iii*) *proofs for the absence of attacks* with respect to our model for 14 combinations of properties and threat models; and, more generally, (*iv*) a novel *workflow* that enables the faithful modeling of a variety of adversarial scenarios and optional protocol features in a cryptographic setting.

2 PRELIMINARIES

System description. Uptane aims to *protect the ECU software update process*, focusing specifically on the delivery and verification of software updates. Figure 1 gives an example ECU network architecture with two network segments (called *buses*): a Controller Area Network (CAN) bus and a Local Interconnect Network (LIN) bus. Different network segments are connected by *gateway ECUs* that forward traffic from one bus to another.

ECUs updates are usually installed OTA. Since many ECUs lack computing power, only a small number of the ECUs connect to a remote repository over the internet to retrieve new software images for themselves and other ECUs in the vehicle. The new images are distributed to the other ECUs over the buses.

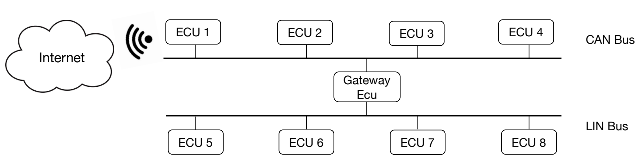

Figure 1: Example ECU Network Architecture

Motivation. Automotive buses were designed with efficiency and convenience in mind, so they *do not include built-in security features* [10]. Therefore, compromising even a single ECU renders the vehicle's internal network vulnerable to man-in-the-middle attacks. Many techniques to compromise ECUs are *completely remote*. Some common attack vectors are through the OBD-II port, vehicle entertainment systems (*e.g.* CD drive, USB port), keyless entry systems, and bluetooth and cellular channels [14].

Most efforts in automotive bus security have focused on CANs. Efforts in CAN security [6, 24, 40, 52, 54, 55, 60, 61, 67] employ a threat model where the adversary can arbitrarily read, drop, and inject messages in communications between ECUs. Notably, the model allows *ECU impersonation attacks*, where non-safety-critical ECUs can drop CAN packets sent by safety-critical ECUs or inject CAN packets that impersonate safety-critical ECUs [18, 24, 66]. Other studies in automotive bus security are surveyed by El-Rewini *et al.* [23] — for example, LIN bus message injection attacks involving the error handling mechanism [63]. There are recent news stories of thieves using CAN injection attacks to steal (relatively new) cars [17, 65].

Several studies show realistic man-in-the-middle (MiTM) attacks between an ECU and a communication partner outside the vehicle. For example, Rouf *et al.* [58] demonstrates a remote attacker's ability to tamper with the tire pressure monitoring system by spoofing ECU sensor data, while Köhler *et al.* [39] involves a remote attacker executing a DoS attack on EV charging through a MiTM attack on the charging protocol.

2.1 Uptane Background

In the Uptane [26, 41] ecosystem (see Figure 2), each vehicle has a *primary ECU* that communicates with two remote repositories, the *director* and *image* repositories. The director repository provides metadata about the software images the ECUs should install. An additional set of metadata, and the images themselves, are retrieved from the image repository. ECUs cross-reference metadata from both repositories, enabling the detection of an attack when only one of the repositories is compromised. These checks on the metadata are called *metadata verification*, and they are performed by all ECUs in the vehicle. There are four types of metadata: *root*, *timestamp*, *snapshot*, and *targets*. Some pieces of data are present in *every* metadata file, including a metadata version number and expiration timestamp. For specifics about other information contained in metadata files, consult Appendix B. The primary ECU downloads and verifies images and metadata, which it then distributes to the other ECUs in the vehicle, the *secondary ECUs*. If an ECU is compromised, other ECUs may still resist downloading malicious software, as each ECU performs its own verification. Thus, verification is performed twice on metadata and images for secondary ECUs — once by the primary ECU, and once by each of the secondary ECUs.

Uptane Update Process. An ECU's software update process can roughly be divided into three stages: (*i*) update discovery, (*ii*) metadata verification, and (*iii*) image verification and acquisition. A high-level Uptane description is given in Figure 8 in the appendix.

Update discovery: In this phase, new images are selected for the ECUs to install. This selection is performed by the director repository, which retrieves information about the vehicle's currently

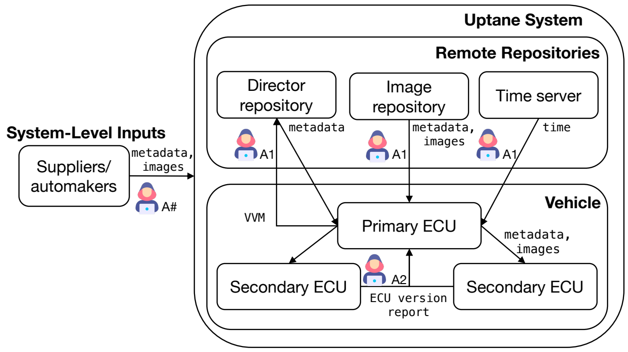

Figure 2: Simplified system architecture of Uptane.

Attack Type	Possible Implications
Freeze (P1)	Expose deprecated vulnerabilities
VVM Replay (P2)	Expose deprecated vulnerabilities, hampered vehicle functionality
Rollback (P3)	Expose deprecated vulnerabilities
Arbitrary Software (P4)	Adversary control over vehicle
Attacker-Authored VVM (P5)	Expose deprecated vulnerabilities
Mix-And-Match (P6)	Hampered vehicle functionality
Mixed-Bundles (P7)	Hampered vehicle functionality
Incompatible Image (P8)	Hampered vehicle functionality

Table 1: Possible implications of attacks

installed information (aka vehicle version manifest or VVM, for short) and performs dependency resolution to select new updates.
Metadata verification: In this stage, ECUs process metadata from both remote repositories, applying verification procedures to ensure that the metadata is not tampered with.
Image verification and acquisition: In this stage, ECUs apply a verification procedure to the new images retrieved from the image repository. If this verification is successful, then the ECU installs the new image. If any of the steps above fails, the offending file (metadata, image, or VVM) is discarded and the cycle is restarted with a fresh update discovery. For concrete steps regarding the above stages, consult Appendix B.2.

Metadata verification is flexible for secondary ECUs. Secondary ECUs with less computing power may choose to verify, at a minimum, only targets metadata files from the director repository (*partial verification*), rather than verifying the full set of metadata. In our analysis, we model the entire Uptane system but focus on the security properties of secondary ECUs performing minimal partial verification. Uptane claims to offer *flexibility* without compromising *security*, and we put this claim to the test.

We omitted from the above discussion another type of verification: *verification of the latest time*. Since ECUs often do not have accurate internal clocks, the Uptane system allows for a remote attestation of the latest time, which ECUs must verify before performing metadata verification. However, the specifics of the ECUs' access to time are not discussed in the Uptane specification; instead, the source of time is assumed to be secure. In our analysis, we consider the possibility of a man-in-the-middle attack between the ECU and the (remote) secure source of time.

3 PROBLEM DEFINITION AND CHALLENGES

The underlying problem we solve requires formally analyzing the newest version of the Uptane protocol (version 2.1) with respect to a set of properties and a set of threat models. We discuss them both in this section, together with the technical challenges they pose.
Desired Functional Properties. We consider eight integrity properties. We do not model confidentiality or availability properties as such properties are not promised by Uptane. The first two properties relate to *freshness*, stating that ECUs and remote repositories will not accept old versions of files when newer versions are available.

P1. ECUs only verify the latest available targets metadata (checks for *freeze attacks*).
P2. The director repository only verifies the latest available vehicle version manifest (VVM, containing information about each ECU's currently installed image) that was sent by the vehicle (checks for *VVM replay attacks*).

If these properties are violated, the ECUs will continue to run old software images that potentially contain known bugs or security vulnerabilities which could be exploited by an adversary in another attack. Possible implications of the attacks under analysis are outlined in Table 1.

The remaining properties are about ECUs installing the correct software images. Violating them could result in ECU installing (i) older images with known vulnerabilities, (ii) adversary-authored images, or (iii) images with compatibility issues. This can lead to a degradation in vehicle functionality or, in the worst case, adversary control over the vehicle.

P3. ECUs always verify images in nondecreasing version order (checks for *rollback attacks*).
P4. ECUs never verify adversary-authored software (checks for *arbitrary software attacks*).
P5. Remote repositories never verify VVMs containing version reports that were authored by an adversary (checks for *attacker-authored VVM attacks*).
P6. ECUs never verify metadata instructing installation of an incompatible set of images (checks for *mix-and-match attacks*).
P7. ECUs always have a compatible set of installed images (checks for *mixed-bundles attacks*).
P8. ECUs never verify images incompatible with their own hardware (checks for *incompatible image attacks*).

Threat models. We consider an adversary who can read, modify, inject, and drop protocol packets within the vehicle's internal network *and* between the vehicle and remote communication partners, all the while conforming to cryptographic assumptions. This problem reduces to protocol analysis in the symbolic model where selected components (including components within the vehicle) are assumed to be under the influence of a Dolev-Yao[21] adversary.

We consider the following threat models with various adversarial capabilities. Each threat model contains a set of man-in-the-middle attackers (see Figure 2) and access to specified compromised keys.

B. The benign case (no adversary).

Adversarial Capability	**Feasibility**
A1 message reading	[58], [39]
A1 message injection	[58], [39]
A1 message dropping	[58], [39]
A2 message reading	[6, 18, 40, 52, 54, 61, 63, 67]
A2 message injection	[6, 18, 24, 40, 52, 54, 61, 63, 65, 67]
A2 message dropping	[40, 54, 61, 63]
A3-A6, A* key compromise	discussed in Section 9
A# supply chain attack	[16, 35, 62]

Table 2: Feasibility of Adversarial Capabilities. A1 operates outside the vehicle, while A2 operates also inside (Figure 2).

A1. The adversary can perform MiTM attacks on network connections *outside* the vehicle (A1 in Figure 2), but not in the vehicle's internal network.
A2. The adversary can perform MiTM attacks *outside and inside* the vehicle (A1 and A2 in Figure 2).
A3. As in A2 but the adversary has also access to compromised director repository metadata signing keys.
A4. As in A3 plus adversary access to compromised image repository timestamp and snapshot metadata signing keys.
A5. As in A4 plus access to compromised image repository targets metadata signing keys.
A6. As in A5 plus access to compromised image repository root metadata signing keys.
A*. As in A1 plus access to the compromised VVM signing key used by the primary ECU.
A#. The adversary has compromised the supply chain (A# in Figure 2).

References to justifications for the technical feasibility of each of these adversarial capabilities are listed in Table 2.

Need for Different Threat Models

While the Dolev-Yao [21] model is generic enough to capture the behavior of all of the above threat models, we still must specify which network connections are susceptible to a Dolev-Yao attacker, as well as which cryptographic keys the Dolev-Yao attacker is assumed to have access to.

3.1 Technical challenges

We now illustrate technical challenges by discussing the shortcomings of five existing protocol verification approaches: (1) *manual analysis*, (2) *cryptographic protocol verification in the computational model*, (3) *cryptographic protocol verification in the symbolic model*, (4) *model checking*, and (5) *the abstraction-refinement paradigm*, which lazily combines a symbolic model checker and a symbolic cryptographic protocol verifier.

(1) **Manual Analyses.** A naive approach is to manually analyze the protocol to verify its security properties. However, our results (discussed in Section 7) provide strong evidence that manual analysis can miss violations easily identified by automated reasoning.

(2) **Computational Cryptographic Protocol Verifiers.** Protocol verifiers in the computational model (*e.g.*, [4, 7]) represent protocol messages as bitstrings, and cryptographic operators as functions from bitstrings to bitstrings. However, this precise, low-level modeling imposes a cost — high manual effort. For Uptane, this difficulty is exacerbated by the presence of optional features, which exponentially increase the required modeling effort. *In this paper, we instead focus on techniques with high automation.*

(3) **Symbolic Cryptographic Protocol Verifiers.** Protocol verifiers in the symbolic model (*e.g.*, [8, 51]), in short **CPV**s, assume cryptographic building blocks to be secure (*i.e.*, the *perfect cryptography assumption*). CPVs only reason about the composition of cryptographic primitives, making the protocol verification task more amenable to automation. We tried this approach first, with the Tamarin CPV [51], but we faced two major challenges. The first, which we will call $\mathbf{C_1}$, is that the analysis failed to terminate. One reason for this is that the search space is large, as Uptane's update process includes a minimum of 10 steps in total (one for VVM verification, one for each metadata file for each repository, and one for image verification), each relying on ECU or repository state. Specifically, the CPV analysis times out when attempting to prove properties that require reasoning over multiple verification cycles (causing *long execution traces*). The second reason is that Uptane requires fine-grained modeling of infinite-domain data types, arithmetic operations, and temporal operators (*e.g.*, not just standard correspondence and injective-correspondence proofs over symbolic messages). While Tamarin supports these features, the analysis was not scalable enough for Uptane. To address the termination issue, we employed custom heuristics (a Tamarin oracle) to help the CPV's proof search, but it was not sufficient for termination.

The CPV termination issue $\mathbf{C_1}$ is also observed by Boureanu in a recent formal analysis of Uptane using Tamarin [9]. In her analysis, powerful compute servers (400GB RAM) were needed, and some proofs still took 20+ hours to complete or failed to terminate. Compared to this approach, ours gives a better performance, even across a wider set of threat models and desired security guarantees in a commodity laptop. For example, [9] considers a hierarchy of five threat models, where the more powerful adversaries can compromise more components of the model (key compromise is not independently considered). Compared to this, our threat hierarchy of nine threat models is more fine-grained, as it explores all possible key compromises. While it may seem like our threat models do not analyze component compromise, it is equivalently captured by the compromise of all the component's keys (*e.g.*, threat model A* effectively models a primary ECU compromise). Additionally, [9] considers seven security properties, three of which are from the standard (relating to confidentiality, denial-of-service, and arbitrary software attacks), and four of which they independently formulated (one dealing with privacy, and two dealing with agreement/synchronization between model components). However, two of the properties from the standard (confidentiality and denial-of-service) are trivially broken, and one of the independently formulated properties is very close to protection from freeze attacks (from the standard). We consider eight security properties, *not including the two trivial ones*, including five from the standard and three independently formulated. The three that we formulated relate to VVM replay attacks and image installation, and thus have a much higher potential impact if broken (because they relate to update *integrity*). Table 8 in the appendix summarizes the comparison of the two approaches.

> **Impact of Non-termination**
> *If the analysis does not terminate for a given Uptane problem instance (defined by a property, threat model, and a set of optional features), we cannot conclude anything about the property under the threat model for the Uptane protocol design, leaving a potentially undiscovered vulnerability.*

Another challenge (C_2) is that Tamarin is unable to efficiently perform a meta-level search over the Uptane framework's set of optional features. That is, our initial attempt only covered one specific instantiation of the framework. However, we seek to analyze the entire set of all possible (1172) instantiations.

(4) **Model checking.** Another tool-based approach is to use a general-purpose model checker for protocol analysis. General-purpose model checkers excel in fine-grained modeling of infinite-domain data types, arithmetic operations, and temporal operators compared to CPVs. We tried this approach this using Kind 2 [13]. However, Kind 2 (as expected) produces cryptographically infeasible attack traces, as it is not designed to reason about Dolev-Yao adversaries. In addition, Kind 2 had non-termination issues due to long execution traces (C_1).

(5) **Abstraction-Refinement.** A fifth approach is the abstraction-refinement paradigm [32, 33], which in this case involves the lazy combination of a cryptographic protocol verifier and a general-purpose model checker. The purpose of the approach is to achieve efficient fine-grained reasoning with a model checker, while reasoning about cryptographic aspects with a CPV. In short, the approach consists in using a model checker to find an attack trace and then invoking a CPV to determine if the counterexample trace is cryptographically feasible. If it is, the trace returned by the model checker describes a potential attack. If it is not, the falsified property is updated to block the infeasible counterexample. Unfortunately, even this approach is fundamentally unable to perform a rich, detailed analysis. Its first challenge, denoted by C_3, is that it relies on predicate abstraction over message payload data. However, in the Uptane use case, an analysis of concrete payload values is required because the desired system-level properties involve arithmetic constraints over packet values (*e.g.*, counters, version numbers, and timestamps). For example, one system-level safety property is protection from rollbacks, formulated as `new_ecu_img.version` ≥ `curr_ecu_img.version`. Here, `new_ecu_img.version` is part of an incoming message payload and *must* be modeled as an integer to accurately capture the property. This approach also suffers from C_2, as it does not have support for a meta-level search over optional features in an automated, scalable way. Additionally, the approach does not address C_1, as the execution traces are still infeasibly long and cause non-termination issues. Finally, this approach requires an undesirably high number of manual interventions in the workflow (C_4).

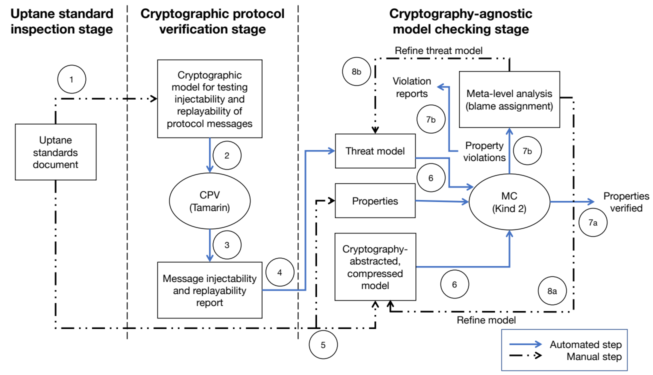

Figure 3: Our Protocol Analysis Workflow (Eager Approach).

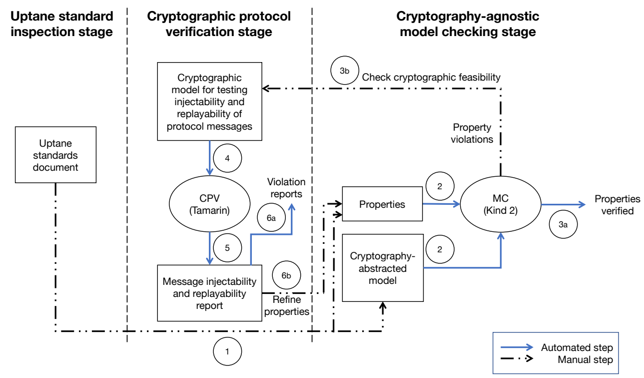

Figure 4: Protocol Analysis Workflow, Lazy Approach [32, 33].

4 APPROACH OVERVIEW

From the above approaches, we consider the abstraction-refinement paradigm to be the most promising as it can reason about both cryptographic and rich protocol features (*e.g.*, statefulness, constraints over payloads). However, we still face four core challenges: long execution traces cause non-termination (C_1); there is no mechanism for meta-level analysis over optional protocol features (C_2); message payload data is modeled with predication abstraction, disallowing analysis of concrete payload values (C_3); and there must be manual intervention for every counterexample trace generated by the model checker (C_4). We provide a high-level description of our approach and explain how it is a fundamental improvement over prior work [32, 33] next, using a running example.

4.1 High-level Approach Overview

The high-level approach is outlined in Figure 3. In step ①, we construct a CPV model by consulting the Uptane standards document. In the CPV model, we abstract away control flow and statefulness, only modeling the cryptographic aspects of the protocol. In step ②, we consult CPV to determine the adversary's cryptographic capabilities, generating a report (step ③). For example, the adversary may not be able to modify messages *arbitrarily*, as this could result in breaking cryptographic assumptions (*e.g.*, an adversary forging a digital signature without knowledge of the corresponding secret signing key). In step ⑤, we create system $\mathcal{S}$ and model the desired properties by consulting the Uptane standards document. We model an adversary who modifies messages sent between model components (protocol participants). Based on the CPV report from step ③, we instrument each of $\mathcal{S}$'s network connections in step ④ by making each connection vulnerable to either (i) no attacks, (ii) only *replay* attacks (*i.e.*, by adversaries that can only replay past messages but not inject new messages), or (iii) arbitrary injections attacks (from unrestricted adversaries). Specifically, if CPV's corresponding weak authentication lemma is disproven, we consider a standard attacker, where the injection of arbitrary messages is possible. If CPV's corresponding weak authentication lemma is proven but the injective authentication lemma is disproven, we consider a replay attacker, as replays are feasible but new message injections are not. If CPV's injective authentication lemma is proven, then we mark the connection as invulnerable to all attacks, as even replays are infeasible. The relationship between the CPV lemma results and adversarial capabilities is outlined in Table 4. In step ⑥, we execute the model checker with the corresponding threat model and property. Either the property is verified (step ⑦a), or there is a counterexample trace (step ⑦b).

Contrast Between Lazy and Eager Approaches

The lazy approach (see Figure 4) requires crossing back into the CPV verification stage through arrow 3b (a manual backtracking step). This is completely eliminated in the eager approach (see Figure 3).

4.2 Our Approach with Working Example.

We view our approach as an idealized version of the prior abstraction-refinement paradigm [32, 33]. We walk through a working example, pointing out when one of the challenges $\mathbf{C_1}, \ldots, \mathbf{C_4}$ arises. In this section, instead of immediately explaining how each challenge is addressed, we simply assume that there exists some way to overcome it. We will outline our novel insights and how they enable us to overcome each challenge in the next section.

We will walk through a simplified version of the analysis of desired functional requirements P3 and P6, using the idealized workflow. In English, the properties are "ECUs always verify images in nondecreasing version order" and "ECUs never verify metadata that instructs them to install an incompatible set of images," respectively. Both analyses are primarily concerned with the secondary ECUs' verification of targets metadata, which is graphically illustrated in Figure 5. Specifically, Figure 5 illustrates the high-level steps taken by the various components, numerically indexed to indicate the order of execution. Additionally, the pseudocode describes some of the finer details of each component's local actions. At a high level, targets metadata is first generated and sent from both repositories to the primary ECU. The ECU performs verification of both metadata files individually, and also performs a cross-reference of the two files to hedge against either repository (or the connection to either repository) being compromised. Finally, the primary ECU sends targets metadata to the secondary ECU, which performs its own verification of metadata. However, in this case, the secondary ECU only processes the director repository's metadata, so cross-referencing is not possible.

We start with an analysis of P6. In this case, we are assuming threat model A3 (internal adversary with compromised director targets metadata keys).

Create CPV Model. We construct a CPV model by consulting the Uptane standards document.

Execute CPV. For P6, the most relevant lemmas are those corresponding to (i) sending targets metadata from the director repository to the primary ECU (step ①), and (ii) sending targets metadata from the primary ECU to the secondary ECU (step ④). In both cases, the strong authentication lemma fails, meaning that targets metadata replays are cryptographically feasible. However, in case (ii), the weak authentication lemma *also* fails with a counterexample trace where the adversary injects an arbitrary targets metadata file, and it is still verified by the secondary ECU.

Create Model Checker (MC) Model. We create system $\mathcal{S}$ by consulting the Uptane standards document. *We assume, in an ideal world, that the model is constructed such that counterexample traces are short and that message payload data is modeled concretely (without predicate abstraction), addressing* $\mathbf{C_1}$ *and* $\mathbf{C_3}$.

Specify MC Threat Model. In our example, the results from the CPV model mandate that a *replay attacker* should be placed on the connection from the director to the primary ECU (step ①), and a *standard attacker* should be placed on the connection from the primary ECU to the secondary ECU (step ④).

Execute MC. We execute the model checker with the corresponding threat model. *We assume, in an ideal world, that there is a meta analysis which automatically determines which optional protocol features (if any) contributed to the property violation, addressing* $\mathbf{C_2}$.

For P6, MC produces a counterexample (with no optional features disabled) where an adversary injects a targets metadata file on the connection between the primary ECU and the secondary ECU (step ④). This metadata file contains instructions for the ECUs to install an incompatible set of images, representing a violation of P6. *We assume, in an ideal world, that for analyzing P3, we do not have to repeat steps "execute CPV" and "specify MC threat model." This addresses challenge* $\mathbf{C_4}$.

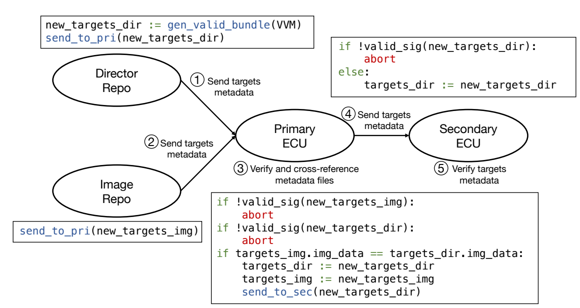

Figure 5: Simplified Targets Metadata Verification Overview.

4.3 Working Example (Prior Work)

We now analyze P3 and P6 again, this time with the approach in [32, 33], to further illustrate the extra manual steps it requires. We

Characteristic	Our approach	Hussain *et al.* [32, 33]	Tamarin [51]
MC + CPV combination strategy	Eager	Lazy	NA
Type of instrumentation	Model instrumentation	Property instrumentation	NA
Concrete message payload?	Yes	No	Yes*
Meta-level analysis (blame assignment)?	Yes	No	No
Compression with meta-theorem?	Yes	No	Yes*

* With scalability issues

Table 3: Comparison of various approaches

Relationship between attacker capabilities and Auth. Lemmas	*Auth. Lemmas Proven (✓)/ Falsified(×) by CPV*	
	Weak Auth. Lemma	**Strong Auth. Lemma**
Message injection, replay both possible	×	×
Message replay possible, not injection	✓	×
Neither possible	✓	✓

Table 4: Relationship between Authentication Lemmas and Attacker Capabilities (*i.e.*, message injection, message replay)

start with P6, assuming threat model A3 (internal adversary with compromised director targets metadata keys).

Execute MC. We start with executing the cryptography-agnostic MC model. In this case, MC generates a counterexample trace where a targets metadata file is injected on the connection between the director repository and the primary ECU (step ①).

Execute CPV. We must check the cryptographic feasibility of the attack. In this case, the weak authentication lemma for targets metadata being sent from the director to the primary is proven (step ①), so the trace is a spurious counterexample.

Property Instrumentation. To address the spurious counterexample, *P*6 is instrumented to state "If there is no arbitrary targets metadata injection on the connection from the director to the primary, then ECUs never verify metadata that instructs them to install an incompatible set of images."

Execute MC. We invoke MC again with the updated property, generating a trace where the targets metadata file is injected on the connection from the primary ECU to the secondary ECU (step ④).

Execute CPV. Again, we check for cryptographic feasibility. Here, the weak authentication lemma for targets metadata sent from the primary ECU to the secondary ECU fails, so the attack is cryptographically feasible. ***To analyze P3 in the lazy approach, the above five steps all have to be repeated.*** This is, in part, because the updates to our system model that block cryptographic traces were *property-specific*, *i.e.*, our analysis for P6 does not carry over to P3. Further, this is not the only problem. As discussed in the previous subsection, the abstraction-refinement paradigm still lacks mechanisms for dealing with long execution traces ($\mathbf{C_1}$), performing meta-level analysis over optional protocol features ($\mathbf{C_2}$), and modeling message payload data without predicate abstraction ($\mathbf{C_3}$).

5 TECHNICAL INSIGHTS

In this section, we present an overview of our key technical insights and how they enable the idealized workflow described in the previous section. We outline the differences between our approach and that of Hussain *et al.* [32, 33] in Table 3.

Insight 1: Compression. To address C_1, we introduce two compression techniques, *input compression* and *model compression*, which dramatically reduce the length of execution traces. The core ideas are that (i) stuttering can be eliminated from modeling, and (ii) multiple Uptane protocol steps can be coalesced into a single step while provably preserving semantic correctness with respect to the properties under analysis. In Hussain *et al.*, the length of attack traces makes analysis intractable. However, with compression, attack traces are limited to a quarter of the length as compared to without compression, leading to termination in more cases (see Table 5). A detailed discussion of compression is in Section 6.

Insight 2: Blame Assignment. To address $\mathbf{C_2}$, we model optional security-enhancing features symbolically as Boolean parameters in our model and use the *blame assignment* feature of the Kind 2 model checker [45, 46] for fine-grained information about the features that are insufficient for protecting against a certain attack. The blame assignment functionality on a model attempts to find a minimal truth assignment of some given Boolean parameters that is sufficient to trigger a violation of the property. In the truth assignment identified by blame assignment, having value *true* for the Boolean parameter corresponding to an optional feature suggest that the protocol should disable that feature. As a concrete example, consider two optional features in Uptane: (i) repositories should increment version numbers of metadata files when they are updated, and (ii) targets metadata should include image version numbers. With two optional features, there are four concrete protocol instantiations (an example instantiation is to require that version numbers are incremented, but allow targets metadata files to forgo version numbers). In Hussain *et al.*'s approach, the entire workflow has to be replicated for each optional feature (in general, 2^n times for n optional features) to determine which optional features must be disabled for which attacks.

Impact of Blame Assignment

In our approach, a single run of the workflow is comprehensive, as the model checker's blame assignment feature searches for attack traces that minimize the number of disabled optional features. In our Uptane analysis, we consider four optional features, with a 16-fold reduction in the manual analysis effort.

Insight 3: Replay Attacker. The most common approach of incorporating an abstract version of Dolev-Yao-style adversary is to place a component between every public communication channel that can non-deterministically modify the data being sent through the channel. Without further behavioral restrictions, this adversary actually violates the requirement of the original Dolev-Yao model as it may not conform to cryptographic assumptions (*e.g.*, decrypting a ciphertext without possessing the decryption key). In particular, when the CPV analysis suggests that the attacker can only replay packets but cannot forge new packets, placing a *replay* attacker is more accurate. We designed a non-trivial way to explicitly capture the behavior of a bounded or unbounded replay attacker (see **Insight 6** and Appendix D for more details). The use of the replay attacker helps us address both C_4 and C_3.

Insight 4: Eager combination with Replay Attacker. To address C_4, we optimize the workflow from Hussein *et al.* [32, 33], reducing the number of manual steps. One major difference regards the combination of the MC and CPV analyses. Those previous works combine them *lazily* as they account for cryptographic influence through *property instrumentation* of the form $\alpha \rightarrow \beta$, where β is the property under consideration and the antecedent α rules out cryptographically infeasible traces. In contrast, we combine the two analyses *eagerly*, accounting for cryptographic influence through *model instrumentation* with the replay attacker.

We demonstrate how we differ with an example. Suppose the existence of four functions, f, g, adv, and $main : \mathbb{Z} \rightarrow \mathbb{Z}$, where we want to prove some properties about the output of $main$:

$$\begin{array}{rclcrcl} f(x) & = & |2x| & \quad & adv(x) & = & ??? \\ g(x) & = & 3x & \quad & main(x) & = & g(adv(f(x))) \end{array}$$

We aim to analyze the output of $main$, which composes g and f through adv, where adv models an adversarially-controlled function. In this setting, adv is an *uninterpreted function*, that is, a function whose signature is known but its semantics is not. This simulates the modeling of an adversary whose capabilities are only knowable by consulting an oracle (representing a CPV). Assume in this case that it is only possible for adv to return even integers.

Suppose then we want to verify two properties φ_1 and φ_2 using a suitable automated reasoner (for instance, an SMT solver), where φ_1 states that the output of $main$ is always even and φ_2 states that the output of $main$ is always positive. With no adversarial influence, *i.e.*, when adv is the identity function, both properties trivially hold. In the lazy approach, the behavior of adv is initially unconstrained. So, in the first SMT query for φ_1, a spurious counterexample is raised where adv returns -1 (or in general, any odd negative number). The counterexample is manually inspected to find that the property failure was due to the injection of an arbitrary value. Only then does the lazy approach invoke the oracle and determine that adv's behavior was infeasible. Consequently, φ_1 is instrumented to rule out the attack and rewritten in the form of $\alpha \rightarrow \varphi_1$, where the antecedent α specifies that adv must return an even integer. An inefficiency of this approach is that when querying φ_2, the same initial counterexample (where adv returns -1) is raised, but the manual work of counterexample inspection and property instrumentation has to be duplicated for φ_2.

We improve this process by invoking the CPV ahead of time and specifying adv's behavior in the model, *not* the property, by giving adv a concrete definition. This minimizes manual trace inspections and completely avoids the need to block similarly infeasible counterexamples multiple times. In other words, the lazy approach from Hussain *et al.* involves twice as much manual work in this example than our approach: *two* CPV invocations and property instrumentations against just a single CPV invocation and model instrumentation. If the CPV results indicate that protocol components are vulnerable to message replays, then model instrumentation is performed using the replay attacker, discussed previously. (If protocol components are vulnerable to injections of arbitrary messages, we use the standard technique of placing an adversarial component that can arbitrarily, nondeterministically modify messages sent between components, as discussed in **Insight 3**.)

Significance of the Eager Approach
If there are n counterexamples targeting the same cryptographic vulnerability, the eager approach takes n times less manual effort than prior work [32, 33].

Insight 5: Context Independence. We improve the approach further to address C_4. We note that in general, we cannot rely on CPV's output without manually inspecting counterexample traces. For example, if CPV says that a replay attack is possible, then manual analysis of counterexample traces is needed to determine if the attack is *generalizable*, or if it can only take place under specific circumstances. If it is the latter, the adversary's behavior cannot be accurately modeled with a replay attacker, as additional restrictions are needed on the adversary's behavior. In our analysis of all the CPV lemma counterexamples, we found that they were all generalizable, *i.e.*, context independent. This enabled us to invoke the CPV only once for each lemma in the workflow, rather than having to revisit the CPV for each discovered attack.

Insight 6: Payloads with Replay Attacker. To address C_3, we appeal to the *principle of maximum logical revelation*, which states that as many details as possible should be captured in the structure of logical formulae, rather than being abstracted away in the atoms. The replay attacker, described previously, is not restricted to any specific datatype, so it can naturally reason over concrete (*e.g.*, integer) values in message payloads. This was not feasible in Hussein *et al.* [32, 33] because their strategy of accounting for adversarial influence in the system model relies on predicate abstraction.

Consider adding a restriction to our Uptane system model that an arbitrary (previously unseen) ECU version report cannot be injected on the connection from component A to component B. For simplicity, assume the ECU version's payload can be accurately modeled with a pair of natural numbers $\langle a, b \rangle$. In Hussain *et al.*'s approach, the report is abstracted into a Boolean variable `v_rep`, and a property φ is instrumented to $(\mathit{Inject}(\texttt{v\_rep}) \Rightarrow \mathit{Once}(\texttt{v\_rep})) \Rightarrow \varphi$, blocking the injection of `v_rep` unless it has been sent before. This does not naturally extend to a more precise model of `v_rep` as a pair of natural numbers. Below, suppose `v_rep'` represents the value of the version report injected by the adversary. Blocking the injection

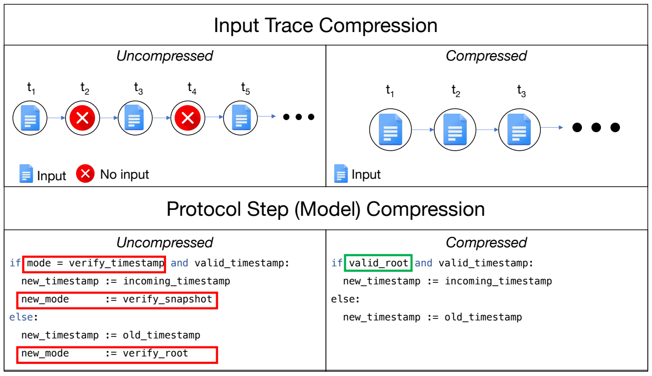

Figure 6: Illustration of Input and Model Compression.

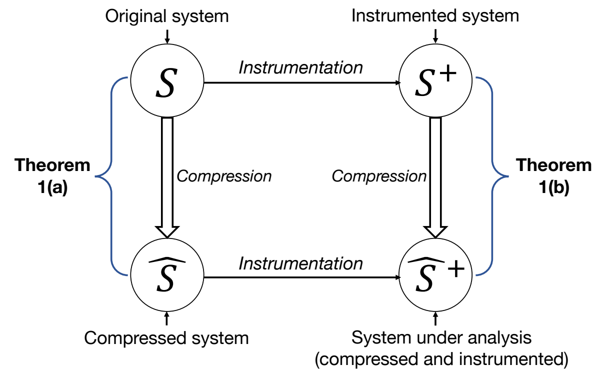

Figure 7: Compression.

of every pair of natural numbers with property instrumentation is not feasible, as it requires an infinite case split:

$$(\mathit{Inject}(\mathsf{v\_rep}') \land \mathsf{v\_rep}' = \langle 0, 0\rangle \Rightarrow \mathit{Once}(\mathsf{v\_rep} = \langle 0, 0\rangle)) \land$$
$$(\mathit{Inject}(\mathsf{v\_rep}') \land \mathsf{v\_rep}' = \langle 0, 1\rangle \Rightarrow \mathit{Once}(\mathsf{v\_rep} = \langle 0, 1\rangle)) \land$$
$$\dots$$
$$\Rightarrow \varphi$$

*The replay attacker, described previously (**Insight 3**), is thus necessary to model replay attacks generically over infinite domains.*

6 MODEL AND INPUT COMPRESSION

Motivation for Compression. Our model $\mathcal{S}$ of the Uptane framework is complex enough to make model checking unfeasible. The model checker times out for 12 system-level property/threat model combinations. To speed up the automated analysis of the model, we apply the model compression techniques described below.

We apply two types of compression, *input compression* and *model compression*, graphically illustrated in Figure 6. Intuitively, input compression only considers *meaningful* time steps, where input is provided at the system level. In contrast, model compression coalesces multiple protocol steps into a single step. Both types of compression reduce the length of verification steps (and hence, of attack traces).

Without compression, MC can only verify the absence of attacks in five of the eight properties in the benign case. With compression, all eight are verified, and we also get five additional proofs for the absence of attacks in other threat models. We denote by $\widehat{\mathcal{S}}$ the compressed model and by $\mathcal{S}^+$ the model obtained by instrumenting $\mathcal{S}$ with Dolev-Yao adversaries at selected communication points. Let then $\widehat{\mathcal{S}}^+$ denote the instrumented version of $\widehat{\mathcal{S}}$.

The results obtained with the compressed model $\widehat{\mathcal{S}}$ (resp., $\widehat{\mathcal{S}}^+$) lift to $\mathcal{S}$ (resp., $\mathcal{S}^+$) thanks to a meta-level theorem stating, intuitively, that proofs or violations of properties in $\widehat{\mathcal{S}}$ correspond to proofs or violations of properties in $\mathcal{S}$. The meta-theorem (Theorem 1) is described graphically in Figure 7. We point out that this correspondence between the compressed and uncompressed system is not a general result; it is restricted to $\mathcal{S}$ and the specific integrity properties we consider for Uptane.

Description of Compression. We present more details about $\mathcal{S}$ and $\widehat{\mathcal{S}}$ (the original and the compressed systems) and outline the differences between the two. In both $\mathcal{S}$ and $\widehat{\mathcal{S}}$, the system-level input is comprised of four metadata files (one for each metadata file type) and three software images (one for each ECU). The system-level output is comprised of each ECU's currently verified targets metadata and currently installed image, as well as the director repository's latest verified VVM. Both the system-level input and system-level output are *event-based*, intuitively meaning that input and output values can be present or not. The absence of a system-level input means that the corresponding image or metadata file is not available for the primary ECU to download.

In $\mathcal{S}$, a full metadata verification cycle spans nine *(execution) steps* of the Uptane protocol, where a step corresponds to a transition of the model from a state $\mathbf{s}$ to a state $\mathbf{s}'$. Eight steps are each associated with the verification of a single input metadata file, while the ninth step is used to cross-reference metadata from both repositories. Concretely, at every step, the primary ECU is in one of eight *(execution) modes*: (1) waiting for root director metadata, (2) waiting for root timestamp metadata, . . . , (8) waiting for image targets metadata, and (9) cross-referencing. At every step, the primary ECU attempts to verify the corresponding metadata file. If it succeeds, the primary advances to the next mode on the next step. If it fails, the cycle restarts. If there is no incoming metadata file to verify, the ECU considers this a failure and restarts the cycle. If the primary receives an input corresponding to a metadata file that is not currently being verified, it ignores it. Also, images are ignored until the last step of a successful verification cycle. In other words, ECUs do not install new images in the middle of metadata verification. Finally, the primary ECU does not produce output events in the middle of the verification cycle (but only when it starts a fresh verification cycle).

In $\widehat{\mathcal{S}}$, the sequential verification steps above in the primary ECU are collapsed to a single step where verification procedures are applied to all input metadata files in parallel. We call this compression *model compression*, to highlight differences in the definition of the system itself in the two models $\mathcal{S}$ and $\widehat{\mathcal{S}}$. For example, the lower half of Figure 6 demonstrates model compression with timestamp metadata verification. In the uncompressed model, the ECU has to be in the correct mode in order to update its current timestamp metadata. However, in the compressed version, no mode check is required, and the ECU can update a metadata file if it is valid (and all previous metadata files in the verification cycle were valid).

We also perform *input compression*, illustrated in Figure 6, which amounts to considering only execution traces that have a system-level input event at every step. Formally, this is achieved by adding the temporal logic assumption (**always** Event) to the set of system-level assumptions, where Event is a formula defined to be true in the current state if and only if a system-level input event is present.

Both forms of compressions aim at reducing the length of traces that MC has to reason about to find an *attack trace*, an execution trace that violates one of the properties, or rule out their existence.

The following theorem (which we prove in a separate technical document [47]) states, in essence, that applying compression on our model $\mathcal{S}$ does not change the satisfiability of the desired properties in either the benign case or with adversarial instrumentation. (In the theorem, $\models$ is the entailment relation in Linear Temporal Logic.)

Theorem 1. *Let A be a formula denoting assumptions on system-level inputs for $\mathcal{S}$. For each desired functional requirement R from Section 2:*

(a) $\mathcal{S} \wedge \textbf{always } A \models R$ *iff* $\widehat{\mathcal{S}} \wedge \textbf{always } (A \wedge \text{Event}) \models R$.

(b) $\mathcal{S}^{+} \wedge \textbf{always } A \not\models R$ *iff* $\widehat{\mathcal{S}}^{+} \wedge \textbf{always } (A \wedge \text{Event}) \not\models R$.

7 ANALYSIS FINDINGS

We now discuss the vulnerabilities uncovered by our protocol verification strategy and explain how they could be exploited. We use the Kind 2 [13] model checker to model the non-cryptographic aspects of Uptane and the Tamarin [51] CPV to model the adversary's capabilities with respect to cryptographic assumptions. Appendix D explains how to reproduce our analysis. Table 5 summarizes the results with respect to property violations, and Table 6 lists the attacks found using our workflow.

We consider extra attack scenarios A1 and A* which were not present in previous work [37, 38, 48]. For each property, in threat models where no attack was found, we distinguish between MC proving that the property holds and timing out (with ✓ and "?"). We demonstrate vulnerabilities of three types: (i) showing that previously-known attacks are possible with fewer adversary capabilities; (ii) analyzing attacks that were not previously considered (namely, replayed VVM, attacker-authored VVM, and incompatible image attacks, to be discussed soon); and (iii) demonstrating that the omission of optional features leads to a degradation in security. Our analyses focus on attacks against partial verification secondary ECUs. None of the attack scenarios require the adversary compromising the Primary ECU.

We found several new vulnerabilities. For brevity, we only discuss two findings (**Finding 1** and **Finding OPT1**) in the main text. Details about the rest of the findings are in Appendix E. To further clarify our analysis strategy, the first two findings discussed in the appendix explicitly outline the workflow steps necessary to discover the attacks.

Finding 1: Freeze Attack. The adversary causes a secondary ECU to re-verify the same targets metadata file rather than updating to the latest version.

Threat Model. Internal adversary (A2).

Vulnerability. An adversary (internal or external) can repeatedly replay the same metadata files to the ECU. The ECU checks that the metadata file version number (and for targets metadata, image release counters) are nondecreasing, which will pass. Also, the ECU will check if the metadata file's expiration time is later than the current time. The adversary can block an ECU's access to the current time by dropping the message containing the latest time, either between the external source of time and the primary ECU or between the primary ECU and the secondary ECU. Alternatively, the adversary can inject garbage time messages that will not pass the ECU's verification. The ECU's response to the adversary's tampering with time messages is not well defined in the standard.

In Section 5.4.3.1 of the standard [28], there is no indication to abort the update process if the attestation of the latest time is absent or invalid. This contrasts with Sections 5.4.3.2-5.4.3.4 [28], which recommend aborting if metadata or image verification fail. One could interpret this to mean that the verification process should continue *without the ECU updating its clock*, and so the most recent securely attested time would be identical to the last update cycle. This interpretation leads to the freeze attack succeeding.

For more information, one may consult the deployment recommendations [27]. It states in Section 3.1.1.1 [27] that the primary ECU should continue without updating its current time if it cannot verify a time message, further supporting the previous paragraph. But, Section 3.1.1.3 [27] says that *all* ECUs should abort, seemingly contradicting Section 3.1.1.1 [27] as well as the standards document— more clarification is needed to resolve the situation.

Detection. Based on Tamarin's output, we insert a replay attacker on the connection from the director to the primary and the connection from the primary to the secondary. By assumption, the source of time is secure, so the adversary's only capability is to drop the attestation of the latest time (there are no availability guarantees). This can be modeled by allowing the adversary to replay the time sent along the channel in the previous time step (both from the time server to the primary and from the primary to the secondary).

MC gives a counterexample for the property "When the secondary ECU verifies new targets metadata, the new targets metadata is the latest available." The adversary here drops the message from the primary to the secondary ECU containing the latest time and replays an older targets metadata file. The older targets metadata file is verified by the secondary, even though there is a newer targets metadata file available.

7.1 Analysis of Optional Security Features

A distinct advantage of our approach over previous work is that it makes it computationally feasible to reason about complex combinations of optional features. While some features of Uptane are *technically* optional, their exclusion would lead to immediate and obvious degradation in security. So, rather than disabling *all* optional features, we pick a reasonable subset, that is, we consider a few features whose absence may impact the integrity of the system: ($O1$) Repositories should increment version numbers of metadata files when they are updated (a repository may update the metadata file but leave the version number unchanged); ($O2$) targets metadata should include image version numbers (image version numbers are distinct from metadata version numbers); ($O3$) the

	Property																							
	P1			P2			P3			P4			P5			P6			P7			P8		
Threat Model	PW	WC	C	PW	WC	C	PW	WC	C	PW	WC	C	PW	WC	C	PW	WC	C	PW	WC	C	PW	WC	C
B	?	✓	✓	NA	✓	✓	?	?	✓	?	✓	✓	NA	✓	✓	?	✓	✓	NA	?	✓	NA	?	✓
A1	NA	×	×	NA	×	×	NA	?	✓	NA	✓	✓	NA	✓	✓	NA	✓	✓	NA	?	✓	NA	?	✓
A2	?	×	×	NA	×	×	?	?	?	?	✓	✓	NA	✓	✓	?	✓	✓	NA	×	×	NA	?	?
A3	?	×	×	NA	×	×	?	×	×	?	×	×	NA	✓	✓	×	×	×	NA	×	×	NA	×	×
A4	×	×	×	NA	×	×	?	×	×	?	×	×	NA	✓	✓	×	×	×	NA	×	×	NA	×	×
A5	×	×	×	NA	×	×	×	×	×	×	×	×	NA	✓	✓	×	×	×	NA	×	×	NA	×	×
A6	×	×	×	NA	×	×	×	×	×	×	×	×	NA	✓	✓	×	×	×	NA	×	×	NA	×	×
A*	NA	×	×	NA	×	×	NA	?	✓	NA	✓	✓	NA	×	×	NA	✓	✓	NA	?	✓	NA	?	?

✓ ↦ Attack proven impossible with respect to model, × ↦ Attack found
? ↦ Attack presumed impossible, but no proof, NA ↦ Not analyzed
PW ↦ Prior work[41], WC ↦ Our approach without compression, C ↦ Our approach with compression
Green highlight ↦ Needed compression for termination

Table 5: Partial Verification ECU Security Properties (no optional features omitted)

Attack	**Property violated**	**Our threat model**	**Prior threat model**	**Optional features omitted**	**Contribution**
Freeze	P1	≥ A1	≥ A4		⇊
VVM Replay I	P2	≥ A1	NA		★
VVM Replay II	P2	≥ A2	NA		★
Rollback I	P3	≥ A3	≥ A5		⇊
Rollback II	P3	≥ A2	NA	*O*1, *O*2	★
Arbitrary software	P4	≥ A3	≥ A5		⇊
Attacker-authored VVM I	P5	≥ A1	NA	*O*3	∇
Attacker-authored VVM II	P5	A*	NA		∇
Mix-and-match	P6	≥ A3	≥ A3		≡
Mixed-bundles	P7	≥ A2	NA		∇
Incompatible image	P8	≥ A3	NA		∇

NA ↦ Not analyzed or not specified, Gray ↦ no optional features omitted
★ ↦ New attack, ⇊↦ Weaker threat model, ≡↦ Equivalent to prior analysis
∇ ↦ Mentioned in a previous document, *e.g.* [27], but there was no analysis of adversarial capabilities necessary for the attack

Table 6: Attacks

director repository should verify digital signatures when checking VVMs for legitimacy; and (*O*4) the director repository should use information about image dependencies and conflicts when selecting images for vehicles to install. Without using blame assignment, comprehensively analyzing Uptane's security properties with respect to this set of optional features would *require us to execute the entire workflow* $2^4 = 16$ *times* — the blame assignment feature narrows this to just once, greatly reducing manual efforts.

Finding OPT1: Rollback Attack. Disabling just *O*1 and *O*2, allows an adversary of weaker capability (A1, compared to A3) to execute a specific kind of rollback attack.

Threat Model. External adversary (A1).

Vulnerability. To verify new targets metadata, the ECU checks the signatures, version number, expiration time, and ECU IDs.

If an adversary replays an older targets metadata file, but the older targets metadata file does not have a lower version number (due to *O*1), then the version number check will pass. The metadata's signatures and ECU identifiers will still be valid. For the expiration time check, messages containing the latest time can be dropped, or garbage time messages can be injected, as discussed for freeze attacks. This could be performed in the background by the adversary (keeping the ECU's clock at the same value for multiple update cycles) until they spot a vulnerability in one of the ECU's images and are ready to execute the rollback. During image verification, the ECU checks for rollbacks by comparing release counters in current and previous targets metadata files. However, this check is not performed if these version numbers are absent (due to the second omission), so the rollback succeeds. An automaker may omit these features if they view two images as essentially equivalent and want to allow an ECU to freely switch between them. This is not intuitively a "rollback," but the attack demonstrates how important it is to reason about combinations of protocol features.

Detection. Based on output from Tamarin, we insert a replay attacker on the connection from the director repository to the primary and from the primary to the secondary. MC finds a counterexample where the adversary replays a previous targets metadata file with the same version number as the current targets metadata file. (The older targets metadata file contains information about an older image.) The metadata file passes verification. When the primary ECU requests the corresponding image from the image repository, the adversary replays the older image. Image verification passes because the ECU cannot detect that the image is older (because the image's version number is absent from the targets metadata), and the image matches the ECU's current metadata.

8 RELATED WORK

There are a few formal analyses of Uptane in the literature, including one [9] which was discussed in depth in Section 3.

Additionally, there is an approach based on Communicating Sequential Processes [37, 38] and an approach based on attack-defense trees [48]. However, these studies have the following limitations: (*i*) they [37, 38, 48] don't consider attacks outside of those addressed in [41]; (*ii*) they [37, 38, 48] don't consider adversaries operating within the vehicle; (*iii*) they [37, 38] don't consider a system with multiple secondary ECUs; (*iv*) they [37, 38, 48] don't consider the omission of optional protocol features; (*v*) they [48] only consider a threat model where the adversary is very powerful (always assuming *all* repository keys are compromised); and (*vi*) their [37, 38, 48] models did not lead to the discovery of new vulnerabilities. We overcome these limitations and construct a rich model that accounts for multiple different adversarial scenarios and produces counterexamples representing new vulnerabilities.

Previous works with the most similar methods are LTEInspector [32] and 5GReasoner [33], which combine model checkers with CPVs to analyze 4G and 5G cellular protocols. Our approach builds on these methods in several ways, as discussed in Section 5.

There is also previous work related to OTA updates in general—Uptane is built on top of a general framework called The Update Framework, which is discussed and analyzed in [11, 12, 42, 43, 59]. In addition, many works analyze OTA updates in the context of IoT systems [5, 22, 31, 53], which have similar limitations as automotive systems (*e.g.*, devices having limited computing power). Finally, some works analyze OTA updates in vehicles [3, 15, 34, 36, 56, 57], but consider approaches separate from the Uptane protocol.

9 DISCUSSION

Responsible Disclosure. We reported our findings to the Uptane standards body, and the legitimacy of our results has been positively acknowledged. To address the issues discussed in this paper, we are collaborating with the Uptane standards body, including working on updates to the core specification documents [27, 28]. For example, since reporting our findings, both VVM replay attacks (see Appendix E) have been addressed in the newest version of Uptane (version 2.1), and there is an active GitHub issue addressing the freeze attack. Further updates are currently in progress.

Performance. Our Uptane system model consists of around 2K lines of specs and eight functional properties. To prove properties and to find counterexamples for disproven properties in Table 5, MC took 7 minutes and 49 seconds and 2 minutes and 33 seconds on average, respectively. Out of 20 instances where no attack was found, MC was able to prove the absence of an attack in 14 instances. In the other 6 instances, MC timed out. (Performance metrics were gathered using an Apple M2 CPU and 16 GB RAM.)

Feasibility of Threat Models. Threat models A3 through A* rely on various *key compromises*, which are strong assumptions. However, the Uptane standard and deployment recommendations state that director repository keys ought to be kept *online* to automate creation of fresh metadata. In contrast, image repository keys ought to be kept *offline*, as the corresponding metadata is updated less frequently. Note that *all* of our attacks only require compromising *online director* keys, which are much more vulnerable.

Generalizability of Insights. While **Insight 1** (compression) is Uptane-specific, all five other insights (comprising the automated verification workflow, including eager combination with replay attackers and blame assignment) are *protocol-agnostic*.

Testbed. To the best of our knowledge, there are no *complete* and *freely available* implementations of Uptane at the time of writing this paper. For example, the reference implementation [44] is no longer mentioned on Uptane website, as it is based on an obsolete version of the standard. Also, the implementation in AGL omits core aspects of the standard, including parts of metadata verification. While experimentation in a testbed would be ideal, we argue that specification-level issues and ambiguities can lead to vulnerable implementations and should be resolved at the specification level.

Threat to Validity. We put forth a good-faith attempt to model the standard to the best of our ability. Implementations that interpret the standard differently, deviate from the standard, or provide additional security measures not prescribed by the standard may suffer from a different set of vulnerabilities.

10 CONCLUSION

We present a novel workflow for the formal analysis of security protocols and apply it to Uptane, a state-of-the-art OTA update protocol. Our strategy leverages the strengths of model checkers and CPVs, and its application to Uptane reveals five known and six new security vulnerabilities. Since previous manual efforts failed to document these vulnerabilities, we argue that automated reasoning is a uniquely comprehensive and rigorous tool for protocol analysis.

ACKNOWLEDGMENTS

This work was supported in part by the State University of New York's Empire Innovation Program.

REFERENCES

[1] Airbiquity. 2024. Airbiquity. https://www.airbiquity.com/. Accessed: March 25, 2023.

[2] Alessandro Armando and Luca Compagna. 2008. SAT-based model-checking for security protocols analysis. *International Journal of Information Security* 7 (2008), 3–32.

[3] N. Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. 2018. ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2290–2300. https://doi.org/10.1109/TCAD.2018.2858422

[4] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2012. Easycrypt: A tutorial. *International School on Foundations of Security Analysis and Design* (2012), 146–166.

[5] Jan Bauwens, Peter Ruckebusch, Spilios Giannoulis, Ingrid Moerman, and Eli De Poorter. 2020. Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles. *IEEE Communications Magazine* 58, 2 (2020), 35–41. https://doi.org/10.1109/MCOM.001.1900125

[6] Rohit Bhatia, Vireshwar Kumar, Khaled Serag, Z. Berkay Celik, Mathias Payer, and Dongyan Xu. 2021. Evading Voltage-Based Intrusion Detection on Automotive CAN. In *Network and Distributed System Security Symposium.*

[7] Bruno Blanchet. 2007. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, Vol. 117. 156.

[8] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Found. Trends Priv. Secur.* 1, 1–2 (Oct. 2016), 1–135. https://doi.org/10.1561/3300000004

[9] Ioana Boureanu. 2023. Formally Verifying the Security and Privacy of an Adopted Standard for Software-Update in Cars: Verifying Uptane 2.0. In *IEEE SMC 2023*.

[10] Mehmet Bozdal, Mohammad Samie, and Ian Jennions. 2018. A survey on can bus protocol: Attacks, challenges, and potential solutions. In *2018 International Conference on Computing, Electronics and Communications Engineering (iCCECE)*. IEEE, 201–205.

[11] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. 2008. A Look in the Mirror: Attacks on Package Managers. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. Association for Computing Machinery, New York, NY, USA, 565–574. https://doi.org/10.1145/1455770.1455841

[12] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. 2008. Package management security. *University of Arizona Technical Report* (2008), 08–02.

[13] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 Model Checker. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 510–517. https://doi.org/10.1007/978-3-319-41540-6_29

[14] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *20th USENIX Security Symposium (USENIX Security 11)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/usenix-security-11/comprehensive-experimental-analyses-automotive-attack-surfaces

[15] Thomas Chowdhury, Eric Lesiuta, Kerianne Rikley, Chung-Wei Lin, Eunsuk Kang, BaekGyu Kim, Shinichi Shiraishi, Mark Lawford, and Alan Wassyng. 2018. Safe and Secure Automotive Over-the-Air Updates. In *Computer Safety, Reliability, and Security*, Barbara Gallina, Amund Skavhaug, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 172–187.

[16] Catalin Cimpanu. 2017. Petya Ransomware Outbreak Originated in Ukraine via Tainted Accounting Software. Available at https://www.bleepingcomputer.com/news/security/petya-ransomware-outbreak-originated-in-ukraine-via-tainted-accounting-software/.

[17] Linday Clark. 2023. CAN do attitude: How thieves steal cars using network bus. https://www.theregister.com/2023/04/06/can_injection_attack_car_theft. Accessed: March 25, 2023.

[18] Gianpiero Costantino and Ilaria Matteucci. 2020. KOFFEE-Kia OFFensivE Exploit. *Istituto di Informatica e Telematica, Tech. Rep.* (2020).

[19] Sam Curry. 2023. Web Hackers vs. The Auto Industry. https://samcurry.net/web-hackers-vs-the-auto-industry/. Accessed: March 25, 2023.

[20] Debian. 2003. Debian Investigation Report after Server Compromises. https://www.debian.org/News/2003/20031202. Accessed: March 25, 2023.

[21] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

[22] Saad El Jaouhari and Eric Bouvet. 2022. Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions. *Internet of Things* 18 (2022), 100508. https://doi.org/10.1016/j.iot.2022.100508

[23] Zeinab El-Rewini, Karthikeyan Sadatsharan, Daisy Flora Selvaraj, Siby Jose Plathottam, and Prakash Ranganathan. 2020. Cybersecurity challenges in vehicular communications. *Vehicular Communications* 23 (2020), 100214. https://doi.org/10.1016/j.vehcom.2019.100214

[24] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. 2015. Fast and Vulnerable: A Story of Telematic Failures. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot15/workshop-program/presentation/foster

[25] Linux Foundation. 2024. Automotive Grade Linux. https://www.automotivelinux.org/. Accessed: Mar 15, 2023.

[26] Linux Foundation. 2024. Uptane – Securing Software Updates for Automobiles. https://uptane.github.io/. Accessed: Mar 15, 2023.

[27] Linux Foundation. 2024. Uptane Deployment Best Practices v.2.1.0. https://uptane.org/docs/2.1.0/deployment/best-practices. Accessed: Mar 15, 2023.

[28] Linux Foundation. 2024. Uptane Standard for Design and Implementation 2.1.0. https://uptane.org/docs/2.1.0/standard/uptane-standard. Accessed: Mar 15, 2023.

[29] P. W Frields. 2008. Infrastructure report, 2008-08-22 UTC 1200. https://listman.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html. Accessed: March 25, 2023.

[30] Inc. GitHub. 2012. Public Key Security Vulnerability and Mitigation. https://github.blog/2012-03-04-public-key-security-vulnerability-and-mitigation/. Accessed: March 25, 2023.

[31] Xinchi He, Sarra Alqahtani, Rose Gamble, and Mauricio Papa. 2019. Securing Over-The-Air IoT Firmware Updates Using Blockchain. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems* (Crete, Greece) *(COINS '19)*. Association for Computing Machinery, New York, NY, USA, 164–171. https://doi.org/10.1145/3312614.3312649

[32] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. 2018. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2018.23313 tex.ids: hussainLTEInspectorSystematicApproach2018a.

[33] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 2019. 5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 669–684. https://doi.org/10.1145/3319535.3354263

[34] Muhammad Sabir Idrees, Hendrik Schweppe, Yves Roudier, Marko Wolf, Dirk Scheuermann, and Olaf Henniger. 2011. Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates. In *Communication Technologies for Vehicles*, Thomas Strang, Andreas Festag, Alexey Vinel, Rashid Mehmood, Cristina Rico Garcia, and Matthias Röckl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 224–238.

[35] Swati Khandelwal. 2018. CCleaner Attack Timeline—Here's How Hackers Infected 2.3 Million PCs. Available at https://thehackernews.com/2018/04/ccleaner-malware-attack.html.

[36] Muzaffar Khurram, Hemanth Kumar, Adi Chandak, Varun Sarwade, Nitu Arora, and Tony Quach. 2016. Enhancing connected car adoption: Security and over the air update framework. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. 194–198. https://doi.org/10.1109/WF-IoT.2016.7845430

[37] Rhys Kirk, Hoang Nga Nguyen, Jeremy Bryans, Siraj Shaikh, David Evans, and David Price. 2021. Formalising UPTANE in CSP for Security Testing. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 816–824. https://doi.org/10.1109/QRS-C55045.2021.00124

[38] Rhys Kirk, Hoang Nga Nguyen, Jeremy Bryans, Siraj Ahmed Shaikh, and Charles Wartnaby. 2023. A formal framework for security testing of automotive over-the-air update systems. *Journal of Logical and Algebraic Methods in Programming* 130 (2023), 100812. https://doi.org/10.1016/j.jlamp.2022.100812

[39] Sebastian Köhler, Richard Baker, Martin Strohmeier, and Ivan Martinovic. 2022. Brokenwire: Wireless disruption of ccs electric vehicle charging. *arXiv preprint arXiv:2202.02104* (2022).

[40] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*. 447–462. https://doi.org/10.1109/SP.2010.34

[41] Trishank Karthik Kuppusamy, Lois Anne DeLong, and Justin Cappos. 2018. Uptane: Security and Customizability of Software Updates for Vehicles. *IEEE Vehicular Technology Magazine* 13, 1 (2018), 66–73. https://doi.org/10.1109/MVT.2017.2778751

[42] Trishank Karthik Kuppusamy, Vladimir Diaz, and Justin Cappos. 2017. Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 673–688. https://www.usenix.org/conference/atc17/technical-sessions/presentation/kuppusamy

[43] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. 2016. Diplomat: Using Delegations to Protect Community Repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 567–581. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/kuppusamy

[44] NYU Secure Systems Lab. 2023. Uptane Reference Implementation. https://github.com/uptane/obsolete-reference-implementation. Accessed: March 25, 2023.

[45] Daniel Larraz, Mickaël Laurent, and Cesare Tinelli. 2021. Merit and blame assignment with Kind 2. In *Formal Methods for Industrial Critical Systems: 26th International Conference, FMICS 2021, Paris, France, August 24–26, 2021, Proceedings 26*. Springer, 212–220.

[46] Daniel Larraz and Cesare Tinelli. 2023. Finding Locally Smallest Cut Sets Using Max-SMT. *ACM SIGAda Ada Letters* 42, 2 (Apr 2023), 32–39. https://doi.org/10.1145/3591335.3591337

[47] Robert Lorch, Daniel Larraz, Cesare Tinelli, and Omar Chowdhury. 2024. A Comprehensive, Automated Security Analysis of the Uptane Automotive Over-the-Air Update Framework: Technical Report. https://github.com/lorchrob/UptaneRaid2024TechReport/tree/master. Accessed: Nov 29, 2023.

[48] Shahid Mahmood, Alexy Fouillade, Hoang Nga Nguyen, and Siraj A. Shaikh. 2020. A Model-Based Security Testing Approach for Automotive Over-The-Air Updates. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 6–13. https://doi.org/10.1109/ICSTW50294.2020.00019

[49] Farhad Manjoo. 2010. I'm Sorry, Dave, I'm Afraid I Can't Make a U-Turn. https://slate.com/technology/2010/02/should-we-be-worried-that-our-cars-are-controlled-by-software.html. Accessed: Mar 15, 2023.

[50] Anthony Martin. 2020. Vehicle Dynamics International. https://www.vehicledynamicsinternational.com/features/vehicle-cybersecurity-control-the-code-control-the-road.html. Accessed: Mar 15, 2023.
[51] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 696–701.
[52] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* 2015, S 91 (2015), 1–91.
[53] Imanol Mugarza, Jose Luis Flores, and Jose Luis Montero. 2020. Security Issues and Software Updates Management in the Industrial Internet of Things (IIoT) Era. *Sensors* 20, 24 (2020). https://doi.org/10.3390/s20247160
[54] Sen Nie, Ling Liu, and Yuefeng Du. 2017. Free-fall: Hacking tesla from wireless to can bus. *Briefing, Black Hat USA* 25 (2017), 1–16.
[55] Sen Nie, Ling Liu, Yuefeng Du, and Wenkai Zhang. 2018. Over-the-air: How we remotely compromised the gateway, BCM, and autopilot ECUs of Tesla cars. *Briefing, Black Hat USA* (2018), 1–19.
[56] Dennis K. Nilsson, Lei Sun, and Tatsuo Nakajima. 2008. A Framework for Self-Verification of Firmware Updates over the Air in Vehicle ECUs. In *2008 IEEE Globecom Workshops*. 1–5. https://doi.org/10.1109/GLOCOMW.2008.ECP.56
[57] Anam Qureshi, Murk Marvi, Jawwad Ahmed Shamsi, and Adnan Aijaz. 2022. eUF: A framework for detecting over-the-air malicious updates in autonomous vehicles. *Journal of King Saud University - Computer and Information Sciences* 34, 8, Part A (2022), 5456–5467. https://doi.org/10.1016/j.jksuci.2021.05.005
[58] Ishtiaq Rouf, Robert D Miller, Hossen A Mustafa, Travis Taylor, Sangho Oh, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. 2010. Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study.. In *USENIX Security Symposium*, Vol. 10.
[59] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable Key Compromise in Software Update Systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/1866307.1866315
[60] Khaled Serag, Rohit Bhatia, Akram Faqih, Muslum Ozgur Ozmen, Vireshwar Kumar, Z. Berkay Celik, and Dongyan Xu. 2023. ZBCAN: A Zero-Byte CAN Defense System.
[61] Khaled Serag, Rohit Bhatia, Vireshwar Kumar, Z Berkay Celik, and Dongyan Xu. 2021. Exposing New Vulnerabilities of Error Handling Mechanism in CAN.. In *USENIX Security Symposium*. 4241–4258.
[62] SolarWinds. 2019. SolarWinds Security Advisory. Available at https://www.solarwinds.com/securityadvisory.
[63] Junko Takahashi, Yosuke Aragane, Toshiyuki Miyazawa, Hitoshi Fuji, Hirofumi Yamashita, Keita Hayakawa, Shintarou Ukai, and Hiroshi Hayakawa. 2017. Automotive attacks and countermeasures on lin-bus. *Journal of Information Processing* 25 (2017), 220–228.
[64] Apache Infrastructure Team. 2009. apache.org incident report for 8/28/2009. https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html. Accessed: March 25, 2023.
[65] Ken Tindell. 2023. CAN Injection: keyless car theft. https://kentindell.github.io/2023/04/03/can-injection/. Accessed: March 25, 2023.
[66] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. 2020. Plug-N-Pwned: Comprehensive Vulnerability Analysis of OBD-II Dongles as A New Over-the-Air Attack Surface in Automotive IoT. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 949–965. https://www.usenix.org/conference/usenixsecurity20/presentation/wen
[67] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. 2014. A practical wireless attack on the connected car and security protocol for in-vehicle CAN. *IEEE Transactions on intelligent transportation systems* 16, 2 (2014), 993–1006.

A ABBREVIATIONS

Table 7 contains the abbreviations we use throughout the paper and their meanings.

Abbreviation	Meaning
MC	model checker
CPV	cryptographic protocol verifier
$\mathcal{S}$	Uptane model
$\widehat{\mathcal{S}}$	compressed Uptane model
$\mathcal{S}^+$	adversarially instrumented Uptane model
$\widehat{\mathcal{S}}^+$	adversarially instrumented, compressed Uptane model
Properties	
P1	checks for freeze
P2	checks for VVM replay
P3	checks for rollback
P4	checks for arbitrary software
P5	checks for attacker-authored VVM
P6	checks for mix-and-match
P7	checks for mixed-bundles
P8	checks for incompatible image
Threat models	
B	benign
A1	outside vehicle MitM
A2	outside + inside vehicle MitM
A3	Full MitM + director keys
A4	A3 + timestamp & snapshot image repo keys
A5	A4 + targets image repo keys
A6	A5 + root image repo keys
A*	A1 + primary ECU key
A#	supply chain attack

Table 7: Abbreviations and their Meanings

B UPTANE BACKGROUND

B.1 Uptane Metadata

Uptane uses four types of metadata: *root*, *timestamp*, *snapshot*, and *targets* metadata, each requiring a threshold of digital signatures. Root metadata specifies the public keys associated with each metadata type (root of trust). Targets metadata includes information about images to be installed (image filenames, hashes, etc.). Snapshot metadata contains a filename and version number for each targets metadata file currently on the repository (to guarantee consistency). Timestamp metadata includes the filename and version number of the latest snapshot metadata file, as well as its hash (to guarantee freshness). Every metadata file has a version number and expiration timestamp. At construction time, the vehicle is manually initialized with a set of valid metadata.

Update discovery: In the *update discovery* phase, new images are selected for the ECUs to install. This selection is performed by the director repository, which retrieves information about the vehicle's currently installed information and performs dependency resolution to select new updates.

1) The primary ECU queries the director repository for new updates. This query includes information about the vehicle's currently installed images in a file called the *vehicle version manifest* (VVM), signed with the primary ECU's secret key.
2) The director repository verifies the VVM by checking digital signatures, ECU IDs, and nonces.
3) The director repository generates a fresh set of metadata on the new updates to be installed and sends it to the primary ECU.

B.2 Protocol Steps

Metadata verification: In the *metadata verification* stage, ECUs process metadata from both remote repositories, applying verification procedures to ensure that the metadata is not tampered with.

4) The primary ECU verifies each metadata file from the director repository. If verification succeeds, the primary ECU overwrites its current set of metadata. We will not discuss the specifics of metadata verification; for more information see [28, 41].
5) The primary ECU queries the image repository for fresh metadata and performs a similar verification.
6) The primary ECU cross-references the targets metadata from the two repositories to check for discrepancies.
7) If the verification succeeds, the primary ECU sends the new metadata to all secondary ECUs, which then perform their own metadata verification.

Image verification and acquisition: In the *image verification and acquisition* stage, ECUs apply a verification procedure to the new images retrieved from the image repository. If this verification is successful, then the ECU installs the new image.

8) The primary ECU downloads new images from the image repository which correspond to those listed in its latest set of metadata.
9) The primary ECU verifies the images by comparing the images' hashes to those specified in the targets metadata.
10) If verification succeeds, the primary installs its new image and forwards the other new images to the secondary ECUs, which also perform verification and installation.
11) Each secondary ECU reports back to the primary whether or not the updates were successfully installed in an *ECU version report*. Each report is signed with the corresponding secondary ECU's secret key and is included in the VVM at the beginning of the next update cycle (update discovery).

If any of the steps above fails, the offending file (metadata, image, or VVM) is discarded and the cycle is restarted.

B.3 Alice-Bob Description of Uptane

We give a high-level description of the operations performed and the messages sent in a successful Uptane update cycle using Alice-Bob notation (see Figure 8). The protocol steps are numerically labeled to illustrate which step of the three stages (outlined previously) they correspond to. We include four principles, P (primary ECU), S (secondary ECU), D (director repository), and I (image repository). We denote keys in the form $skey_{dr}$, where (for example) *skey* denotes a secret key, d denotes the Director principal, and r denotes the root role. Local operations/checks are encoded by the *check* keyword. The Alice-Bob notation assumes a "happy path" where all checks succeed. For simplicity of presentation, we assume that the threshold number of signatures required for each metadata file is one and that there is a single secondary ECU. Further, we leave some of the details implicit: (i) that all signatures are verified according to the latest root metadata, and (ii) that each ECU checks that each incoming metadata file has a nondecreasing version number and is not expired.

C PROPERTY FORMULATION

P1.
```
always (
  Event^c => targets_meta_secondary
             = director_latest_targets
)
```

Attack Type	Our Approach	Approach from[9]
Number of threat models	9	5
Number of security properties	8	7
Meta-level analysis of optional protocol features with blame assignment?	Yes	No
Compression?	Yes	No
Eager combination with replay attacker?	Yes	No

Table 8: Comparison of approach with [9].

P2.
```
always (
  Event^c => director_latest_manifest
             = primary_latest_manifest
)
```

P3.
```
always (
  Event^c => image_secondary_version
             >= old_image_secondary_version
)
```

P4.
```
always (
  Event^c => AuthoredByOem(image_secondary)
)
```

P5.
```
always (
  Event^c =>
  AuthoredByEcu(director_latest_manifest)
)
```

P6.
```
always (
  Event^c => Compatible(targets.image_one,
             targets.image_two,
             targets.image_three)
)
```

P7.
```
always (
  Event^c =>
  (Historically(pri_soft_version
                = sec_soft_version) =>
   Compatible(pri_image, sec_image, sec2_image)
  )
)
```

P8.
```
always (
  Event^c =>
  CompatibleHardware(image_sec, hardware_sec)
)
```

D IMPLEMENTATION

In this section, we describe the details of our approach.

D.1 Modeling Adversarial Influence

To model adversarial influence, we place an adversarial component A between each pair of components (M, N) in $\mathcal{S}$. When M sends a message to N (or vice versa), A takes in M's output and nondeterministically modifies it before passing it along to N, acting as a man in the middle.

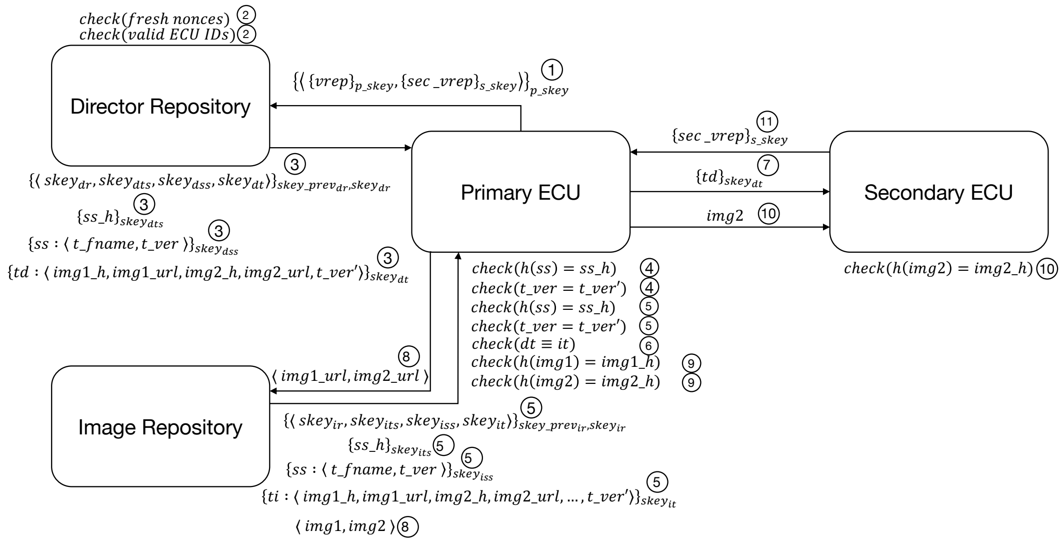

Figure 8: Uptane Alice-Bob Description

The adversary's behavior is defined as an abstract description of its output values in terms of its input values, called a *contract*. This allows for nondeterminism, where the adversary is free to pick any possible modification that respects the contract.

Standard Attacker. No restrictions are imposed on the output value of the component, so the adversary can select an arbitrary message to inject.

Bounded Replay Attacker. The bounded replay attacker can only inject messages that were sent along the network connection in the last k timesteps (for some constant k). The adversary's contract involves storing each message sent from M to N in a wraparound queue of size k. This wraparound queue can be seen as a sliding window where the attacker only holds the last k messages in its memory. Then, the attacker must pick an element from this window when injecting messages along the network channel. This disallows the adversary from *arbitrarily* modifying the data sent from A to B, as the adversary can now only replay one of the last k messages.

Unbounded Replay Attacker. We also model a replay attacker with unbounded memory, *i.e.*, the ability to replay messages that were sent arbitrarily far in the past. Instead of storing previous messages in a queue, the unbounded replay attacker stores the values of previous messages as outputs of a partial function $f : \mathbb{N} \rightharpoonup B$, where B is the type of the message being sent along the channel. At each timestep i, the adversary extends the definition of f such that $i \mapsto \text{msg}_i$, where msg_i is the current message being sent along the channel. Then, at timestep i, the adversary is free to nondeterministically select any message $m \in \{f(j) \mid 0 \leq j \leq i\}$.

D.2 MC Model

Uptane System Architecture. We begin by outlining Uptane's system architecture which is graphically illustrated in Figure 2. We define the top level component and each of its subcomponents and then specify each component's interface and connections. For example, here is the `SecondaryECU` component's interface:

```
component SecondaryECU
  (in_primary: PrimaryToSecondary)
returns
  (out_primary: SecondaryToPrimary,
   installed_image_secondary: Image,
   verified_metadata_secondary:Metadata)
```

The `SecondaryECU` component has one input where it receives data from the primary ECU and three outputs where it (i) sends data to the primary ECU, (ii) reports its currently installed image, and (iii) reports its current set of verified metadata. The `in_primary` and `out_primary` variables contain the data being sent from the primary to the secondary, and vice versa (*e.g.* metadata, ECU version reports). The `installed_image_secondary` and `verified_metadata_secondary` variables are also records, the former containing information about an image and the latter containing four metadata files.

Component-Level Design. Next, we specify the behavior of each component of the architecture with assume-guarantee contracts. Consider the following functional specifications for the `SecondaryECU`: (1) The ECU's version report has correct information with respect to the currently installed image (filename and hashes); and (2) the ECU updates metadata according to the Uptane standards document.

We formalize these specifications using abstract syntax for conciseness. First, for (1), we add the following guarantee stating that the filename and hash listed in the output ECU version report equal the installed image's filename and hash (`image` abbreviates `installed_image_secondary`):

```
image.filename = out_primary.report.filename and
hash(image) = out_primary.report.hash
```

To formalize aspect (2), we add the following guarantee where `image` abbreviates `installed_image_secondary`:

```
if new_image_verified
then image' = new_image
else image' = image
```

where `image'` denotes the value of variable `image` in the next state, and `new_image_verified` is a predicate defined according to the image verification instructions in the Uptane standards document. The guarantee states that if the new image passes verification, the ECU updates to the new image; otherwise, it keeps the previously-installed image. In the above example, the initial state predicate enforces that `image` is equal to `initial_image` initially.

Any system execution that satisfies these constraints will be considered valid during the analysis performed by MC. We follow a similar process to specify the other components.

Desired Functional Requirements. Recall the list of desired functional requirements from Section 3. We can express, for example, property P3 (checks for rollback attacks) with the following top-level guarantee, where the primed version of `installed_image_secondary` refers to the value of the variable in the next state:

```
installed_image_secondary'.version >=
installed_image_secondary.version
```

Simplifying Assumptions. To make the analysis feasible, we make further simplifying assumptions: (*i*) We assume there are no delegations[1] in targets metadata; (*ii*) we model a vehicle with exactly one primary and two secondary ECUs; and (*iii*) we omit parts of the standard that relate to implementation details, *e.g.*, how filenames are encoded.

D.3 CPV Model

To accurately model the adversary's capabilities with respect to cryptographic assumptions, we consult Tamarin [51], a cryptographic protocol verifier. We formulate a Tamarin model by including rules capturing the cryptographic aspects of each message sent in the Uptane protocol. As an example, the following rule models the primary ECU sending targets metadata to the secondary ECU:

```
rule primary_send_metadata:
  [!PriLatestTargetsDir(targets, <sig1, sig2>)]
  -->
  [Out(< targets, sig1, sig2 >)]
```

The primary ECU retrieves the most recently verified targets metadata (and its signatures) and sends it along the insecure network channel. The secondary ECU verifies metadata with the following rule:

```
rule secondary_verify_metadata:
  [ In(<targets, sig1, sig2>),
    SecMetaDir(<pubkey1, pubkey2>, old_targets) ]
  --[ Eq(verify(sig1, targets, pubkey1), true),
      Eq(verify(sig2, targets, pubkey2), true), ]->
  [SecMetaDir(<pubkey1, pubkey2>, targets)]
```

The secondary ECU receives the incoming metadata from the insecure channel and performs verification by checking the digital signatures. The Tamarin model only captures the *cryptographic aspects* of the protocol while abstracting away other aspects (*e.g.*, checking version numbers).

In the Tamarin model, we formulate 10 correspondence (weak authentication) and 10 injective-correspondence (strong authentication) lemmas to learn about the cryptographic aspects of the protocol. The correspondence lemmas are of the form:

$$\forall msg. \forall i.\ \text{Receive}(msg)@i \Rightarrow (\exists j.\ \text{Send}(msg)@j \wedge (i < j).$$

In other words, if a message (*e.g.*, metadata) was received (*i.e.*, verified) at time i, then it must have been sent at an earlier time j. If this lemma is proven for some network connection, then it is cryptographically infeasible for the adversary to inject new messages.

The injective-correspondence lemmas are of the form:

$$\forall m. \forall i.\ \text{Receive}(m)@i \Rightarrow (\exists j.\ \text{Send}(m)@j \wedge (j < i) \\ \wedge \neg(\exists i2.\ \text{Receive}(m)@i2 \wedge \neg(i2 = i))).$$

This lemma states that if a message was received at time i, then it must have a *unique* matching sender at an earlier time j. If this lemma is proven, then the adversary cannot inject *or* replay messages; failure suggests that replays are possible. The two previous lemmas are formulated for each step of metadata verification, as well as for VVM and image verification. In addition to the authentication lemmas, it is necessary to prove *sanity-check* lemmas of the form $\exists msg. \exists i.\ \text{Receive}(msg)@i.$ that demonstrate that the premises of the implications are reachable. These lemmas are called a "sanity-check" lemma because if it is disproven by Tamarin, then there is likely a mistake in the model. We proved sanity-check lemmas for every protocol message.

Termination. Tamarin often has to find long traces to (dis)prove lemmas, resulting in termination issues. We thus implemented our own heuristic (called an *oracle*) to guide Tamarin in its proof search. This heuristic orders Tamarin's proof goals, instructing Tamarin on which unsolved premises to solve first. Our heuristic is Uptane-specific and sets proof goals in the logical order that metadata verification occurs in.

Another essential step in achieving termination is to impose bounds on the number of times each Tamarin rule can be applied (in a trace). In some cases, for tractability, we only allow each Tamarin rule to apply only once or twice in a trace. Although this is limiting, Tamarin was still able to disprove several lemmas, leading to the attacks we found.

E FINDINGS

E.1 Working Example, Finding 2 and Finding 3 (VVM Replays)

To further illustrate the workflow, we will walk through the analysis of desired functional requirement P2. In English, the property we want to check is "The director repository never verifies old VVMs." In the attack, the adversary intercepts and replays an old VVM, tampering with the director repository's task of generating new metadata (potentially causing it to direct the vehicle to install the wrong images).

Execute CPV. First, we execute the relevant lemmas in the Tamarin model— those corresponding to (*i*) sending the VVM from the primary ECU to the director, and (*ii*) sending an ECU version report from the secondary to the primary. In both cases, the weak authentication lemma is proven, meaning that injection of new VVMs/version reports is cryptographically infeasible. Also, in both cases, the strong authentication lemma fails, producing counterexample traces where the adversary replays the same VVM/ECU version report twice, and yet they are still verified by the director.

[1]Delegations refer to when signing ability is deferred to another party. See the Uptane standard [28] for more details. To the best of our knowledge, support for delegations is not yet present in open source implementations.

Specify MC threat model. The results from the CPV model (weak authentication lemma proven, strong authentication lemma disproven) mean that a *replay attacker* should be placed on the connections from the primary ECU to the director and from the secondary ECU to the primary ECU. All other connections are labeled as invulnerable to attack.
Execute MC. MC produces a counterexample where an adversary intercepts a VVM and replays it at a later timestep, a straightforward replay that represents a violation of P6.

When the director receives the VVM, it checks that the VVM is valid by verifying the digital signatures, comparing the ECU IDs to its internal database, and checking if the nonces are fresh (based on if they have been seen before). However, the nonce check is not effective against an adversary who intercepts a VVM and replays it at a later time. The check still succeeds, as this is still the first time the director has seen the nonce (assuming the adversary does not replay the same VVM multiple times). Since the replayed VVM still contains valid signatures and ECU IDs, the replay succeeds.
Attack chaining. We perform a technique called *attack chaining* where we update the model to block the previous attack trace and see if the tool can find a different (and potentially more interesting) attack. We update the model by (i) updating the system-level property to “The director never verifies an old ECU version report,” a more fine-grained property, and by (ii) modeling the connection from the primary ECU to the director as invulnerable to attack. These changes guarantee a trace that is different than the previous attack. With these updates to the model, MC produces a counterexample trace representing a unique attack. In this version, the adversary dropped an ECU version report being sent from a secondary to the primary ECU and injected it at a later cycle. In this version of the attack, the adversary created a VVM from ECU version reports that weren’t initially supposed to be in the same VVM.

When the primary builds the VVM at the beginning of each update cycle, it queries all its secondaries for their latest ECU version reports. However, the primary does not do any verification (all verification is performed by the director). The director is thus still vulnerable to replay attacks. Instead of replaying at the manifest level, the adversary can intercept and replay individual version reports within the vehicle. When the ECU version reports are replayed, the primary still signs the manifest, so all the signatures are still valid.

E.2 Other Findings

Finding 4: Arbitrary Software Installation Attack. The adversary causes a secondary ECU to install an adversary-authored software image.
Threat Model. Internal adversary with compromised director targets metadata keys (A3).
Vulnerability. When the secondary ECU performs partial verification on metadata, there is no cross-referencing of metadata from the image repository. The compromise of only director targets metadata keys (no image repository keys) leaves the secondary ECU vulnerable to any attack. If the adversary performed a man-in-the-middle attack solely outside the vehicle, the primary would detect the attack (when cross-referencing metadata from both repositories) and would refrain from forwarding the offending targets metadata file to any secondary ECU. But, an internal adversary can directly inject a malicious metadata file, bypassing the primary ECU’s verification.
Detection. Based on Tamarin’s output, we insert a standard attacker on the connection from the primary to the secondary. With the standard attacker, MC finds a counterexample to the property “The secondary ECU never verifies an attacker-authored image.” In the counterexample, the adversary first forges and injects a targets metadata file to the secondary ECU that contains the hash of an attacker-authored image. The forged metadata file is verified by the ECU because the digital signatures are valid and the adversary has complete control over the metadata file’s contents. Then, the attacker injects the corresponding attacker-authored image, which passes the secondary’s verification because its hash matches the hash found in the secondary’s (forged) targets metadata.
Finding 5: Incompatible Image Installation Attack. The adversary causes an ECU to install an image that is incompatible with the ECU’s hardware. This is different from the incompatibilities discussed in previous attacks, which were in the form of images being incompatible with *each other*, not with the ECU’s hardware.
Threat Model. Internal adversary with compromised director targets metadata keys (A3).
Vulnerability. Same as the arbitrary software attack.
Detection. Similar to arbitrary software attack.
Finding 6: Mix-and-Match, Rollback Attacks. We also analyzed mix-and-match attacks and rollback attacks. These attacks are possible in attack scenario A3 due to the same vulnerability that allows arbitrary software attacks. But, they have a lower impact than arbitrary software, so they will not be discussed in depth.
Finding OPT2: Attacker-authored VVM. Disabling O3 allows an attack in threat model A1 where the adversary injects a VVM that is verified by the director because the digital signature is not checked. Including O3, this attack also succeeds in threat model A*.
Finding OPT3: Supply chain attacks. We model supply chain attacks by placing a standard attacker on the system-level inputs (see Figure 2 attacker A#) and annotating every other connection as invulnerable to attack. With the supply chain attacker, all desired functional requirements are violated except for P2 (checks for VVM replays), which is maintained because there is no adversary between the primary ECU and director repository to replay VVMs. Supply chain attacks are out of scope for Uptane, but this analysis serves as a sanity check and speaks to the importance of supply chain security.