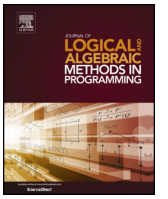
Journal of Logical and Algebraic Methods in Programming 130 (2023) 100812

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



A formal framework for security testing of automotive over-the-air update systems

Rhys Kirk [a,*], Hoang Nga Nguyen [b,*], Jeremy Bryans [a,*], Siraj Ahmed Shaikh [b], Charles Wartnaby [c]

[a] Centre for Future Transport and Cities, Coventry University, UK
[b] Systems Security Group, Department of Computer Science, Swansea University, UK
[c] Applus IDIADA, UK

ARTICLE INFO

Article history:
Received 30 March 2022
Received in revised form 19 August 2022
Accepted 14 September 2022
Available online 19 September 2022

Keywords:
Automotive cybersecurity
Security testing
Automotive OTA
Uptane
CSP

ABSTRACT

Modern vehicles are comparable to desktop computers due to the increase in connectivity. This fact also extends to potential cyber-attacks. A solution for preventing and mitigating cyber attacks is Over-The-Air (OTA) updates. This solution has also been used for both desktops and mobile phones. The current de facto OTA security system for vehicles is Uptane, which is developed to solve the unique issues vehicles face. The Uptane system needs to have a secure method of updating; otherwise, attackers will exploit it. To this end, we have developed a comprehensive and model-based security testing approach by translating Uptane and our attack model into formal models in Communicating Sequential Processes (CSP). These are combined and verified to generate an exhaustive list of test cases to see to which attacks Uptane may be susceptible. Security testing is then conducted based on these generated test cases, on a test-bed running an implementation of Uptane. The security testing result enables us to validate the security design of Uptane and some vulnerabilities to which it is subject.

1. Introduction

Modern vehicles have become part of the Internet of Things (IoT), connecting them to their users, manufacturers and the rest of the world. Consequently, vehicle connectivity introduces the threat of cyberattacks. Many attacks have been discovered for connected vehicles. Examples include attacks on tyre pressure monitoring systems (TPMS) [1–3], ECUs, CAN bus [1,2,4–6], entertainment systems (disks, USBs, etc) [1,2,4] and many others [7]. Notoriously, the Jeep Cherokee was vulnerable to having the users' control being entirely overridden by a remote attacker [4]. This led to the largest recall of 1.4 million vehicles.

OTA updates have been considered as a promising solution for remotely patching security vulnerabilities and avoiding the use of expensive and insufficient recalls. Traditional update methods can be secured as both parties have the power and resources. However, automotive OTA update faces a main challenge where vehicles do not have such required power and resources. They have many Electronic Control Units (ECUs) that each need to be updated but aren't able to handle the necessary encryption. Because of this, alternative software update solutions need to be developed to solve this problem. Many car makers have planned to implement OTA update systems since 2018 [8]. However, an OTA update needs to be

* Corresponding authors.
E-mail addresses: Kirkr@uni.coventry.ac.uk (R. Kirk), h.n.nguyen@swansea.ac.uk (H.N. Nguyen), ac1126@coventry.ac.uk (J. Bryans).

https://doi.org/10.1016/j.jlamp.2022.100812

resilient and not become an attack vector. Otherwise, it would introduce new vulnerabilities into the vehicle. For example, attackers could compromise firmware and software updates with malicious packages and ransomware. This will compromise vehicle control, functionality and user privacy by tracking and tracing vehicle user activities. Therefore, it is vital to ensure the resilience of automotive OTA. Uptane [9] has been proposed as a secure and standard design for automotive OTA. It aims to withstand known attacks whilst delivering OTA software to vehicle ECUs. Nevertheless, there has been a limited research effort in the literature on security assessment methods and techniques for automotive OTA, and on Uptane in particular. To the best of our knowledge, the most relevant work is from [10] where Mahmood et al. define a semi-formal model-based framework to evaluate the security properties of the Uptane reference implementation. They provide an informal model of Uptane while the attacker is modelled formally by attack trees. By using attack trees, Mahmood et al. generated test cases, however, attack trees can only capture known threats and cannot deal with zero-day attacks. Furthermore, how to construct attack trees for an OTA update system is still an open question.

In this paper, we present a model-based approach by modelling Uptane and an attack model inspired by Dolev-Yao [11] in Communicating Sequential Processes (CSP). A Dolev-Yao attacker is assumed to have full control over communication, and hence, has the capabilities to violate security properties, such as confidentiality, authenticity, and integrity. CSP is a formal modelling language for describing concurrent systems. Via refinement checking on these models, we verify if there are any attacks as traces of system events which lead to a security violation. We consider each trace as a security test case that we translate into a real attack. To this end, we can identify (even zero-day) attacks within the system. Similar methods of model-based security testing have been applied to automotive security and have yielded promising results [12–14]. In [12], they modelled the vehicle network CAN bus in CSP. The model allows for the generation of how a vehicle network would behave against cyber attacks. Similarly, the ECU application code can be translated into a formal model in CSP [13]. Such a model enables security analysts to identify if any potential vulnerabilities are present in the original code exhaustively and effectively. Formal models and attack trees were also employed in [14] for proposing a systematic security evaluation approach. This approach suggests using attack trees to model attackers' behaviour. These attack trees are then formalised in CSP for test case generation. This shows how such an approach can be applied to the case of automotive Bluetooth interfaces [15]. However, its limitations include: (i) all attacks captured in an attack tree are known and (ii) there are no vehicle models to drive the test case generation process.

1.1. Contributions

Our contributions in this paper are (i) a model-based method for evaluating Uptane; (ii) a formal model of Uptane in CSP; (iii) a mutation of the Uptane model; (iv) a formal model of our attacks inspired by Dolev-Yao; (v) a method to exhaustively generate security test cases and (vi) a rigorous security evaluation of the Uptane reference implementation. Although this approach was first introduced in [16], it was limited in several aspects including: (i) The Uptane model was limited and did not include the secondary ECU. By including the secondary ECU, the model now has a full verification and a partial verification, as well as an internal network that the attacker doesn't have control over. This provides more points of attack to discover and makes the model more accurate to the specification. (ii) The attack model only included three attacks. To address this, we added two more attacks, the spoof and the freeze. Spoofing can send malicious images to the vehicle and the freeze attack allows the attacker to prevent updates. This is mentioned as defend against in the specification, but will test the model as well as allowing for model mutation. (iii) The lack of an automatic mechanism to generate test cases through refinement checks that led to the generation of only 3 test cases. To address this, a Python script was created to search for more points of attack by the same attacker within the model, allowing for better discovery of attacks across the model. (iv) In addition to this, we added a mutated version of the Uptane model. This mutated model removes the defences of the freeze attack. This lets us see if the real world Uptane implementation has followed the specification: if not, it may be vulnerable to this attack.

1.2. Structure

The structure of the paper is as follows. Section 2 outlines CSP and the attack model that provide the foundation for the proposed approach. In addition, it recalls the structure and the logic of Uptane. Section 3 introduces the methodology and its application to the Uptane system. Section 4 describes how we modelled Uptane in CSP. Section 5 introduces the attack model, the mutated model of Uptane and how FDR is used to generate an exhaustive list of test cases. Section 6 presents the experiment, its setup and how we translate the test cases into real attacks. Finally, Section 7 gives a conclusion of the paper and highlights areas of improvement.

2. Background

In this section, we review CSP and Failures-Divergences Refinement (FDR) for use in our modelling and test case generation. Furthermore, we also discuss the use of the attack model.

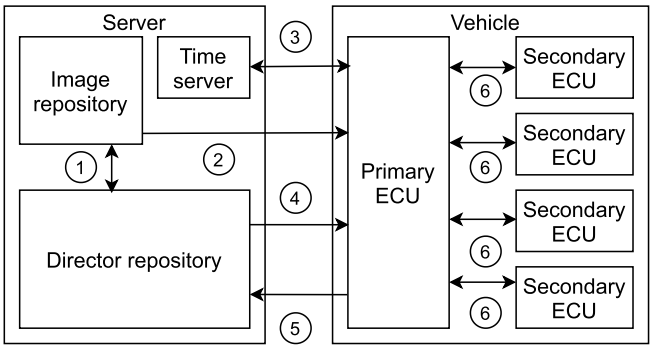


Fig. 1. The Uptane System. Sent messages are: ① and ② Image; ③ Signed token & time; ④ Metadata & image; ⑤ Vehicle Manifesto; ⑥ Metadata, images and manifesto.

2.1. CSP and FDR

We give here a brief overview of the subset of CSP that we use in this paper. A more complete introduction may be found in [17]. Given a set of events Σ, the CSP processes used in the following are defined by the following syntax:

$$P ::= Stop \mid RUN \mid e!x?y \rightarrow P \mid P_1 \Box P_2 \mid \textbf{if } b \textbf{ then } P \textbf{ else } Q \mid \Box a_i : A@P(a_i) \mid P_1 \underset{A}{\parallel} P_2 \mid P_1 \,{}_A\|_B\, P_2 \mid P(X)$$

where $A, B \subseteq \Sigma$ and $a_i \in A$. For convenience, the set of CSP processes defined via the above syntax is denoted by CSP. The term e!x?y indicates that in the communication of e x is communicated to the environment and that the variable y is instantiated by the environment. The process *Stop* is the most basic one. It does not engage in any event and represents deadlock. $RUN(A) = ?x : A \rightarrow RUN(A)$ is the process which, for a set of events $A \subseteq \Sigma$, can always communicate any member of A desired by the environment. The prefix $e \rightarrow P$ specifies a process that is only willing to engage in the event e, then behaves as P. The external choice $P_1 \Box P_2$ behaves either as P_1 or as P_2. The generalised external choice $\Box a_i : A@P(a_i)$ offers the environment any action from the set A, then behaves as $P(a_i)$. The conditional **if** b **then** P **else** Q behaves as P if b is true, Q otherwise. The generalised parallel operator $P_1 \underset{A}{\parallel} P_2$ requires P_1 and P_2 to synchronise on events in $A \cup \{\checkmark\}$, where $\checkmark$ is a special event used to mark the termination of a process. All other events are executed independently. The alphabetised parallel $P_1 \,{}_A\|_B\, P_2$ must synchronise on any event in $A \cap B$. Both processes can engage in events outside this intersection. $P(X)$ indicates recursion. There are different semantics models for CSP processes [17]. For the purpose of this paper, we recall the finite trace semantics. A trace is a possibly empty sequence of events from $\Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$. In general, the trace semantics of a process P is a subset $traces(P)$ of $(\Sigma^{\checkmark})^*$ consisting of all traces which the process may exhibit. Further details and a full definition of the traces can be found in [17]. A process P is said to trace-refine a process Q (written $Q \sqsubseteq_T P$) if $traces(P) \subseteq traces(Q)$. Failures-Divergences Refinement4 (FDR4) is a refinement checker for CSP. FDR4 can be used to troubleshoot and debug CSP scripts as well as to generate test cases from CSP models.

2.2. Attack model

The attack model employed in this paper is inspired by that of Dolev-Yao [11]. Dolev-Yao's attack model is considered to be one of the most powerful attack models [18]. It has been used widely to provide formal proofs of security protocols [19–21] and to guide security testing of system implementations [22]. In the Dolev-Yao attack model, the attacker controls the network, allowing the attacker to manipulate all packets that are sent through the network if not properly defended.

Our attack model comprises of five attacks: edit, block, listen, spoof, and freeze. The editing attack can manipulate packets sent on the network. The blocking attack blocks packets on a network. Eventually, this can cause a complete denial of service. The listening, i.e., eavesdropping, attack monitors what is being sent on the network. A spoofing attack imitates a legitimate source, so the victim thinks it is interacting with the intended recipient. The freeze attack prevents the victim from being able to update by sending it the currently installed update. This causes the victim to remain on its current update. The separation in our attack model aims at providing the nature of each generated test case. This avoids the explosion of the formal models' state space, thus facilitating the refinement checks in FDR4 and reducing time for generating test cases.

2.3. The uptane system

The structure of the Uptane [9] System is depicted in Fig. 1. The server-side comprises the director server, the image repository server, and the time server. The director checks the metadata of the system and holds the manifest. The image repository is a simplified version of the director. When the image repository receives a role request, it gets the role metadata

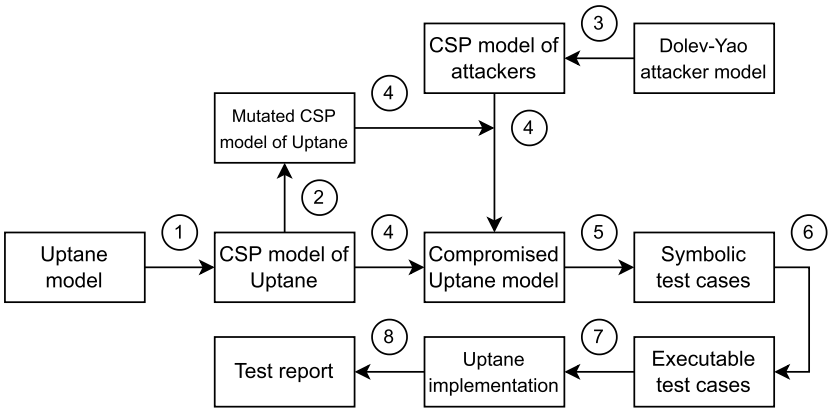


Fig. 2. Security testing methodology for Uptane: ① Formalising Uptane; ② Mutate the Uptane model; ③ Model the attacks; ④ Combining models; ⑤ Generating symbolic test cases; ⑥ Translating symbolic test cases; ⑦ Executing test cases; ⑧ Reporting the test result.

from storage and then sends it to the primary ECU. The time server holds the current time to update any ECU that does not have a clock. Inside a vehicle, i.e., the client-side, the Uptane system consists of one primary ECU and multiple secondary ECUs. The primary ECU is responsible for communicating with the server to receive the time, firmware, and metadata. The server will process the manifest sent by the primary and respond if there is a new update. Commonly, secondary ECUs have limited computing power since they are designed for specific tasks. Therefore, they have no direct communication with the server in the Uptane system. A secondary ECU communicates with the primary ECU to receive new updates. In order to verify the updates, the system utilises metadata. Metadata is given as one of four roles: root, timestamp, snapshot, and targets. All roles follow the same structure, containing role-nonspecific data (version and expiration time), a unique payload and a signature. Root's payload contains public keys for each role. Timestamp's payload contains snapshot filename and snapshot version. Snapshot's payload contains target filenames and target versions. Targets' payload contains image names and ECU IDs. Roles are tested to verify their integrity and legitimacy. The verification method takes 7 steps. If any of the steps fail, the verification system will stop and the update will not be installed. The 7 steps are: ① Load the current metadata from the storage; ② Send a request to the server (director or image repository) for role metadata; ③ Decrypt the signature in the received metadata using the private key; ④ Verify the payloads legitimacy from the decrypted signature; ⑤ Ensure that the new metadata version is higher than the old metadata version; ⑥ Check that the current time is lower than the expiration time of the new metadata; and ⑦ Install the new firmware once the verification of the metadata has been confirmed.

3. Methodology

In this section, we present our methodology employed for Uptane's security evaluation. First, formal models of the automotive OTA system and the attacks are created. We will then combine these models to generate test cases via refinement checks. This is a combination of formal modelling and penetration testing, creating a rigorous and more cost-effective way to identify potential vulnerabilities within a system. The methodology consists of 7 steps, seen in Fig. 2. Steps ① and ③ can be done concurrently, where all other steps must be done sequentially.

①*Formalising Uptane:* We construct a model of the Uptane system. Typically, a bottom-up approach is employed where we first model the components of the Uptane system. We combine the components to capture the behaviours of the overall system.

②*Mutate the Uptane model:* The Uptane model is altered by removing some of the security features. For example removing the time check that prevents freeze attacks as detailed in Section 5.2. These removals allow us to generate test cases that can check for the corresponding security features in an implementation of Uptane.

③*Model attacks:* Then we construct the attacks in CSP. We model the behaviour of the attacks: block, eavesdrop, freeze, spoof and edit.

④*Combining models:* In this step we combine the Uptane and attack models to represent a compromised system.[1] Typically, this is done by replacing the network model with the attack model. This reflects that the network is compromised, and the attack has full control of the network.

⑤*Generating symbolic test cases:* In this step trace refinements are used to generate test cases. However we first capture the secure behaviours where there are no attacks. We check if they are refined by Uptane's compromised models in FDR4. If it is not the case, a counterexample is generated and used as the symbolic test case.

[1] The model discussed is available at https://github.com/ssgrepos/LCA22.

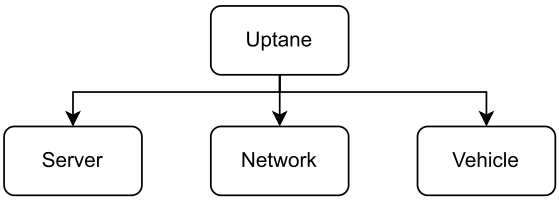


Fig. 3. Uptane model showing the connectivity between the server, network and vehicle.

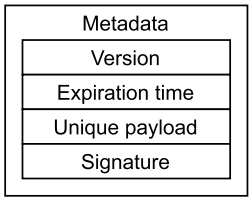


Fig. 4. Role metadata contents: firmware version, expiration time of metadata, the unique payload of the metadata and the signature.

⑥*Translating symbolic test cases:* We then translate the generated symbolic test cases into executable ones employing Ettercap [23].
⑦*Executing test cases:* The translated attacks are then tested on the Uptane implementation.
⑧*Reporting test cases:* We take the results of the attacks and formulate a test report.

4. Modelling uptane

This section describes how we use CSP to create a model of the Uptane system. Due to the complexity of the Uptane specification, the scope of the model has been limited to the: director, image repository, primary ECU, secondary ECU, the verification process of the metadata and sending the firmware and manifest. The Uptane model is broken down into three processes: a server, a wireless network, and a vehicle as seen in Fig. 3. In the following we discuss data types that form the basis of the model, the models of communication network, the vehicle including primary and secondary ECUs and the server including roles, manifest and verification systems.

In our approach, we limit the Uptane model to one update cycle. The rationale behind this is to manage the complexity of the model. Without this limitation, the running time required to generate the test cases, by adding multiple update cycles to the model, increases exponentially. The full Dolev-Yao model would allow all attacks to happen simultaneously, requiring more than one update cycle. Therefore, splitting the attack model inspired by Dolev-Yao into separate attacks fits the one-update-cycle limitation of the Uptane model. However this prevents us from catering to complex attacks that combine multiple simple attacks. We leave this for the future work.

4.1. Data types

We define the following data types for events and processes in the model: Roles (values root, timestamp, snapshot and targets); Public and private keys (labelled as Pkey and Skey, respectively); ECU ID (one primary and one secondary); File names (used for role metadata); Image names (current image, new image, and bad image for attacks, see Section 5.1); Filename Metadata Pair of a filename and a version number to capture the structure of the storage for vehicle ECUs; Components (image repository, director, primary, secondary and an attacker); Message (metadata request, a response containing the metadata).

4.2. Role metadata

The role metadata structure comprises its version, expiration time, payload and signature, this can be seen in Fig. 4. Version and expiration time ranges from 1 to 3 to represent past, present and future. The role metadata unique payloads are modelled by sequences to hold multiple sets of metadata. The timestamp is the exception to this and holds a single set of metadata. Root's attributes are role and Pkey. Timestamp contains the snapshot file name, version and hash. Snapshot contains the targets filename and version, targets attributes and image name, ECU ID and hash. The payloads can only be a set of functional payloads, this reduces the model outputs significantly. Payloads can also be 'Junk'. If a payload is junk, the update is cancelled. The payload signatures are computed using the SKey and the payload of the role.

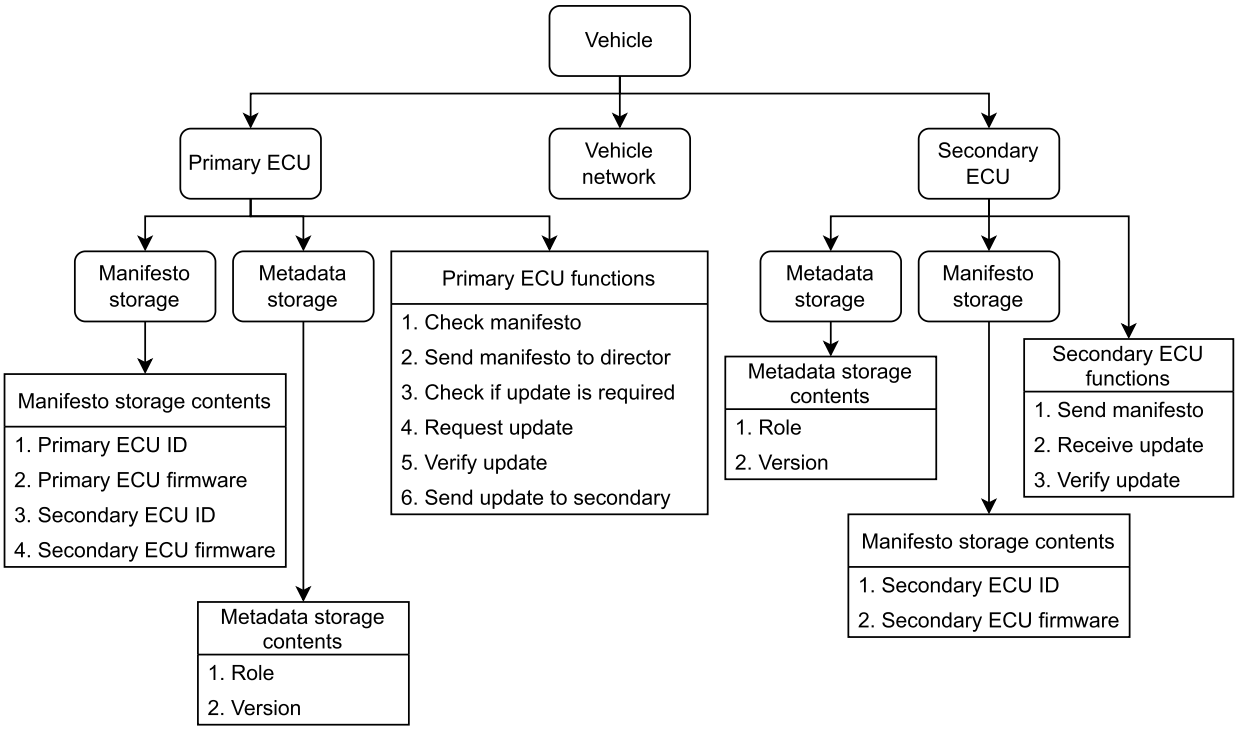


Fig. 5. The vehicle hierarchy in the Uptane model, showing storage and functions of both the primary and secondary ECU.

4.3. Modelling the communication network

The network contains three main portions: sending metadata, sending manifests and sending the update status. Each of these has a channel to send and a channel to receive, defined as: '`channel Send:Component.Component.Message`' '`channel Recv:Component.Component.Message`'. Component one of the send channel is the sender, the second is the destination. The first component of the Recv channel is the recipient, the second is the sender. The network is modelled as "$Network(X) = \Box a1 : X, a2 : X@Send.a1.a2?m \rightarrow Recv.a2.a1.m$", taking the send message and converting it to the receive message, as well as swapping the location parameters of the send and receive components.

4.4. Modelling the vehicle

This section describes the models of the primary ECU, the secondary ECU and their relevant functionalities. Fig. 5 shows the hierarchy of the Uptane model on the vehicle side. On the left branch, the figure shows the processes modelling the components and functions of the primary ECU. We present more modelling details in Section 4.4.1. On the right branch, the figure shows the processes modelling components of the secondary ECU with further details in Section 4.4.2.

4.4.1. Primary ECU

The primary ECU is modelled by a recursive process. Fig. 6 shows the CSP code used for the primary ECU. First the process creates a vehicle manifest based on the secondary ECU's manifest and its own, lines 2 and 3. This is sent to the director, line 4. The director's response informs the primary if the vehicle needs to update, lines 5 to 6. This begins the role verification process described in Section 4.4.3, requesting and verifying the metadata from all roles, from both the director and image repository. Once this is done, the vehicle will update.

4.4.2. Secondary ECU

The secondary ECU is modelled by first getting its manifest from the secondary storage and then sending it to the primary. This can be seen in lines 2 and 3 of Fig. 7. The primary will respond by telling the secondary whether to update or not, lines 4 and 5. If the secondary needs to update, it will request the targets metadata and do a partial verification. The partial verification only checks the target's metadata and then will accept the update if successful, lines 8 to 11.

4.4.3. Role verification

Each verification process starts by requesting the role metadata. This is encoded by CSP codes on lines 3 and 4 of Fig. 8. After receiving the metadata, the verification process will check that the public key matches the private key, as encoded by

```
1  Primary_ECU_Loop(current_time) =
2    ECU_Recv.PrimaryECU.SecondaryECU?ECU_mani →
3    get_manifesto_from_Primary_storage?mani →
4    Send.PrimaryECU.Director.ManifestoHeader.FMPH.⟨...⟩ →
5    Recv.PrimaryECU.Director?status →
6     if status = Fine then
7       ECU_Send.PrimaryECU.SecondaryECU.Fine →
8       inform.Spoof Attack →
9       STOP
10   else
11      Download_and_check_Metadata_Full(current_time);
12      get_metadata_from_storage.TargetsFilename1?version →
13      if version = 1 then
14         STOP
15      else
16         Primary_ECU_Loop(current_time)
```

Fig. 6. CSP model of the primary ECU function.

```
1  Secondary_ECU_Loop(current_time) =
2    get_manifesto_from_Secondary_storage?ECU_mani →
3    ECU_Send.SecondaryECU.PrimaryECU.ECU_mani →
4    ECU_Recv.SecondaryECU.PrimaryECU?status →
5    if status = Fine then
6       STOP
7    else
8       download_and_P_check_targets_metadata(current_time);
9       get_manifesto_from_Secondary_storage?ECU_mani →
10      ECU_Send.SecondaryECU.PrimaryECU.ECU_mani →
11      STOP
```

Fig. 7. CSP model of the secondary ECU function.

```
1  download_and_check_metadata(current_time, server) =
2    get_metadata_from_storage.Filename?version →
3    Send.PrimaryECU.server.RequestMetadata.Role.Filename.(next_version(version)) →
4    Recv.PrimaryECU.server.ResponseMetadata?metadata →
5    if roledecrypt(RolePKey.Role, get_signature(role_metadata)) ≠
6       get_payload(role_metadata) →
7       STOP
8    else
9       if get_version(Role_metadata) ≤ version then
10         STOP
11      else
12         if get_expiration_time(Role_metadata) ≤ current_time then
13            inform.FreezeAttack →
14            STOP
15         else
16            replace_metadata_from_storage.Filename.(next_version(version)) →
17            SKIP
```

Fig. 8. CSP model of the verification of metadata in Uptane.

lines 5 to 7. Next is to verify that the version number in the new metadata is higher than the installed version, as encoded by lines 9 and 10. The system will then check that the current time is less than the expiration time of the new metadata, as encoded by lines 12 to 14. After the checks, the role metadata will update to the new version, as encoded by lines 16 and 17. If any of the verification steps fail, the update stops.

4.4.4. Manifesto and manifesto storage

The manifest storage contains one manifest for the primary and one for the secondary. These manifests have two attributes: the ECU ID and the firmware version. The ECU ID identifies what ECU is outdated, and the firmware version will let the server know if the vehicle is outdated. There is a manifest for each ECU and combination of all the manifests for the vehicle manifest, constructed by the primary ECU. The director, primary and secondary ECU all have their own manifest storage.

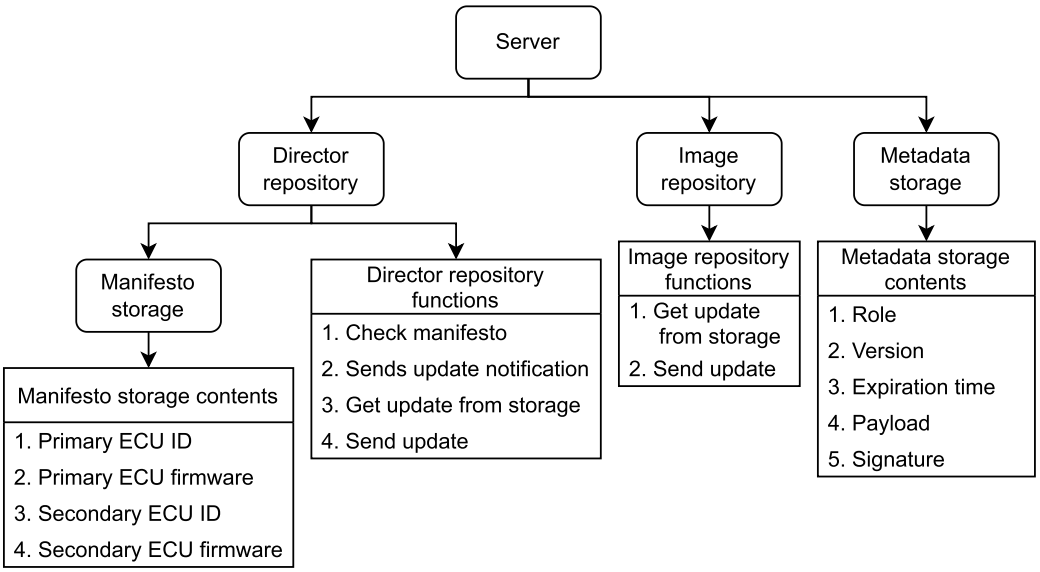


Fig. 9. The server hierarchy in the Uptane model, showing the connectivity between the metadata storage, image and director repository, with functionality on content of each.

4.5. Modelling the server

The server has two components and two storage systems. The time server has been abstracted from the model to simplify the system. A director and image repository are the two main components within the server. There is also the role metadata storage and manifesto storage, the latter of which is only used by the director. Fig. 9 shows the hierarchy of the Uptane model on the server side. In the following, let us explain the Uptane model on the server side in more detail.

4.5.1. Role metadata storage

The role metadata storage is called by the director and the image repository when a request is made for the primary to update. The role metadata storage is shown as a single entity when in fact it is four separate storage systems, one for each role. Each consists of the version, expiration time, unique payload and signature.

4.5.2. Manifesto storage

The server manifesto storage is exclusive to the director, as the image repository never needs to call the manifesto storage. Within the manifesto storage are the up-to-date ECU manifestos from the primary and the secondary. Each ECU manifesto will contain the ECU ID and the image name. This is identical to the manifesto storage found in the vehicle.

4.5.3. Director and image repository

Fig. 10 shows the CSP code that encodes the director and image repository loops. The director loop is modelled by the CSP code from lines 1 to 15; image repository loop is from lines 17 to 19. The server starts with the director. Upon receiving the primary ECU's manifest, the director compares this to its currently held one, checking if it is up to date. This is encoded from lines 2 to 6. The director will inform the primary of its update status, up to date or out of date, shown on line 7. The primary will request metadata and the update from both the director and image repository, sending the data upon the requests. After meeting all the requests, the update cycle is complete, lines 11 to 16 for the director and lines 18 and 19 for the image repository.

4.6. Model integration

In order to function, the system needs to be synchronised. To do this, we combine two operations, running them in parallel. With this method, we connect all the components of the server, vehicle, and the network. This gives us our synchronised Uptane model.

5. Test-case generation

This section discusses the attacks and their implementation within the model in Section 5.1, mutating the model to generate more test cases in Section 5.2. The generation of an exhaustive list of test cases from these mutations is covered in Section 5.3 and Section 5.4.

```
1  Director_Loop() =
2    Recv.Director.PrimaryECU?rmani →
3    get_manifesto_from_Director_storage?mani →
4    if get_Pfirmware_from_manifesto(rmani) = get_Pfirmware_from_manifesto(mani) then
5      if get_Sfirmware_from_manifesto(rmani) =
6        get_Sfirmware_from_manifesto(mani) then
7        Send.Director.PrimaryECU.Fine →
8        Director_Loop()
9      else
10       Send.Director.PrimaryECU.Update →
11       Server_Role_Get_Resposnse(Director);
12       Director_Loop()
13   else
14     Send.Director.PrimaryECU.Update →
15     Server_Role_Get_Resposnse(Director);
16       Director_Loop()
17 Image_Repo_Loop() =
18   Server_Role_Get_Resposnse(ImageRepo);
19   Image_Repo_Loop()
```

Fig. 10. Example CSP code of the director and image repository main functions.

```
1  MITMblock(X) =
2    □a1 : X, a2 : X@Send.a1.a2?m → MITMblock(X)
3  MITMeaves(X) =
4    □a1 : X, a2 : X@Send.a1.a2?m → Recv.attacker.a1.m → Recv.a2.a1.m → MITMeaves(X)
5  MITMFreeze_Attack(X) =
6    (□a1 : X, a2 : X@Send.a1.a2?Payload?Image?Keys →
7      Recv.a2.a1.Old_Payload.Old_Image → MITMFreeze_Attack(X))□
8    (□a1 : X, a2 : X@Send.a1.a2?m → Recv.a2.a1.m → MITMFreeze_Attack(X))
9  MITMSpoofing_Attack(X) =
10   (□a1 : X, a2 : X@Send.a1.a2?m → Recv.a2.a1.Fine → MITMSpoofing_Attack(X))□
11   (□a1 : X, a2 : X@Send.a1.a2?m → Recv.a2.a1.m → MITMSpoofing_Attack(X))
12 MITMeditwithkeys(X) =
13   (□a1 : X, a2 : X@Send.a1.a2?Payload?Image?Keys → Recv.a2.a1.Payload.BadImage.
14     BadKeys → MITMeditwithkeys(X))□
15   (□a1 : X, a2 : X@Send.a1.a2?m → Recv.a2.a1.m → MITMeditwithkeys(X))
```

Fig. 11. CSP models of attacks for: block, eavesdrop, freeze, spoof and edit with keys.

5.1. Modelling attacks

In this section, we model five attacks, mentioned in Section 2.2, and integrate them into the Uptane model. Each attack model replaces the network in the Uptane model and carries out attacks on the system from the point of an outside attack on a compromised network.

5.1.1. Block

The block attack can be seen in Fig. 11 on lines 1 and 2. On line 2 the block attack will receive the messages intended to be sent, but does not send them, as it calls itself and the process starts again. The blocker does not use the "`Recv.a2.a1.m`" that the network uses to deliver messages, preventing the model from sending messages.

5.1.2. Eavesdrop

The eavesdropping attack will take every message and deliver it to its intended destination as well as the attacker. This creates a one-in two-out flow on a compromised network, with the attacker receiving every message sent. The eavesdropping attack can be seen on lines 3 and 4 in Fig. 11. Line 4 is where the eavesdropper sends all traffic to the attacker before sending it to the intended recipient.

5.1.3. Freeze

The freeze attack interrupts the update and sends a properly signed - but old - update to the vehicle. This prevents the vehicle from updating even if newer updates exist. In Fig. 11 the freeze attack is on lines 5 to 8, lines 6 and 7 showing the attack looking for metadata and the update. Once found, it sends the old update. If the metadata is not being sent, it acts as the normal network, allowing the messages to be sent untouched, as seen on line 8.

5.1.4. Spoof

When the server informs the vehicle of an update, the spoofing attack will relay back to the vehicle that it is already up to date, stopping any updates. This attack can be seen on lines 9 to 11 in Fig. 11. On line 10, the attack looks for the vehicle

to request if it needs an update, and then informs it that it is up to date (using the dedicated message "`Fine`"). "`Fine`" is used to inform a vehicle that it is fully up-to-date regardless of its current update status. Line 11 allows the attack to act as the normal network in all other instances.

5.1.5. Edit with and without keys

We split the edit into two attacks, one with keys and one without. Both will have similar processes, but with an extra step for the edit with keys. The editing attack can be seen on lines 12 to 15 in Fig. 11. On line 13, the editing attack checks if the message matches its criteria, the criteria being: does this message contain an update? If the message isn't sending an update, the attack will act as the network, as seen on line 15. If the message contains an update, the attack converts the message to a receive. It then replaces the update to a bad update with the keys to verify the bad update, noted as bad keys. The edit attack without keys is the same but without replacing the keys or signature, leaving the signature and keys for the legitimate update.

5.1.6. Compromised models

Within Section 4.3 we discuss how we connected the individual components. With each of the 5 attacks created in Section 5.1, the method to combine them is the same: replace the network with one of our attacks. This makes 6 compromised Uptane models: eavesdropping Uptane, blocking Uptane, freezing Uptane, spoofing Uptane, editing Uptane with keys and editing Uptane without keys.

5.2. Mutating the uptane model

The Uptane model created, discussed in Section 4, allows us to test attacks on a live implementation of the Uptane system. Our model assumes complete compliance by the implementation, complete compliance being that all checks and verification are implemented. However, if it does not fully adhere to the specification, an implementation may introduce some vulnerabilities. This means security test cases generated by the Uptane model and the attack model will not detect this. The reason is that our model encodes all the defences specified by Uptane. If the implementation has missed one, there is no test case generated to check for these vulnerabilities. In order to test for vulnerabilities that may have been introduced via straying from the specification, we use mutated models.

To test this, we created mutations of the model that have defences removed. This then generates test cases based on the mutations, giving us the opportunity to check if the implementation has vulnerabilities associated with each mutation. The Uptane specification records the attacks it can prevent. Finding the associated attacks in our Uptane model and removing the defences give us a mutated model. This mutated model is then tested using the same attack model to see if the mutations have given us more test cases. We then create new attacks based on the new test cases and check them against the implementation. The check assesses compliance and any vulnerabilities that this may have introduced.

In particular, for the freeze attack, the model of Uptane has been changed so that it no longer checks the current time in the metadata verification process. This can be seen in Section 4.4.3 and lines 12 to 15 in Fig. 8. The verification process checks for a freeze attack by checking the time to ensure that the update is indeed the most recent and not outdated. The mutation considered here removes this check in each one of the verification steps no longer requiring the time check. This makes the model vulnerable to freezing attacks. All other aspects of the model remain the same. The time check at the verification of the root, timestamp, snapshot and targets for both director and image repository are the only changes made to this mutation. Using the mutated model, we will generate test cases to validate if the implementation of Uptane is indeed vulnerable to freezing attacks.

Other mutations to the model that we find are doing a partial verification instead of a full verification. This means that they are only updating from the Director repository and doing a partial verification and update, even when the system is up to date.

5.3. Generating security test cases using refinement checks

To generate test cases from the compromised models we first define what a successful attack is so we can identify when the system has been breached.

Eavesdropping is successful if the attacker receives any messages, and is demonstrated when a '`Recv.attacker`' message is transmitted. The block attack is successful if no message is received by its recipient. This means no '`Send`' messages are translated into '`Recv`' messages. Freeze and spoof are successful if, in the case of this mode, the version remains the same. The edit attack is successful if a malicious update is received by the secondary ECU. By adding an if statement that checks for bad firmware, running the behaviour '`do_bad_thing`' indicates a successful attack. These definitions make up our refinement checks: NoEaves, NoBlock, NoFreeze, NoSpoof and NoEdit respectively.

5.3.1. Refinement checks

Refinement checks in FDR4 take a process, in our case what a failed attack should look like. They then compare it to another process, in our case a compromised model. FDR4 uses this comparison to explore the states of the processes in order. If a counter example is found, the refinement is stopped and then the trace (or attack) is reported.

```
1 NoEaves ⊑T Eaves_Uptane
2 NoBlock ⊑T Block_Uptane
3 NoEdit ⊑T Edit_W_Uptane
4 NoEdit ⊑T Edit_WO_Uptane
5 NoSpoof ⊑T Spoof_Uptane
6 NoFreeze ⊑T Freeze_Uptane
```

Fig. 12. Attack refinement checks for: eavesdrop, block, edit with keys, edit without keys, spoof and freeze.

The refinement checks (i.e., assertions) allow us to check if the security properties defined in Section 5.3 are met. Refinement checks against compromised systems fail if the system is potentially vulnerable to the attacks. In such cases, each failure is evidenced by a counter example and considered as a test case. We will use refinement checks to generate security test cases. Each refinement has the following form: "SecurityProperty $\sqsubseteq_T$ CompromisedSystem" where the SecurityProperty and CompromisedSystem processes are corresponding to each attack. In particular, we use FDR4 to check the following refinements, see Fig. 12, to check if the Uptane system is secure against: 1 eavesdropping attack, 2 blocking attack, 3 editing attack with keys, 4 editing attack without keys, 5 spoofing attack and 6 freezing attack.

For example, in the eavesdropping attack case, the SecurityProperty is "NoEaves" and the CompromisedSystem is "Eaves_Uptane" to check if the Uptane system is secure against the eavesdropping attack. FDR4 is used to generate test cases to check for refinements.

5.4. Exhaustive test cases

Each attack may have multiple points where it successfully penetrates the system. To discover the further attack points within the model, we developed a Python script to allow FDR4 to test beyond the first successful attack. To use FDR4 with Python we needed to use Python 2.7 [24], as this integration isn't available for Python 3. First, we create a script in Python that starts a session in FDR4, this session is given the compromised Uptane model. Once FDR4 has finished, it will output the trace if the attack is successful. If the attack fails, no trace is sent, and no more attacks are possible. With successful attacks, the Python script will create a file to store the trace, and this trace is then appended to conform to CSP. This is done by adding "TCn = " at the start, the "n" being a number that increases with every new trace. We also add the prefix "$\rightarrow$" to the end of each process and the "$\rightarrow RUN(Events)$" process at the end to create this conformity. This CSP trace is our counter example that is then added to the assertion that Python is using. This stops the same attack from being flagged again, therefore allowing further attacks to be discovered. Looking at the process Python uses, "$assert\ SecurityProperty \ \square\ TCX \ \sqsubseteq_T CompromisedSystem$", we can see that the attack being used is shown as "SecurityProperty". This is a representation of what should happen if there is no attack or if an attack fails. The counter examples that are generated are imported to the assertion shown by "TCX", and for every counter example found there will be a new "TCX". We then tested this on an attacked model represented by "CompromisedSystem".

Using this, we generated 23 test cases from the normal Uptane model: 18 for eavesdrop, 1 for block, 0 for freeze, 2 for spoof, 2 for edit with keys and 0 for edit without keys. The 23 generated test cases show that the Uptane system suffers from this attack. On the mutated model, there were 25 test cases. 23 of these were from the original and 2 were from the freeze attack.

6. Experiment

In this section, we perform a security evaluation of the Uptane reference implementation based on the test cases generated in the previous section. These test cases are manually translated into real world attacks to be tested on the demo implementation of Uptane on a physical test-bed. The tests (a.k.a. attacks) start by using Ettercap to perform Address Resolution Protocol (ARP) poisoning on the network. This directs all packets through the tester machine (a.k.a. attacker). Then, Ettercap is used to manipulate these packets by defining suitable Ettercap filters.

6.1. Test-bed

The Uptane reference implementation is realised in a physical test-bed consisting of: Two Raspberry Pi 4s running Raspbian, acting as the primary and secondary ECUs, a laptop running Ubuntu as the server, and another laptop with Kali Linux acting as the tester machine. All of the components are connected via Ethernet to a router acting as our network that allows them to communicate. To monitor the system and the attacks, the server and the attacker are running Wireshark, a packet sniffing tool that shows all of the traffic passing through a given network.

6.2. Test cases 1 to 18 - eavesdropping attack

Fig. 13 shows the trace of a eavesdrop test case. Line 3 shows that the hacker has received the message sent over the network. This attack is successful and we can use this trace to translate this into a real world attack. These test cases

```
1 get_manifesto_from_secondary_storage...
2 ...
3 Send.PrimaryECU.Director.ManifestoHeader.FMPH.⟨...⟩
4 Recv.Hacker.PrimaryECU.ManifestoHeader.FMPH.⟨...⟩
```

Fig. 13. The counter example trace generated by the eavesdrop attack.

```
1 Send.PrimaryECU.Director.ManifestoHeader.FMPH.⟨...⟩
2 Recv.Director.PrimaryECU.ManifestoHeader.FMPH.⟨...⟩
```

Fig. 14. Example trace provided by the block test case.

```
 1 get_manifesto_from_Secondary_storage...
 2 get_manifesto_from_Primary_storage.ManifestoHeader.FMPH.⟨MPCH.ID_Primary.
 3    Firmware_current, MPCH.ID_Secondary.Firmware_current⟩
 4 Send.PrimaryECU.Director.ManifestoHeader.FMPH.⟨MPCH.ID_Primary.
 5    Firmware_current, MPCH.ID_Secondary.Firmware_current⟩
 6 Recv.Director.PrimaryECU.ManifestoHeader.FMPH.⟨MPCH.ID_Primary.
 7    Firmware_current, MPCH.ID_Secondary.Firmware_current⟩
 8 get_manifesto_from_Director_storage.ManifestoHeader.FMPH.⟨MPCH.ID_Primary.
 9    Firmware_new, MPCH.ID_Secondary.Firmware_new⟩
10 Send.Director.PrimaryECU.Update
11 Recv.PrimaryECU.Director.Fine
```

Fig. 15. Example trace provided by the spoof test case.

are translated into an executable test case consisting of two steps: (i) setting up with a listener using Ettercap, to see all messages sent; (ii) requesting an update from the secondary ECU. In carrying out both of these steps, Ettercap can see the packets being passed between the two, including the firmware update. The packets that Ettercap intercept correctly correspond to the packets collected from Wireshark on the host machine. Therefore, the attack is successful.

6.3. Test case 19 - blocking attack

Fig. 14 identifies line 2 as a variation from the block Uptane model. This shows that this message was not received. Using this trace, we can translate the attack into a real world attack. Test case 19 consists of two steps: (i) setting up a block filter in Ettercap; (ii) requesting an update. Ettercap's filters have a lot of applications and work much the same as most scripting languages [25]. The filter calls two commands "`kill();drop()`". The kill command ends the connection and stops the packet from reaching its destination. Carrying out these two steps causes the ECUs to appear to be unresponsive and eventually send the connection timed out error. Wireshark on the host machine also confirms that the packets are being stopped: although TCP reset packets are still being sent, trying to re-establish a connection. Therefore, the attack is successful.

6.4. Test cases 20 and 21 - spoofing attack

Fig. 15 shows the trace of test case 20. Line 10 shows that the director sends an "Update" message, but on line 11 the received message it shows "Fine". This prevents the vehicle from updating. Using this trace, we can translate this attack into a real world attack for our test bed. The spoof attack contains the steps (i) recording the response of the server when the vehicle is up to date, (ii) creating a filter to send this response instead of a legitimate one, and (iii) requesting an update. However, this shares many aspects with the freeze attack, seen in Section 6.6. The attack is unsuccessful, and we have discovered that the implementation we are working with differs from the specification.

6.5. Test cases 22 and 23 - editing attack with keys

In Fig. 17, lines 3 to 6 show the Director sending the Primary the update payload with "Image_new", but on the receiving side, lines 7 to 10, the image has been replaced with "Bad_Image". Using this trace, we can translate the model theoretical attack into a real world attack. Test cases 22 and 23 have three steps: (i) create a binary file of the bad target's metadata image repository and Director; (ii) apply an Ettercap filter to the network to change the firmware image in transit, see Fig. 16; (iii) request an update. First, the filter checks the packets for strings present at the start of the Director and Image repositories' encrypted target payloads (lines 1 and 3). Next, we drop the information from the packet (1st statement on lines 2 and 4), and then inject the malicious payload (2nd statement on lines 2 and 4). Line 5 replaces the good update

```
1 if (search(DATA.data, "Director Targets String")){
2   drop(); inject("Directory/Director_attack.bin"); }
3 if (search(DATA.data, "Image Targets String")){
4   drop(); inject("Directory/Image_attack.bin"); }
5 replace("Fresh firmware image", "BadXX firmware image")
```

Fig. 16. Ettercap filter to realise the Editing attack with keys.

```
 1 get_manifesto_from_Secondary_storage...
 2 ...
 3 Send.Director.PrimaryECU.ResponseTargetsMetadata.TM.3.3.TargetsPayloadHeader
 4   .⟨TargetsContentHeader.Image_new.ID_Primary.HashingFirmware.Firmware_new,
 5     TargetsContentHeader.Image_new.ID_Secondary.HashingFirmware.Firmware_new⟩.
 6       EncT.RoleSKey.Targets.TargetsPayloadHeader.⟨...⟩
 7 Recv.PrimaryECU.Director.ResponseTargetsMetadata.TM.3.3.TargetsPayloadHeader
 8   .⟨TargetsContentHeader.Bad_Image.ID_Primary.HashingFirmware.Firmware_new,
 9     TargetsContentHeader.Bad_Image.ID_Secondary.HashingFirmware.Firmware_new⟩.
10       EncT.RoleSKey.Targets.TargetsPayloadHeader.⟨...⟩
11 replace_metadata_from_storage.TargetsFilename1.2
12 Do_bad_thing
```

Fig. 17. This is the relevent trace from FDR for the edit with keys.

with the bad. Both the Primary and Secondary show no signs of attack; the Secondary files show "BadXX firmware image" as the current update. Therefore, the attack is successful in both areas recognised by the test case.

6.6. Mutated test cases 24 and 25 - freeze attack

Turning test case 24 and test case 25 into executable attacks requires 5 steps: (i) listen and record the metadata and update from the server to vehicle using Ettercap; (ii) create a filter that injects this update into the new update and metadata replacing the intended messages; (iii) ARP poison the system before its next update; (iv) activate the filter; (v) perform an update. This attack is similar to the one found in Fig. 16, but it injects the old recorded metadata and update instead of the malicious one. However, when this attack is carried out, it fails and relays back to the vehicle that the update is out of date and it may be a potential attack. The test cases are generated by the mutated model and the freeze attack model. This testing result shows that the implementation of Uptane is indeed secure against freeze attacks.

7. Conclusion & future work

In this paper, we have expanded on our proposed model-based approach for systematically evaluating automotive OTA updates [16]. The approach employs CSP, FDR4, a simplified Dolev-Yao attacker and mutation modelling to enumerate potential attacks. By combining the CSP models of Uptane and the attacks, we have tested Uptane's resilience to the theoretical attacks. With this method, we have tested five attacks on the Uptane system, generating 25 test cases to identify vulnerabilities. We have shown a promising approach to testing OTA systems in a methodical manner, allowing for a bombardment of security tests on a live implementation. Within the scope of the model and the attacks, we can conclude that Uptane is vulnerable to some of these attacks. Being vulnerable to the attacks listed can prevent vehicles from updating, reverse engineering the updates sent and even installing malicious firmware to allow for devastating attacks.

To help prevent the 3 successful attacks - edit with keys, blocker and eavesdropper - Uptane can take the following steps. (i) Edit with keys: To prevent this kind of attack, we can employ better key management, prevent the exposure of keys and instigate regular updates. (ii) Blocker: This attack cannot be prevented in this context. To better prepare against a potential blocking attack, the vehicle could make regular contact with the server. By contacting the server once a day, for example, the vehicle can determine how long it has been without contact. Using this, the manufacturer can determine how much time may pass without contact before the vehicle sends the user a message informing them that there is no contact to the server. This can then tell the user to go to a garage to check the vehicle's integrity. (iii) Eavesdrop: This attack cannot be stopped, but can be rendered next to useless. By employing a strong encryption only between the server and the Primary ECU, the attacker cannot see what is being transmitted and gains little information from the attack. Although this requires better hardware for the Primary ECU, the manufacturer can retain the same hardware for other Secondary ECUs. The Uptane system has shown itself to be resilient to some attacks, but it has also shown various vulnerabilities. The implementation has also included some changes that provide improvements on the specification, preventing the original simplistic spoofing attack from taking place.

Further work will expand the model by adding more attacks and variations of the same attacks. We also plan to create more model mutations by removing and including various aspects of the original model. This will generate more test cases and lead to a more comprehensive security assessment for this automotive OTA system. In addition to adding more to the

model to allow for more test cases, we intend on testing defences to the Uptane model. By adding suggested defences to the model, we can test if the same attacks are still successful on the model, allowing the model to be used to test suggested defences.

Uptane is designed to enable the automotive industry to comply with a compelling need emerging from standards such as ISO/DIS 24089 on software update engineering for road vehicles and regulations such as UNECE WP29 R155 and R156 cybersecurity and software update management for automotive systems, which ultimately mandates a secure OTA software update system for vehicles. Our contribution provides for a framework, whereby implementations of Uptane (which may vary at a lower level across vehicles) could be assured for using effective test case generation. As such, while the formal model suffices to represent the logical architecture of the updating system, the test cases generated would ultimately test the implementations; increasing model mutations will ensure test cases work towards an increasingly comprehensive assessment.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, Comprehensive experimental analyses of automotive attack surfaces, in: Proceedings of the 20th USENIX Security Symposium, 2011, pp. 77–92.
[2] V.H. Le, J. den Hartog, N. Zannone, Security and privacy for innovative automotive applications: a survey, Comput. Commun. 132 (2018) 17–41, https://doi.org/10.1016/j.comcom.2018.09.010.
[3] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, I. Seskar, Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study, 2010, pp. 323–338.
[4] C. Miller, C. Valasek, Remote exploitation of an unaltered passenger vehicle, Defcon 23 (2015) (2015) 1–91.
[5] S. Nie, L. Liu, Y. Du, Free-fall: hacking tesla from wireless to can bus, Defcon (2017) 1–16.
[6] S. Parkinson, P. Ward, K. Wilson, J. Miller, Cyber threats facing autonomous and connected vehicles: future challenges, IEEE Trans. Intell. Transp. Syst. 18 (11) (2017) 2898–2915, https://doi.org/10.1109/TITS.2017.2665968.
[7] T. van Roermund, In-vehicle networks and security, Automot. Syst. Softw. Eng. (2019) 265–282.
[8] Synopsys, SAE International, Securing the modern vehicle: a study of automotive industry cybersecurity practices, https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/securing-the-modern-vehicle.pdf, 2018.
[9] Uptane: securing software updates for automobiles, https://uptane.github.io/.
[10] S. Mahmood, A. Fouillade, H.N. Nguyen, S.A. Shaikh, A model-based security testing approach for automotive over-the-air updates, in: ICSTW 2020, IEEE, 2020, pp. 6–13.
[11] D. Dolev, A.C. Yao, On the security of public key protocols, IEEE Trans. Inf. Theory 29 (2) (1983) 198–208.
[12] E. dos Santos, A. Simpson, D. Schoop, A formal model to facilitate security testing in modern automotive systems, Elect. Proc. Theoret. Comput. Sci. 271 (2018) 95–104.
[13] J. Heneghan, S.A. Shaikh, J. Bryans, M. Cheah, P. Wooderson, Enabling security checking of automotive ECUs with formal CSP models, in: Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop, DSN-W 2019, 2019, pp. 90–97.
[14] M. Cheah, H.N. Nguyen, J.W. Bryans, S.A. Shaikh, Formalising systematic security evaluations using attack trees for automotive applications, in: WISTP 2017, 2017.
[15] M. Cheah, S.A. Shaikh, O.C.L. Haas, A.R. Ruddle, Towards a systematic security evaluation of the automotive bluetooth interface, Veh. Commun. 9 (2017) 8–18, https://doi.org/10.1016/j.vehcom.2017.02.008.
[16] R. Kirk, H. Nguyen, J. Bryans, S. Shaikh, D. Evans, D. Price, Formalising UPTANE in CSP for security testing, in: The 21st IEEE International Conference on Software Quality, Reliability, and Security, QRS2021, 06–10 December, 2021, 2021.
[17] A.W. Roscoe, The Theory and Practice of Concurrency, Pearson, 2005.
[18] I. Cervesato, The Dolev-Yao intruder is the most powerful attacker, in: 16th Annual Symposium on Logic in Computer Science, in: LICS, vol. 1, 2001.
[19] B. Blanchet, Composition theorems for cryptoverif and application to tls 1.3, in: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), 2018, pp. 16–30.
[20] F. Tao, W. Shuaishuai, G. Xiang, F. Junli, Formal security evaluation and improvement of industrial ethernet EtherCAT protocol, J. Comput. Res. Dev. 57 (11) (2020) 2312.
[21] M.S. Bauer, R. Chadha, M. Viswanathan, Modelchecking safety properties in randomized security protocols, in: Logic, Language, and Security, Springer, 2020, pp. 167–183.
[22] X. Hu, C. Liu, S. Liu, W. You, Y. Li, Y. Zhao, A systematic analysis method for 5G non-access stratum signalling security, IEEE Access 7 (2019).
[23] Ettercap home page, https://www.ettercap-project.org/.
[24] The FDR API, https://cocotec.io/fdr/manual/api/api.html.
[25] A. Ornaghi, M. Valleri, Ettercap(8) - Linux man page, https://linux.die.net/man/8/ettercap.