Formally Verifying the Security and Privacy of an Adopted Standard for Software-Update in Cars: Verifying `Uptane 2.0`

Ioana Boureanu, University of Surrey, i.boureanu@surrey.ac.uk

Abstract— **In this paper, we formally analyse the security of `Uptane 2.0` – the latest version[1] of a framework for *over-the-air* (online) delivery of software to cars. We are doing so by using the threat model and security requirements found in standard document that accompanies `Uptane 2.0`, as well as a modulation of this threat model and requirements added by ourselves, for a deeper analysis. To undertake this verification, we use the well-known formal protocol-verifier and theorem prover called `Tamarin`. We discuss our responsible disclosure to and work with the `Uptane` Alliance.**

I. Introduction

`Uptane 2.0` is the latest version an open and secure software update framework, which consists of a design of *over-the-air (OTA)* delivery of software to automobile *electronic control units (ECUs)*. It is under standardisation and it is integrated into Automotive Grade Linux, an open source system currently used by many large OEMs, and has also been adopted by a number of U.S. and international manufacturers. Its website, `https://uptane.github.io/`, states that: “the framework protects against malicious actors who can compromise servers and networks used to sign and deliver updates. Hence, it is designed to be resilient even to the best efforts of nation state attackers.”

In this paper, we follow the standard document of `Uptane` [1] and put this type of claims to the test. We do focus on the requirements in the standard rather than those on the website.

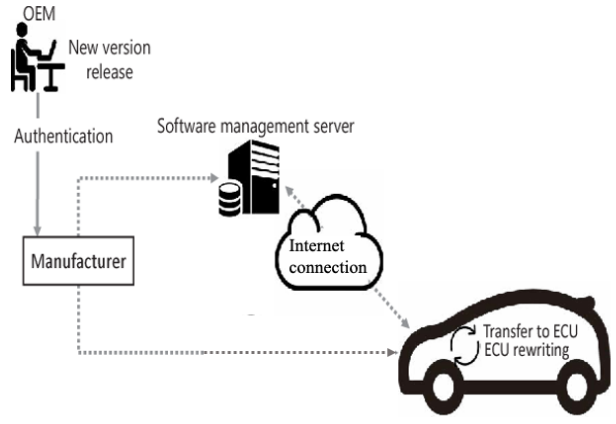


Fig. 1. OTA Software-Updates in Cars

Surrey Centre for Cyber Security (SCCS)

[1]“Latest” is meant at the time of this writing, i.e., in April 2023.

A. An Overview of `Uptane`

a) OTA Software-Updates in Cars: This operates in the setting sketched in Figure 1. In Figure 1, we see that the idea is that different components (i.e., ECU) inside the car, such as the electronic brake control module, can get their software updates from/via an online server managed by the manufacturer of the car, e.g., Audi, with the software provided by the different manufacturers of components, also known as Original Equipment Manufacturers (or *OEMs*, for short), e.g., SIEMENS.

b) Overview of `Uptane`'s Components: But in reality, the ECUs inside the cars are organised in a hierarchy: *primary ECUs* and tiers of *secondary ECUs*. Also, the server functionality is often separated into different services: part-cataloging the software and book-keeping which ECU in which car may have which version, the time/synchronisation part, etc. In this vein, the `Uptane` framework has the

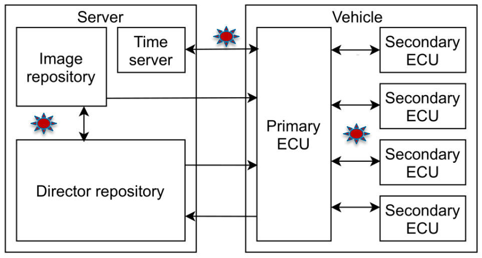


Fig. 2. `Uptane` Components (figure adapted from [2])

components shown in Figure 2, and it describes each of them and the protocols they need to follow such that a software update gets from an OEM to an ECU.

c) Overview of `Uptane`'s Design Flow: Without yet detailing the `Uptane`'s mechanisms, we preempt that the `Uptane` end-to-end updates are based on cryptographic primitives and protocols. Take use-case 3.2.2.2 of the `Uptane` standard, for instance: say SIEMENS has created a revised software image for an electronic brake control module; it will digitally sign it together with the relevant metadata, $Sign_{privateKey\_Siemens}(new-sw-image||metadata)$, it will sent the new software and the signature to the servers of, say, the VW group who – upon verification– will sign more details, $Sign_{privateKey\_VW}(\ldots||added\_metadata.||conflicts\ldots)$, upload the new software to the Image repository and update the Directory Index (see Figure 2). The security requirements of the channels between parties (i.e., authentication, integrity) are also of importance to the security of these updates (and this is included in part in the standard). On Figure 2, the channels envisaged are marked via red stars. Finally, `Uptane` specifies how the car and its relevant ECUs can obtain such an update.

B. *Contributions*

Our contributions are as follows:
– we devise a hierarchical threat model, including and extending the one dictated by the `Uptane 2.0` standard;
– we consider a series of security requirements for `Uptane-2.0`, including those found in standard;
– in the `Tamarin` tool, we model the `Uptane-2.0` system and the aforesaid requirements in these threat models;
– we find a series of flaws and disclose them to the "owners" of `Uptane 2.0`.

C. *Structure*

The structure of the paper is as follows. In Section II, we give a summary of the `Uptane` system, its requirements and threat model as per the the `Uptane 2.0` standard. In Section III, we present our incorporation and augmentation of the `Uptane-2.0` threat and requirements in our own refined attacker model and set of security goals. In Section IV, we show how we model this in the `Tamarin` tool, and the security-verification results obtained. In the remainder, we discuss related and future work, and conclude.

II. Distilling the `Uptane 2.0` Standard

We now describe the parts, components, assumptions and requirements which we extracted out of the `Uptane 2.0` standard and are essential to the formal verification of its security and privacy.

A. *`Uptane`'s Components*

These components, as described in Sections 5.1 and 5.2 of the standard, are:

- *Uptane Server*, which consists of the following.
  (a) *Time server* – provides a secure way for ECUs to know the time.
  (b) *Image repository* – contains binary images to install and signed metadata about those images. This is the server that the primary ECU will download its updates from.
  (c) *Director repository* – is connected to an inventory database containing information on vehicles, ECUs, and software revisions. It instructs ECUs as to which images will be installed in response to a primary ECU uploading its vehicle version manifests.
- *Vehicle*, where the components of interest are as follows.
  (i) *Primary ECU (PECU)* – capable of downloading images and associated metadata from the `Uptane` servers, verifying the signatures on all update metadata and downloading updates on behalf of its associated Secondary ECUs.
  (ii) *Secondary ECU (SECU)* – capable of performing either full or partial verification of metadata. It also sends signed information about its installed images to its associated Primary ECU.

Requirements, described in Sections 5.1, 5.2 and 5.3 of the standard, intrinsically linked to the components above, and to functionality, security and privacy, are:

- Any ECU needs to be able to have a secure way of verifying the current time.
- At deploy time, each ECU will also have a copy of the `Uptane` metadata, e.g., the public keys of all the `Uptane` roles relevant to the ECU as well as the current time and their own ECU signing/MAC-ing key.
- While signing keys are required to be unique to an ECU, the secret keys used to decrypt images need not be unique.

The standard, Sections 5.1– 5.3, describes other entities/roles (e.g., root, snapshot, etc.), but –for our purposes here– these can be abstracted away.

B. *`Uptane`'s OTA Software-Updates*

As per Section 5.4.2 of the standard, the primary ECU will do the following during the update process:

a. Construct and send vehicle version manifest (Figure 3);
b. Download and check current time (Section 5.4.2.2 of the standard);
c. Download and verify metadata (Section 5.4.2.3 of the standard);
d. Download and verify images (Section 5.4.2.4 of the standard);
e. SHOULD (not SHALL) send latest time to SECUs (Section 5.4.2.5 of the standard);
f. Send metadata to SECUs (Section 5.4.2.6 of the standard);
g. Send images to SECUs (Section 5.4.2.7 of the standard).

Sending an Update Request: We now describe the process of sending a request for a software update. In Figure 3, we show the message flow which initiates the update process. Each secondary ECU sends a signed hash

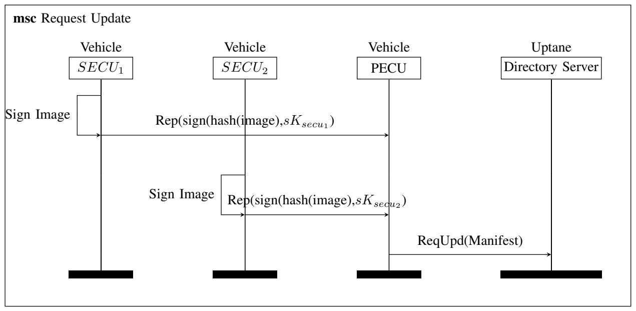


Fig. 3. Requesting Updates in `Uptane`

of its image to the primary ECU which sends aggregates all these signatures into a manifest and sends that to `Uptane`'s Directory Server (see Section 5.4.2.1 of the standard for details). SECUs can send their version reports at any time so that they are already stored on the Primary when it wishes to check for updates. Alternatively, the Primary can request a version report from each Secondary at the time of the update check.

A symmetric flow exists for updating an image by, e.g., a SECU, but in the opposite direction: i.e., the server sends an answer to the SECU's request in a manifest, proxied via a PECU.

Also, there exists an identical protocol for asking the correct time, i.e., the SECUs, PECUs, request not a software update but re-sync their clocks with the server. This is done in case with every software-image update

C. `Uptane`'s System and Security Assumptions

We gather the following defining system-engineering requirements from all the aspects that the standard sets out.

– **Req. 1**: There are several servers running `Uptane` and each can update the ECUs with different software.

– **Req. 2**: There are several secondary ECUs and not all get updated at once, by the same server, via the primary ECU as a proxy. Yet, according to Section 5.3.2.1, all ECUs need to be present in the vehicle manifest sent as part of the update request.

– **Req. 3**: The channel between the primary ECU and secondary ECUs is not secure, even if this may be outdated (nowadays, these channels have integrity protection).

– **Req. 4**: The channel between the primary ECU and the `Uptane` server is secure.

– **Req. 5**: The channels between the components in the `Uptane` server are secure.

– **Req. 6**: "If any step fails, the ECU shall return an error code[2] indicating the failure." (see Section 5.4.4.2).

a) `Uptane`'s Adversaries Capabilities: According to Section 4.2 of standard, an `Uptane` attacker can do any of the following:

a. intercept and modify network traffic, inside and outside cars;
b. compromise the Director repository or Image repository server, but not both;
c. compromise either a primary ECU or a secondary ECU, but not both in the same vehicle

b) `Uptane`'s Security Requirements: According to Section 4.1 of standard, no attacker should be able to do the below.
– deny installation of updates that is doing a denial of service (DoS), which we hereby denote $\mathbf{G}^{\mathbf{std.}}_{\mathbf{no\text{-}DoS}}$;
– read the contents of updates to discover confidential information, which we hereby denote $\mathbf{G}^{\mathbf{std.}}_{\mathbf{conf.}}$;
– inflict changes to alter certain functions or control ECUs, which we hereby denote $\mathbf{G}^{\mathbf{std.}}_{\mathbf{no\text{-}destroy}}$.

III. Security-Models for Uptane

A. Our Threat Hierarchy

Based on the attackers stipulated by the standard and summarised by us in Section II-C.0.a, we create a hierarchy of increasingly stronger attackers, as follows:

[2]This formulation makes us ascertain that the explicit failure needs to be indicated.

Attacker-Tier 1 (Att. 1): The attacker does not control anything, but can listen onto the channels, i.e., normal Dolev-Yao.

Attacker-Tier 2 (Att. 2): The attacker does not control anything but can listen onto the channels + he can corrupt a secondary ECU.

Attacker-Tier 3 (Att. 3): The attacker does not control anything but can listen onto the channels + he can corrupt the primary ECU.

Attacker-Tier 4 (Att. 4): The attacker does not control anything but can listen onto the channels + he can corrupt the primary ECU or a secondary ECU, but not both.

Attacker-Tier 5 (Att. 5): The attacker is capable of compromising and controlling either a Director repository or Image repository server.

B. Our Security & Privacy Goals

Based on the requirements stipulated by the standard and summarised by us in Section II-C.0.b, we create our own requirements which incorporate to the ones in the standard, but also add to them as follows:

– no desynchronisation of parties (subsuming no denial-of-service), e.g., if an Image Server thought it ran with a PECU and vice-versa, they both finish their executions; we denote this as $\mathbf{G}^{\mathbf{formal}}_{\mathbf{no\text{-}desync}}$;

– secure agreement (subsuming confidentiality and authentication), e.g., if an Image Server thinks a SECU is updated, so does its PECU and so does itself, and vice-versa; we denote this as $\mathbf{G}^{\mathbf{formal}}_{\mathbf{agree}}$)

– time correctness, e.g, no attacker can delay an update without some parties realising; we denote this as $\mathbf{G}^{\mathbf{formal}}_{\mathbf{time}}$);

– unlinkability, e.g., no attacker can know if a given SECU has updated or not; we denote this as $\mathbf{G}^{\mathbf{formal}}_{\mathbf{unlink}}$)

We note that our security goals above, i.e., $\mathbf{G}^{\mathbf{formal}}{}_{\dots}$ include and extend the goals in the standard, i.e., $\mathbf{G}^{\mathbf{std}}{}_{\dots}$. Concretely, we use the symbol "$\supset$" for this inclusion and recount this explicitly below:

$$\mathbf{G}^{\mathbf{formal}}_{\mathbf{no\text{-}desync}} \supset \mathbf{G}^{\mathbf{std.}}_{\mathbf{no\text{-}DoS}}$$

$$\mathbf{G}^{\mathbf{formal}}_{\mathbf{agree}} \supset \mathbf{G}^{\mathbf{std.}}_{\mathbf{conf.}}$$

$$\mathbf{G}^{\mathbf{formal}}_{\mathbf{agree}} \supset \mathbf{G}^{\mathbf{std.}}_{\mathbf{no-destroy.}}$$

$$\mathbf{G}^{\mathbf{formal}}_{\mathbf{agree}} \supset \mathbf{G}^{\mathbf{formal}}_{\mathbf{no\text{-}desync}}$$

This means that if any of our goals $\mathbf{G}^{\mathbf{formal}}{}_{\dots}$ fail, the included $\mathbf{G}^{\mathbf{std}}{}_{\dots}$ also fails but not vice-versa.

Also, note that if one writes "$\mathbf{G}_x$", from "x" – it is clear if one means a formal goal, i.e., $\mathbf{G}^{\mathbf{formal}}{}_x$, or a goal in the standard, i.e., $\mathbf{G}^{\mathbf{std}}{}_x$. So, in reporting our verification results (see Figure 6), we simply write $\mathbf{G}_x$.

IV. Dolev-Yao Verification of Uptane

a) Symbolic/Dolev-Yao Verification: Symbolic verification, also known as *Dolev-Yao verification* [3], [4], [5] is a formal method for security analysis. Therein, cryptography is assumed to be perfect/correct [6], yet it allows for a computationally unbounded *Dolev-Yao (DY)* attacker which can hijack all protocol sessions and communication, and corrupt all parties modulo the perfect-cryptography assumption, and a set of abstract, algebraic rules that encapsulate these powers [6]. This makes symbolic analysis amenable to mechanisation into (often automatisable) protocol verifiers such as the `Tamarin` prover [4].

Symbolic tools can analyse arbitrarily-many, concurrent system executions, which computational analysis rarely attains. This leads to symbolic verification having consistently and successfully exhibited dangerous protocol-design errors when interleaving unbounded numbers of protocol runs executing concurrently [7], [8], [9].

b) The `Tamarin` Prover: `Tamarin` models are transition system, whereby the states are modelled as multiset of logical predicates/*facts*, over user-defined protocol variables. The transition system's rule emulates rewriting over these facts; i.e., the semantics is a fragment of multiset rewriting logic [4]. The properties to verify are expressed as lemmas over these facts, in a fragment of first-order logic that allows reasoning in all or in one trace/execution of the transition system.

Due to its support for unbounded verification, `Tamarin` proofs are not guaranteed to terminate or can take a long time to complete. Consequently, `Tamarin` supports various heuristics to cover the search space yielded by the constraint-solving problems underlying the analysis. These heuristics determine which rules should be prioritised during the proof search. Users can also create a bespoke search heuristic or "oracle" [4] to pass to `Tamarin` in cases where `Tamarin`'s default heuristics fail. When *Tamarin* terminates, if an attask it will produce a trace which could help in the discovery of exploits within a protocol.

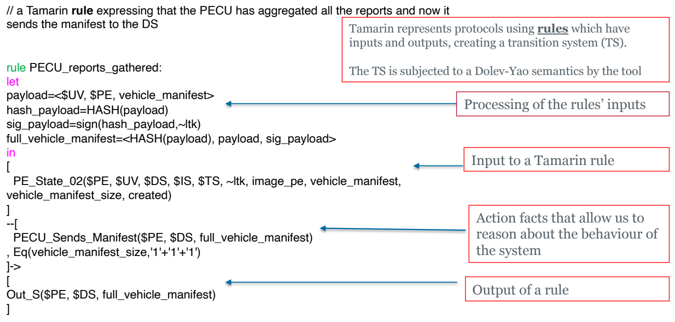
Tamarin has been extensively and successfully used in the security verification of large systems such a TLS 1.3, 5G-AKA, including when they were under standardisation. For more details see `http://tamarin-prover.github.io/`.

A. Modelling

We modelled the `Uptane` in all 5 tiers of attacker above and with all the security goals above in `Tamarin`.

We now give a flavour of modelling in `Tamarin`, which we did in version 1.6.0 of the tool. To this end, Figure 4 shows how a rule in `Tamarin` is modelled and our annotations show how these rules are used to encode systems behaviour.

And, Figure 5 shows how a lemma in `Tamarin` is modelled. This intuitively shows that we included, in our model, predicates/facts that track when SECUs and PECUs are linked as part of our vehicle, and then we use these in a lemma to check the correctness/executability of our model, even in the presence of an attacker. That is, by using the "t01", "t02" labels shown in the figure, we check that if a given SECU were to be linked twice to a PECU, then –in fact– the two times are one and the same: i.e., in our model,



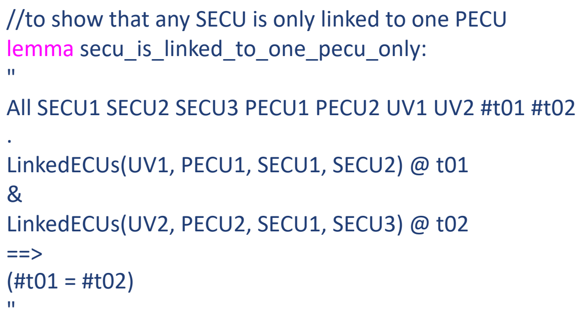
```
// a Tamarin rule expressing that the PECU has aggregated all the reports and now it
sends the manifest to the DS

rule PECU_reports_gathered:
let
payload=<$UV, $PE, vehicle_manifest>
hash_payload=HASH(payload)
sig_payload=sign(hash_payload,~ltk)
full_vehicle_manifest=<HASH(payload), payload, sig_payload>
in
[
  PE_State_02($PE, $UV, $DS, $IS, $TS, ~ltk, image_pe, vehicle_manifest,
vehicle_manifest_size, created)
]
--[
  PECU_Sends_Manifest($PE, $DS, full_vehicle_manifest)
, Eq(vehicle_manifest_size,'1'+'1'+'1')
]->
[
Out_S($PE, $DS, full_vehicle_manifest)
]
```

Tamarin represents protocols using **rules** which have inputs and outputs, creating a transition system (TS).
The TS is subjected to a Dolev-Yao semantics by the tool
Processing of the rules' inputs
Input to a Tamarin rule
Action facts that allow us to reason about the behaviour of the system
Output of a rule

Fig. 4. Example of some `Uptane`'s Logic encoded as a `Tamarin` Rule

SECUs can indeed be linked to a PECU only once, when the system of ECUs of a car is composed (which is captured in our `Tamarin` models).



```
//to show that any SECU is only linked to one PECU
lemma secu_is_linked_to_one_pecu_only:
"
All SECU1 SECU2 SECU3 PECU1 PECU2 UV1 UV2 #t01 #t02
.
LinkedECUs(UV1, PECU1, SECU1, SECU2) @ t01
&
LinkedECUs(UV2, PECU2, SECU1, SECU3) @ t02
==>
(#t01 = #t02)
"
```

Fig. 5. Example of `Uptane`'s Verification in a `Tamarin` Lemma

We modelled more properties and requirements than aforementioned, but – for sake of clarity– we do not explain this here.

Our models are modular, e.g., the security of the channels can be "up"-ed or "down"-ed in the same model, creation and updates of manifest can be interchanged, etc.

However, for the review process to be eased, we artificially created 5 separate `Tamarin` files corresponding to different options (e.g., attacker, channel types). Also, we created `Tamarin` oracles for the more complex models (e.g., strong attackers), as the proofs were intractable without oracles. Even automated with oracles, these proofs take 20+h to run on powerful servers (see next sub-section).

- All our files and replicability instructions can be found at `http://people.itcarlson.com/ioana/files/uptane`.

B. Results on the Security-Verification for `Uptane`

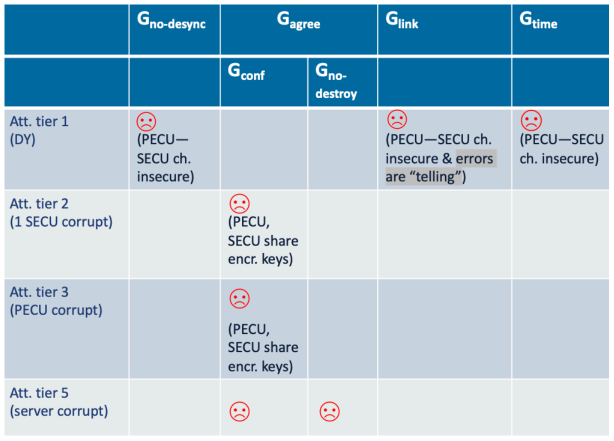
We summarise the results found via all our `Tamarin` models, lemmas and proofs, in our Figure 6. This summary is leveraging our hierarchy of attackers and properties to show only the strongest failures, meaning that properties weaker than the ones shown to fail also fail themselves.

On Figure 6, a "sad face" denotes an attack, whereas no face or no distinctive mark illustrates that no attack found.

The mention in brackets, under the "sad face", gives an explanation of why the lemma failed in `Tamarin`, based on the counterexample trace given by `Tamarin`, but expressed in human-readable language. Note that all these explanations for the attacks found are, in fact, in line with the assumptions by the `Uptane 2.0` standard.

Finally, recall that if a property fails under Tier x of attackers, it also fails on all Tier y of attackers, for $y > x$. I.e., all that fails on a row, also fails on row underneath it. In this spirit, Tier 4 (which combines Tier 2 and Tier 3) is not recounted, since the attacks found at it are already found at Tier 2, and we found no others.

Let us detail on some properties now. Row 1 says that a simple, Dolev-Yao attack can: desynchronise SECUs/PECUs/server, can "link"/tell which SECU has been updated and which not, can desynchronise parties w.r.t. time. Row 2 and 3 say that if the attacker can corrupt PECU, or even SECUs, there is no hope of security amongst the whole network of them, as they become non-authenticated to one another; this is largely due to the fact that PECU and SECUs share encryption keys, as per the `Uptane 2.0` standard. The last row says that if the server is corrupt, then there is no hope of security, irrespective of the above assumption on PECU and SECUs sharing encryption keys.



	$G_{\text{no-desync}}$	G_{agree}		G_{link}	G_{time}
		G_{conf}	$G_{\text{no-destroy}}$		
Att. tier 1 (DY)	☹ (PECU—SECU ch. insecure)			☹ (PECU—SECU ch. insecure & errors are "telling")	☹ (PECU—SECU ch. insecure)
Att. tier 2 (1 SECU corrupt)		☹ (PECU, SECU share encr. keys)			
Att. tier 3 (PECU corrupt)		☹ (PECU, SECU share encr. keys)			
Att. tier 5 (server corrupt)		☹	☹		

Fig. 6. `Uptane`'s Security and Privacy as Verified in `Tamarin`

The thing to underline here is not that we found the above per se, but that we found it using formal-method tools, with systematic modelling which can be re-used and augmented for continuous verification of the standard.

*a) **Verification Statistics:*** Our models have on average cca. 5000 lines of code each. They are written in in `Tamarin` version 1.6.0. We executed the models on a server with 2 Intel Xeon E5-2667 CPUs (16 cores, 32 threads) and 378GB of RAM. The timings of executions of proofs of lemmas on models vary from seconds, to 20+h, to non-terminating. These details can be found on our repository: `http://people.itcarlson.com/ioana/files/uptane`

C. Disclosure &Recommendations

We disclosed this to the `Uptane` Alliance.

Given the findings above, we recommended the following to the `Uptane` Alliance:

- Channels between ECUs should be encrypted and authenticated;
- ECUs should not share encryption keys;
- Error handling should be better specified and error codes should be hidden.

We are working with the `Uptane` Alliance to make amends to the standard.

V. Discussions

In future work, we would like to:

- take this methodology used on `Uptane 2.0` and apply it to `Uptane 2.1` which is to be released in June 2023 (i.e., after the time of this writing);
- test an implementation of `Uptane`, using – of course– different methodology, suited to program (not systems) analysis;
- lift our model as well as work with the `Uptane` Alliance to lift the standard itself to support hierarchies of SECUs and refined control; this means to model cars more faithfully, where SECUs are partitioned in levels, SECUs of level 1, SECUs of level 2, etc., depending on their criticality. Also channels (buses) in between the SECUs themselves and the SECUs and the PECU are more or less secure (e.g., with/without integrity protection) based on these levels. Then, we would like to check security in this hierarchical context: i.e., if an attacker corrupts a SECU of level 2, can it compromise the whole system?
- use the above to build a risk model for safety and security, in the way that UNECE WP.29 regulation (`https://unece.org/wp29-introduction`) demands.

VI. Related Work

Possibly the "reference paper" on `Uptane` is [10] which describes it even before the standard documents were as they are today. A simplistic security analysis using the old-fashioned CSP formalism can be found at [2]; no vulnerabilities of interest are reported. Non-academic security reviews have been carried out by the `Uptane` Alliance [11], and there is one line [12] on supply-chain attacks.

No prior work has done an as-systematic or as-formal analysis as we undertake here.

VII. Conclusions

In this paper, we showed the formal security analysis of `Uptane 2.0` – the latest framework for *over-the-air (OTA)* (online) delivery of software to cars. We did so by using the threat model and security requirements found in standard document that accompanies `Uptane 2.0`, as well as a modulation of this threat model and requirements added by ourselves, for a deeper analysis. To undertake this verification, we used the well-known formal protocol-verifier and theorem prover called `Tamarin`. We found certain shortcomings and we made recommendations to the `Uptane` Alliance, and are working with them on edits to the standard.

Acknowledgements

We thank Thales for making us aware of `Uptane` as part of a joint research project called AutoPaSS, funded by the EPSRC as grant EP/S024565/1. We thank Fortunat Rajaona and Steve Wesemeyer for initial discussions and modelling of `Uptane`, respectively.

References

[1] Uptane Standards Group, "Uptane standard for design and implementation 2.0.0," *Uptane*, accessed: 2022-06-24. [Online]. Available: https://uptane.github.io/papers/uptane-standard.2.0.0.html

[2] R. Kirk, H. N. Nguyen, J. Bryans, S. Shaikh, D. Evans, and D. Price, "Formalising uptane in csp for security testing," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2021, pp. 816–824.

[3] A. Armando, D. Basin, Y. Boichut, and et al., "The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications," in *CAV*, 2005.

[4] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *CAV*, 2013, pp. 696–701.

[5] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *IEEE CSFW*, 2001.

[6] D. Dolev and A. Yao, "On the Security of Public-Key Protocols," *IEEE Transactionson Information Theory 29*, vol. 29, no. 2, 1983.

[7] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A Formal Analysis of 5G Authentication," in *CCS*, 2018, p. 1383–1396.

[8] I. Filimonov, R. Horne, S. Mauw, and Z. Smith, "Breaking unlinkability of the icao 9303 standard for e-passports using bisimilarity," in *ESORICS*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., 2019, pp. 577–594.

[9] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in *SP*, 2016.

[10] T. Karthik, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos, "Uptane: Securing software updates for automobiles," in *International Conference on Embedded Security in Car*, 2016, pp. 1–11.

[11] Uptane Standards Group, "Uptane: Securing delivery of software updates for ground vehicles," accessed: 2022-06-24. [Online]. Available: https://uptane.github.io/papers/uptane_first_whitepaper_7821.pdf

[12] M. Moore, A. S. A. Yelgundhalli, T. K. Kuppusamy, S. Torres-Arias, L. A. DeLong, and J. Cappos, "Scudo: A proposal for resolving software supply chain insecurities in vehicles," accessed: 2022-06-24. [Online]. Available: https://uptane.github.io/papers/scudo-whitepaper.pdf