Trusted Verification of Over-the-Air (OTA) Secure Software Updates on COTS Embedded Systems

Anway Mukherjee, Ryan Gerdes and Tam Chantem
Department of Electrical and Computer Engineering, Virginia Tech, USA.
Email: {anwaym, rgerdes, tchantem} @vt.edu

***Abstract*—Over-the-air (OTA) software updates are an important feature to remotely analyze and upgrade any section of currently running software on battery-operated electric vehicles and its supply equipment. Even though a secure OTA framework can verify and validate updates before installation, the integrity of the framework itself cannot be guaranteed, and can easily introduce system and software vulnerability with potential catastrophic consequences. In this paper, we show how a popular automotive OTA secure update framework (Uptane) can be deployed entirely inside a TEE-enabled commercial off-the-shelf (COTS) embedded device to extend its security considerations and improve its resilience against both internal and external security breaches. We also present a software analysis tool that leverages SAWScript to verify our proposed solution against any functional and logical inconsistency, while validating our approach on a real COTS hardware (Raspberry Pi 3B).**

I. Introduction

With the increase in popularity of smart infrastructure, the complex devices that interconnect them all to form the internet of things (IoTs) lacks a holistic framework that delivers visibility, segmentation, and protection throughout the entire network infrastructure (1) secured to prevent unauthorized access, and (2) support internal isolation to ensure execution correctness. For example, battery electric vehicles (BEVs) [1], operating on a battery management system (BMS), require periodic charging performed by specialized electric vehicle supply equipment (EVSE) deployed at designated BEV charging stations or residences [2]. The EVSEs are smart internet of thing (IoT) devices that can connect the BEVs to an available power source, with support for both grid-to-vehicles (G2V), and a reverse vehicle-to-grid (V2G) energy transactions. However, BEVs and EVSEs can be easily manipulated, individually or collectively, by adversaries to gain access and act as conduits to breach their internal security, leading to malicious attacks and life-threatening situations, for instance, traffic accidents [3] in BEVs, and disrupt or cause power systems failure through high or low voltages on feeder circuits in EVSEs [4].

Both BEVs and EVSEs are prone to external interference as well as internal software vulnerabilities. EVSEs are also limited by their size, weight and power (SWaP) constraints and network capacity, making them more vulnerable to attacks compared to other embedded and general purpose systems [3]. Over the air (OTA) updates is a popular interface to facilitate security and protection for said devices [5], [6]. OTA frameworks assist in remote diagnosis, and upgrades if required, for any reported security breach in compromised EVSEs and BEVs without having to deal with expensive in-person human attendance to fix the issue(s). While an OTA software update framework facilitates security updates, along with operational upgrades, it can also introduce unwanted security loopholes. For instance, in an autonomous BEV, nearly all driving operations (for e.g., where to turn, when to brake, controlling the speed of the car etc.) are automated, while specific manual cognitive decision-making control is left to the discretion of the driver. Thus, if a corrupt OTA update package which contains functional upgrades for the car is installed, an external agent can easily piggyback on the corrupt package to install malicious programs that will compromise the entire BEV thereby leading to catastrophic consequences. Note that even though we motivate our work with specific examples of BEVs and EVSEs for the rest of the paper, our approach can easily be applied to improve the security specifications of any existing system that utilizes secure OTA update framework running on TEE-compatible COTS embedded system.

OTA updates in EVSEs and BEVs, therefore, not only need to be (1) secure to prevent unauthorized access, but also (2) support internal software isolation to ensure functional correctness. Protecting and encrypting the communication channel and checking the authenticity of the new software are longstanding and vital security measures for any BEVs and EVSE devices. However, said security measures are inconsequential if the OTA update verification framework itself is compromised [7], [8]. Therefore, there is a need to tackle such a scenario, and design a framework that can detect and protect against such security breaches. Existing approaches to protect the OTA update verification framework include dedicated secure hardware-based solutions (e.g. trusted platform modules (TPMs), secure networking modules, secure grid etc.) [3], [9], [10]. They all require expensive, custom-built hardware with long time-to-market or time-to-deployment cycle. A readily available cheaper alternative is the use of trusted execution environment (TEE) [11] on commercial off-the-shelf (COTS) embedded processors. Examples of hardware support for TEE include the ARM TrustZone [12] which is popular in embedded devices. TEE leverages hardware security extensions to provide platform virtualization to run an application in secure isolation from the rest of the system. TEE can be quickly redeployed if an exploit is found since they are implemented using platform virtualization. Hence, TEE does not require

Workshop on Automotive and Autonomous Vehicle Security (AutoSec) 2021
25 February 2021, Virtual
ISBN 1-891562-68-1
https://dx.doi.org/10.14722/autosec.2021.23028
www.ndss-symposium.org

hardware redesign. In each platform-specific OS that supports TEE (for e.g., ARM TrustZone), an instance of TEE execution is initiated by a setup context, followed by the actual trusted execution inside an isolated environment, and exits through a destroy context. TEE is utilized by partitioning the code such that the portion that requires secure execution is run inside the TEE context and communicates with the rest of the partitioned code via a software-controlled exceptions called secure monitor calls (SMCs) [13].

This paper proposes a new secure software update framework that ensures that secure OTA updates, along with the entire OTA verification framework, can be deployed and serviced inside a trusted secure environment fully protected from any external or internal attacks. For proof of concept, we utilize the Uptane [6] framework to model a secure OTA update and verification framework in BEVs and EVSEs. Uptane is a secure open-source industrial software update standard for automobiles. The framework services protection against malicious attackers by signing and delivering software updates through a secure communication channel in EVSEs and BEVs. Our proposed solution isolates and deploys the entire OTA verification framework inside TEE to protect its integrity and avert any security breaches. We also formally verify our proposed solution framework by leveraging an open-source software analysis workbench (SAW) developed by Galois [14]. SAW is an essential tool to analyze semantic models of programs, and check for any software vulnerability that can potentially lead to security breaches. Our main contributions are as follows.

1) We leverage ARM TrustZone to present a novel design that aims to strengthen the security of OTA update and verification framework. Our approach proposes to migrate the entire OTA update and verification framework inside a TEE, and isolate all of its standard functionality within the scope of trusted execution for improved protection in BEVs and EVSEs.
2) We leverage SAW tool to propose a technique which can verify and validate our proposed framework to guarantee that our software implementation does not lead to any security vulnerability.
3) We validate our approach on a real hardware (Raspberry Pi 3B) using ARM TrustZone that runs OP-TEE, an open source trusted OS, side-by-side with real-time Linux `RT_PREEMPT`.

II. Preliminaries

A. Uptane

Uptane [6] is a popular open source secure software update framework that aims to protect software updates that are packaged by the OEM and delivered over-the-air to its recipients through secure wireless communication channels. This framework is the standard for secure updates in the automotive industry, and is designed to successfully prevent a variety of malicious state-of-the-art attacks. A standard open-source implementation of the Uptane framework can be divided into two broad categories; (1) the secure server representing the OEM, and (2) the client representing the recipient. A secure communication channel, for e.g., `https`, is used for data communication between the server and its client(s).

The secure server-side implementation consists of three sub-modules. First, the *images* repository stores the actual image package, along with related metadata which are securely signed by a private key. Second, the *director* sub-module validates vehicle manifests, and generates the vehicle-specific update images and metadata. Finally, the *timeserver* sub-module generates timestamp tokens to validate that the most recent update package is being delivered to the client. The client-side implementation similarly consists of multiple sub-modules. The *primary* client sub-module generates vehicle version manifest(s) with vehicle metadata and sends it back to the server-side director. A *manifest* consists of information pertaining to client-side system-wide status, which is analysed by the director, before generating any update package to redress detected vulnerabilities in the client. The primary client also maintains the update cycle by fetching and validating all signed images, and its metadata, from the director. It also queries the timeserver for signed attestation. The validated metadata in the client is stored in a *local* metadata staging sub-module. Finally, a *secondary* client fetches and validates all signed metadata from the primary before installing the update package(s) on to the system.

B. ARM TrustZone

The TrustZone [11] is ARM's open-source implementation of trusted execution environment for COTS embedded systems. It provides a secure virtualization platform where code that requires trusted execution can be executed in isolation from the entire system without loss of integrity. Mukherjee et. al. [13] details the architectural framework of a TrustZone consisting of side-by-side deployment of a normal execution environment (running the non-trusted OS), and secure execution environment (running the trusted OS). In non-trusted execution environment, the non-secure application(s) or code segments that do not require TEE execution, run on standard embedded hardware. In a trusted environment, all code and/or application(s) that require trusted execution have their data are stored in, and executed in isolation on specially augmented secure hardware components (e.g., CPU, memory, and peripherals). Signal and data communication between the two environments (TEE and non-TEE) is performed through a platform-specific secure message passing protocol. TEE delimits the code running in normal world from (1) changing the trusted OS system configurations, and (2) protect secure environment data/code from being accessed by the non-trusted software. This means that if the non-trusted execution environment and OS is compromised, the attack is confined to the access privileges of only the non-trusted OS.

C. SAW Verification Tool

The software analysis workbench (SAW) [14] provides the ability to formally verify any application code and identify

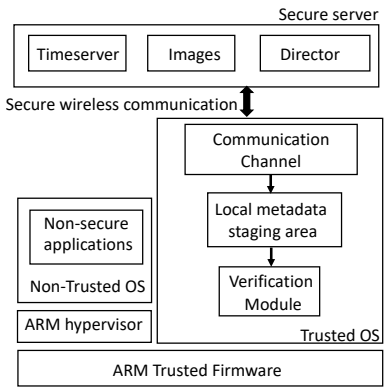

Fig. 1. An overview of our proposed update framework which is decoupled into (1) a secure Uptane server, and (2) a modified Uptane client-side implementation to run as trusted application(s) within the trusted OS inside TEE. The client consists of a communication channel and a metadata staging area, and services client-specific functionalities.

any software-level security vulnerabilities of said code. SAW provides a standard developmental kit (SDK) that can leverage a number of programming language-specific tools and solvers to automate the process of analysis and verification of complex software frameworks. Specifically, SAW utilizes symbolic execution to translate code into formal models, execute code on symbolic inputs, effectively unrolling loops, and translating the code into a circuit representation for cryptographic verification. As a verification tool, SAW provides a level of assurance beyond the capabilities of traditional frameworks since SAW is capable of performing an exhaustive testing of target applications against all combinations of input parameters. In this work, we leverage the SAW tool developed by Galois that uses Cryptol scripting language [15] for the high-level analysis and verification of target software.

III. Proposed Framework

A. Threat Model

While our proposed framework can be seamlessly integrated into any existing system that utilizes secure OTA update framework running on TEE-compatible COTS embedded system, we motivate our work by describing a setup where a standard secure update framework can lead to an internal security breach. For example, consider a BEV-EVSE collective IoT system. We now argue why it is not nearly enough to just have a secure OTA update framework in place inside an EVSE or a BEV. We consider the BEV with nearly all automated driving operations controlled by the on-board network of embedded devices. We also consider a setup where the BEV provides periodic V2G reverse grid charging through EVSE that forms a part of the broader IoT network. Therefore, the OTA update framework (1) acts as a client to acquire important updates from a server (or OEM) through a secure update process for the on-board embedded systems in the BEVs and EVSEs, and (2) directs the BEV/EVSE to download the update packages into its system for installation after in-situ rigorous verification and validation process. Here, we assume that the BEV or the EVSE has been compromised and an attacker has control over its internal software and data. The attacker can now easily manipulate the integrity of any of the following; (1) OTA verification framework, (2) the actual software update package, and (3) its signatures that were previously downloaded and stored in the BEV/EVSE from a secure server for verification and validation. A malicious update package, therefore, can easily be pushed into, and installed in the BEV/EVSE. The attacker can easily gain access to all the automotive features inside BEV leading to a range of catastrophic scenarios. Similarly, a corrupt EVSE can lead to charging termination (e.g., in smart grids), and impact the overall voltage and frequency limits of a system leading to energy-related vulnerabilities. Here we assume that the secure server (OEM) can maintain its own integrity. However, both the EVSE and the BEV, and its OTA update framework requires extended security to combat such threats.

B. Mitigation Strategy

We consider a TEE-enabled system for both EVSE and BEV which can (1) protect the integrity of an OTA update verification framework by running it within a TrustZone in isolation from the rest of the system, and (2) securely store update packages (both software and data) within a trusted storage. Implementing a verification and validation framework for OTA software updates within a secure execution environment has several advantages. A secure environment helps us isolate the update framework from the rest of the system without breaking the backward compatibility. It also gives us the flexibility to seamlessly deploy a trusted execution framework for any commercial off-the-shelf (COTS) embedded system, and use it for data protection over OTA channels without compromising the standard security requirements. Moreover, since the entire OTA update framework runs in seclusion inside a trusted OS, any internal or external attack on the rest of the software or system does not affect the integrity of the isolated OTA framework. In this work, we design (1) a trusted framework using TEE that provides an secure execution environment within which the standard Uptane verification framework operates, (2) instrument and re-factor the Uptane framework to integrate it within the TEE environment, and (3) create a communication gateway for secure data transfer between the Uptane OTA framework and a designated remote secure OEM server.

Figure 1 shows an example overview of our proposed system model inside BEV/EVSE. Since Uptane's software update process is based on a publish-subscribe model, we divide our work into three parts; (1) an Uptane-compliant secure server which complies with the Uptane standards, (2) a TEE-specific client-side implementation (inside BEV/EVSE) that includes all functional components that perform software and data exchange between the client sub-modules and the secure server, and (3) a secure standard communication protocol that services the client requests, and implements a co-ordinated data transfer between the server and the client. The client-side modifications ensure a seamless integration of the Uptane standard policies within the TEE framework. Since, it is the programmer's responsibility to ensure that the application

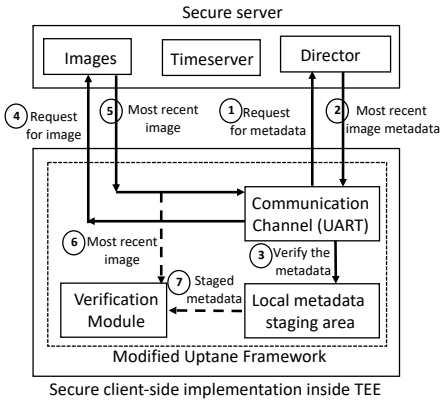

Fig. 2. An detailed overview of our proposed client-side implementation. The numbers associated with each arrow capture the chronological exchange of data between the secure server and the client-side sub-modules running within the TEE.

code running inside TEE do not inadvertently introduce any software-related security vulnerabilities, we formally verify our client-side implementation using the SAW tool to analyse and check for security vulnerabilities (Section V).

Our solution is based on the following preconditions which do not violate the Uptane standard verification policy. First, we assume that there is a secure server in place, which is compatible with the existing Uptane verification standards. Therefore, we keep Uptane's standard server design in tact. Second, we assign the client with the ability to establish connection to a verified server. We assume that the client is programmable, and can initiate an update cycle triggered periodically by a hardware interrupt subroutine. Finally, we attribute the client with the ability to perform a public key cryptography operation, as well as verification and validation of images supplied by a secure server within the guidelines supplied by the Uptane standards.

IV. TEE-enabled Uptane Client

A client, as part of the BEV/EVSE software module, services all the functionalities to verify and validate an Uptane-specifc OTA update framework. The client is deployed inside a TEE for isolated execution. An overview of our proposed solution is shown in Figure 2. We split our work into a two-step procedure; a client-side implementation within a TEE, and a secure server. The client-side implementation (1) issues request for most recent update for the secure server, along with (2) verification and validation of the its metadata before (3) securely installing the update(s) to the system. The secure server is responsible for servicing the update-specific images and all the relevant metadata when requested by the client-side. As previously mentioned, we maintain the functional and logical guidelines in our implementation as recommended by the Uptane standards. First, the client initiates an update cycle through a hardware interrupt. The client sends a request to a secure server to obtain the metadata for the most recent update available for downloading. Second, the secure server sends the requested update-specific metadata back to the client. The client stages the metadata for future validation. Next, the client sends another request to download the actual update image. Once the image has been downloaded from the secure server, the client matches the previously staged metadata with the metadata that is included in the actual update image. Finally, after successful verification, the client can (1) update the system with the downloaded image, and (2) update its local records with the most recently installed update. Next, we will describe in details the various sub-modules of our client-side implementation.

A. Communication Channel

This sub-module provides a communication channel between the server and client in the Uptane framework. In our implementation, we use a protocol similar to the unified diagnostic services (UDS) to service the communication layer. UDS is a diagnostic communication protocol popularly used in the automotive industry. Our communication channel inside the client mediates requests to and responses from the secure server, and services update-specific metadata for verification, download images for system update, and handles queries for a signed attestation of the timestamp for the downloaded images. Similar to the UDS protocol, each feature is associated with a service ID (SID) and the information and or data associated with the services are encapsulated in the payload of the communication message frame. A list of SIDs and their functionalities is described in Table I. For instance, a download metadata request message to secure server from the client is associated with an SID equal to 4, and the payload associated with the request contains various information including the the size of download, the location of the metadata in the server, and the type of metadata requested. In contrast, the SID associated with image download request message is 5. Each request is followed by a response message with a positive or negative validation. The associated payload with the response contains the requested metadata in case of successful message reception at the server, or a negative error code if the request message failed to comply with the SID-specific payload configuration. The SIDs distinguish the different message requests (and responses) between the server and the client.

For our physical medium of communication, we use a UART connection between the secure server and the client-side implementation. This sub-module, therefore, sets up a UART connection and establishes a communication channel to send and receive messages along with essential data between the server and the client. Our design also includes a UART-to-wifi proxy implementation that can support wireless communication for backward compatibility.

B. Local Metadata Staging Area

This sub-module contains a local list of Uptane-compliant files that constitutes the update-specific metadata of the images that have been downloaded from a secure server. Each metadata file can be categorized into four sub-classes based on their roles in the client-side update and verification framework. The following are the categories of metadata files; (1) Root

TABLE I
EXAMPLE SIDS FOR TARGET REQUESTS AND THEIR CORRESPONDING RESPONSES (YY = REQUEST PAYLOAD, ZZ = RESPONSE PAYLOAD)

SID	Request	Description	Positive Response
1	1 YY YY YY	Request to start update	11 ZZ ZZ ZZ
2	2 YY YY YY	Request to stop update	12 ZZ ZZ ZZ
3	3 YY YY YY	Request timestamp attestation	13 ZZ ZZ ZZ
4	4 YY YY YY	Request metadata download	14 ZZ ZZ ZZ
5	5 YY YY YY	Request image download	15 ZZ ZZ ZZ
6	6 YY YY YY	Request transfer data	16 ZZ ZZ ZZ
7	7 YY YY YY	Request transfer exit	17 ZZ ZZ ZZ

metadata is used to verify the metadata associated with all the other sub-classes, (2) Target metadata is used to verify an image. It also contains information on the server-specific metadata, including the current image version number, (3) Snapshot contains the signed metadata for all the other sub-classes, and (4) Timestamp metadata indicates whether the downloaded images and metadata are new and current.

C. Verification Module

If the client does not have latest image, it enters an update cycle through an interrupt procedure call. The local metadata staging area stores a local copy of the metadata of previously installed images in the client secure storage. This sub-module performs the step-by-step process of downloading the most recent image from the secure server to the client and verifying the image before it can be installed into the system.

Download Latest Image: The client first downloads the target metadata from the server. The information contains the filename used to identify the latest known image. If there is no target metadata about this image, we abort the update cycle. Additionally, in the case of failure, the client retains its previous target metadata instead of using the newly downloaded target metadata. Otherwise, we download the image, and verify (described in the section below) that it matches the target metadata. Finally, once the update cycle is completed, the client overwrites the latest image specific metadata to its local copy in the secure storage.

Verify Latest Image: The client verifies and validates that the latest image matches the latest metadata. The first step in the process is to download the latest image specific metadata from the server. The first step of validation includes checking that the client identifier in the metadata matches the actual client identifier, followed by a valid image filename. The next step is to check that the latest image version counter is greater than the previously installed image in the client-side. Then we decrypt the image metadata, followed by a hash checking between the downloaded image and the downloaded metadata in the staging area. If at any time this verification step fails to validate the image, the client aborts the update cycle, and waits for the next update trigger.

V. SOFTWARE ANALYSIS

Heretofore, we have successfully modified the Uptane client to run entirely in isolation inside a TEE to protect it from any external attacks. However, it does not guarantee that the client-side implementation of the trusted code running within TEE does not introduce any new vulnerabilities. Moreover, Uptane documentation provides a list of rules that must be followed to ensure that an Uptane client (and the update process) complies with its set standards. Therefore, we leverage the software analysis workbench (SAW), a popular software analysis tool developed at Galois, to formally verify properties of our implementation code. As an analysis tool, SAW is capable of testing a program against an exhaustive list of all input parameters, and can efficiently capture corner-cases when the application code will fail to comply with its functional and logical requirements. In this work, we have utilized SAW (1) to detect any logical and programming correctness with our Uptane-compliant trusted client application that runs within TEE, and (2) analyze client-specific functionalities that largely utilize cryptographic algorithms (e.g., SHA and ECDSA) to validate and verify downloaded update image(s) and their metadata.

We now demonstrate how we use SAW tool to prove the equivalence of the reference and implementation versions of the members of our image metadata, and detect any bug in an incorrect implementation. Our verification is structured as a sequence of commands, potentially along with definitions of functions that abstract over commonly used combinations of commands. Consider a part of the target client code, a `memcpy()` module that needs to be analyzed and verified against any logical or functional irregularities. SAW tool provides a library of reference implementation that can be used to compare with our target implementation of `memcpy()` to identify any existing bug in the code. Figure 3 shows the SAWScript that can automatically leverage symbolic execution to translate the target code into formal models to compare, analyze and verify its robustness against any security vulnerabilities. For instance, The `LLVM_extract` command instructs the SAW tool interpreter to perform symbolic simulation of our target function, and return the semantics of the function. The `let` statement then constructs a new term corresponding to the assertion of equality between two existing terms. The `prove` command verifies the validity of our assertion, or produce a counter-example that invalidates it. The parameter `thm` indicates the theorem solver that has been used in our target verification process.

Specifically, the SAW tool ensures that the software running inside the TEE complies with the standards outlined by the Uptane framework. Our implementation draws upon the standard `C` programming resources to realize the OTA framework inside TEE, primarily since TEE only supports a limited range of `C` headers and functions. Therefore, we utilize SAW tool to evaluate the memory safety assumptions (length of buffer, data integrity, data overflow etc.) for associated data schemas specific to our implementation. We also utilize the cryptographic algorithm analysis framework included with SAW tool to validate the Uptane client hashing and signature verification functions, including the compositional verification of said algorithmic functions. SAWScript leverages its embedded Cryptol programming tool to service any high-level cryptographic specifications.

```
#Extracting reference term: memcpy_ref
l <- llvm_load_module "memcpy.bc";
memcpy_ref <- llvm_extract l "memcpy_ref";

#Extracting implementation term: uptane_memcpy_imp
uptane_memcpy_imp <- llvm_extract l "uptane_memcpy_imp";


#Proving equivalence: memcpy_ref == uptane_memcpy_imp
let thm1 = {{ \x -> memcpy_ref x == uptane_memcpy_imp x }};
result <- prove abc thm1;
print result;
```

Fig. 3. An example SAWScript to analyze a target application code `uptane_memcpy_imp`, and prove its equivalence when compared against the SAW tool reference implementation `memcpy_ref`.

Fig. 4. Application log of Uptane client running on our experiment testbed. Results indicate that the OTA update requests are triggered periodically and the subsequent responses with image and metadata download from the server through a secure UART communication.

VI. Experiments

To validate our proposed approach in a realistic setting, we selected Uptane as the representative secure software update framework. We use the Raspberry Pi 3B (Rpi3B) as our example TEE-enabled embedded testbench that represents a TEE-enabled COTS-based automotive BEV and EVSE. Rpi3B is a small computer powered by Broadcom BCM2837 chipset consisting of quad-core ARM Cortex A53 processor, 1 GB LPDDR2 RAM, and numerous sensors (for eg., Barometer, IMU etc.). It can run Linux and other non-trusted operating systems. It also extends support for ARM TrustZone. We used the Linux `RT_PREEMPT` Kernel v4.6.3 as our non-trusted real-time OS, along with OP-TEE OS (as TEE) on the Rpi 3B that extends the support for ARM v8 embedded virtualization. Our modified Uptane client application is included as part of the trusted application suite and runs completely inside OP-TEE OS. An off-board Uptane-compliant secure server stores and services the update image and metadata to the client. The data transfer and communication between the server and the client is performed over a secure UART. Figure 4 shows an example update sequence at the Uptane client-side starting with (1) the client triggering a request to the secure server, followed by (2) timestamp attestation and verification, (3) metadata download request and validation by the client, and the subsequent (4) retrieval of update image downloaded from the secure server and ready for validation before installation.

VII. Conclusion

We present a secure design framework that leverages ARM TrustZone to isolate and execute the Uptane OTA update system within EVSEs/BEVs, thereby improving the security of such mission-critical systems against both external and internal system vulnerabilities. Our solution also includes a software analysis and verification tool that utilizes SAWScript to automatically validates our trusted application code against any programming, functional and logical defects that can gravely jeopardize the overall security of the system. The proposed approach was validated on an actual hardware testbed (Rpi3B) that runs the client code in isolation within TEE.

References

[1] A. M. Andwari, A. Pesiridis, S. Rajoo, R. Martinez-Botas, and V. Esfahanian, "A review of battery electric vehicle technology and readiness levels," *Renewable and Sustainable Energy Reviews*, vol. 78, pp. 414–430, 2017.

[2] Y. Huang and K. M. Kockelman, "Electric vehicle charging station locations: Elastic demand, station congestion, and network equilibrium," *Transportation Research Part D: Transport and Environment*, vol. 78, p. 102179, 2020.

[3] A. Fuchs, D. Kern, C. Krauß, and M. Zhdanova, "Securing electric vehicle charging systems through component binding," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2020, pp. 387–401.

[4] J. T. Johnson, "Securing vehicle charging infrastructure." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2019.

[5] D. K. Nilsson and U. E. Larson, "Secure firmware updates over the air in intelligent vehicles," in *ICC Workshops-2008 IEEE International Conference on Communications Workshops*. IEEE, 2008, pp. 380–384.

[6] T. K. Kuppusamy, L. A. DeLong, and J. Cappos, "Uptane: Security and customizability of software updates for vehicles," *ieee vehicular technology magazine*, vol. 13, no. 1, pp. 66–73, 2018.

[7] M. Tullsen, L. Pike, N. Collins, and A. Tomb, "Formal verification of a vehicle-to-vehicle (v2v) messaging system," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 413–429.

[8] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained iot devices using open standards: A reality check," *IEEE Access*, vol. 7, pp. 71 907–71 920, 2019.

[9] G. Macher, A. Höller, H. Sporer, E. Armengaud, and C. Kreiner, "A combined safety-hazards and security-threat analysis method for automotive systems," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2014, pp. 237–250.

[10] X. Wei, C. Du, and J. Zhao, "A network security situation awareness model for electric vehicle shared charging pile system," in *AIP Conference Proceedings*, vol. 2238, no. 1. AIP Publishing LLC, 2020, p. 020009.

[11] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1. IEEE, 2015, pp. 57–64.

[12] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[13] A. Mukherjee, T. Mishra, T. Chantem, N. Fisher, and R. Gerdes, "Optimized trusted execution for hard real-time applications on cots processors," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, 2019, pp. 50–60.

[14] K. Carter, A. Foltzer, J. Hendrix, B. Huffman, and A. Tomb, "Saw: the software analysis workbench," in *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*, 2013, pp. 15–18.

[15] J. R. Lewis and B. Martin, "Cryptol: High assurance, retargetable crypto development and validation," in *IEEE Military Communications Conference, 2003. MILCOM 2003.*, vol. 2. IEEE, 2003, pp. 820–825.