

Secure Over-the-Air Vehicle Updates using Trusted Execution Environments (TEE)

[Augusto Henriques](#)

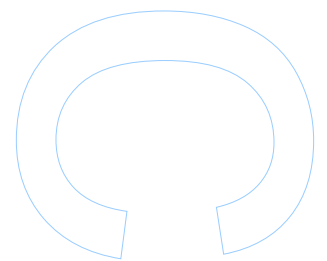
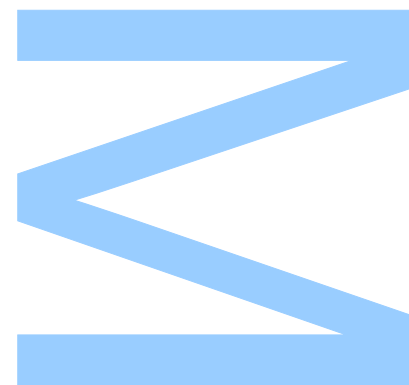
Mestrado Engenharia Redes e Sistemas Informáticos
[Departamento de Ciências de computadores](#)
2022

Orientador

[Prof. Dr. Hugo Pacheco](#), Faculdade de Ciências

Coorientador

[Fernando Alves](#), VORTEX-CoLab



Declaração de Honra

Eu, Augusto César Pereira Henriques, inscrito(a) no Mestrado em Engenharia Redes e Sistemas Informáticos da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação/relatório de estágio/projeto Secure Over-the-Air Vehicle Updates using Trusted Execution Environments (TEE) reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta dissertação/relatório de estágio/projeto Secure Over-the-Air Vehicle Updates using Trusted Execution Environments (TEE), declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação/relatório de estágio/projeto Secure Over-the-Air Vehicle Updates using Trusted Execution Environments (TEE) quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Augusto César Pereira Henriques

30 de setembro, 2022

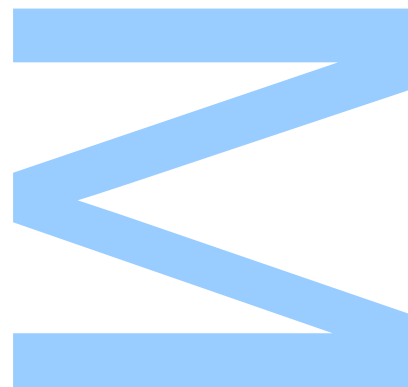
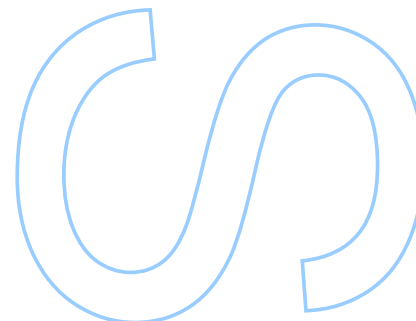
U. PORTO

FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ____/____/____

A large, bold, blue stylized graphic element consisting of a horizontal bar at the top, a diagonal line sloping down to the right, a horizontal bar in the middle, a diagonal line sloping down to the left, and another horizontal bar at the bottom. It resembles a stylized 'N' or a signature.A large, blue, stylized signature or stamp consisting of a continuous, flowing line that forms a shape resembling a cursive 'S' or a similar decorative flourish.A large, blue, stylized signature or stamp consisting of a continuous, flowing line that forms a shape resembling a cursive 'Q' or a similar decorative flourish.

UNIVERSIDADE DO PORTO

MASTERS THESIS

**Secure Over-the-Air Vehicle Updates using
Trusted Execution Environments (TEE)**

Author:

Augusto HENRIQUES

Supervisor:

Hugo PACHECO

*A thesis submitted in fulfilment of the requirements
for the degree of MSc. Engineering Network Engineering and Computer Systems*

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciências de Computadores

September 30, 2022

“ I am and always will be the optimist, the hoper of far-flung hopes and the dreamer of improbable dreams ”

Matt Smith as *The Doctor*, written by Matthew Graham

Acknowledgements

I would like to thank my supervisors, Professor Hugo Pacheco and Doctor Fernando Alves, for all incredible commitment to help and time spent guiding me throughout this dissertation. Finally I would like to thank my family and friends for giving me everything I needed to reach this stage of my life.

Abstract

Over-the-air (OTA) software update system has emerged as an important feature to remotely analyze and upgrade the vehicle inside systems, coordinated by different ECUs (Electronic Control Unit). Inside vehicle architecture, there exists two types of ECU (Primary and Secondary ECU) with specific functions that allow a efficient, cost-effective and convenient workflow mechanism in the vehicle system.

With the rising popularity of the OTA updates in automotive industries it introduces new security challenges requiring immediate precaution, as attackers can gain control of the vehicle through these update systems to undermine vehicle security, and in the worst case, putting in danger the driver's life.

During this journey, we developed a new secure design framework using Uptane framework implementation combined with Trusted Execution Environment (TEE) technology. We leverage Intel SGX to isolate and execute the Verification method performed on Primary ECU, thereby improving the security of Secondary ECUs that perform Partial verification by relying on remote attestation provided by the Primary ECU to verify if the downloaded update is genuine or not.

Resumo

O sistema de actualização de software Over-The-Air (OTA) surgiu como uma característica importante para analisar e actualizar remotamente o veículo dentro dos sistemas, coordenado por diferentes ECUs (Electronic Control Unit) no interior da arquitectura do veículo, existem dois tipos de ECU (Primário e Secundário) com funções específicas que permitem um mecanismo de fluxo de trabalho eficiente, rentável e conveniente no sistema do veículo.

Com a crescente popularidade das actualizações OTA na indústria automóvel, introduz novos desafios de segurança que requerem precaução imediata, uma vez que os atacantes podem ganhar controlo do veículo através destes sistemas de actualização para perturbar a segurança do veículo, e no pior dos casos, por em perigo a vida do condutor.

No decorrer deste projeto, desenvolvemos uma nova estrutura de concepção segura utilizando a implementação da estrutura Uptane combinada com a tecnologia Trusted Execution Environment (TEE). Utilizamos a tecnologia Intel SGX para isolar e executar o método de Verificação realizado na ECU Primária, melhorando assim a segurança das ECU Secundárias que realizam Verificação parcial, confiando na certificação remota fornecida pela ECU primária para verificar se a actualização descarregada é genuína ou não.

Contents

Acknowledgements	v
Abstract	vii
Resumo	ix
Contents	xi
List of Figures	xiii
Glossary	xv
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Goals	3
1.3 Document Structure	3
2 Background and Related Work	5
2.1 Automotive context	5
2.1.1 OTA process	6
2.1.2 TUF	7
2.1.3 Uptane	8
2.1.3.1 Uptane process	8
2.2 Trusted Execution Environment	9
2.2.1 ARM Trustzone	10
2.2.2 Intel SGX	11
2.2.3 Other solutions	12
2.3 Related work	13
3 Extending the Uptane trust-chain using TEE	17
3.1 Standard Uptane client	17
3.1.1 Roles on repositories	17
3.1.2 Types of metadata	18
3.1.3 Server implementation	18
3.1.4 Client-side	20
3.1.4.1 What the Primary ECU does	20
3.1.4.2 Full Verification	21

3.1.4.3	Partial Verification	22
3.1.5	Uptane implementation	22
3.1.5.1	Protocols	22
3.1.5.2	Message Handler	23
3.1.5.3	Operations	23
3.1.5.4	Metadata	24
3.1.5.5	Usage	25
3.1.5.6	Data	25
3.2	Extending Uptane security model using TEE	25
3.3	Extending Uptane client using TEE	29
3.3.1	Refactoring Uptane	29
3.3.2	Isolate the VS within TEE	31
3.3.3	Integrate remote attestation into Secondaries ECUs	32
4	Implementation	33
4.1	Why SGX?	33
4.2	Why we used Gramine?	34
4.3	Remote Attestation	37
4.4	Why we used GRPC?	39
5	Evaluation	43
5.1	Performance Evaluation	43
5.1.1	Uptane with native client	44
5.1.2	Uptane with Verification Service	44
5.1.3	Uptane with Verification Service inside Gramine	44
5.1.4	Uptane with Verification Service inside Gramine and SGX	45
5.1.5	Remote Attestation	45
5.1.6	Summary	46
5.2	Security Evaluation	47
5.2.1	Security flaws in traditional Uptane	48
5.2.2	Additional protection in our proposed Uptane	48
5.2.3	Summary	49
6	Conclusion	51
6.1	Future Work	52
	Bibliography	53

List of Figures

2.1	In-vehicle networks	6
2.2	Uptane workflow [17, 18].	9
2.3	Concept of a TrustZone supported execution environment	11
3.1	An illustration of full and partial verification methods. [67]	21
3.2	Message Handler Table [74]	23
3.3	Types of security attacks [16].	26
3.4	Security attacks 3.3 that affect the primary, organized by attacker capabilities. DR, TS, RS, SP, TR, and RT denote the director, timestamp, release, supplier, targets, and root keys respectively. Red keys are easier to compromise than blue keys. The * symbol denotes that an attack is limited to ECUs signed by the given roles. The # symbol denotes that an attack is limited till the earliest expiration timestamp. The @ symbol denotes that an attack can be detected, if not prevented [16]	27
3.5	Problem under discussion, Secondaries with PV (Partial Verification) does not provide FV (Full Verification) guarantees, especially if the Primary is corrupted. We assume that the Server is trusted.	28
3.6	Security attacks that affect Secondaries if attackers have compromised the Primary. [16]	28
3.7	Modified Full verification process and the communication between client and the VS. The client SD (Serialize Data) to the VS and the service responds with an OK if the verification succeeded. Otherwise, the update cycle stops	30
3.8	Communication between Client-side and Enclave in a Primary ECU performing Full Verification, according to our proposed approach.	31
3.9	The remote attestation flow of Intel SGX in our proposed uptane.	32
4.1	Modified update model proposed by Mukherjee et al. [63] which is divided into a secure Uptane server and a modified Uptane client-side implementation where it can run trusted applications within the trusted OS inside TEE.	34
4.2	Regular integration of Intel SGX [75]	35
4.3	Integration of Intel SGX with Gramine [75]	35
4.4	Example of running application with Gramine [80]	36
4.5	Security policies in manifest file of Gramine [80]	36
4.6	Gramine EPID remote attestation. [81]	38
4.7	Gramine DCAP based remote attestation. [81]	39

Glossary

OTA	Over the air
ECU	Engine Control Unit
TEE	Trusted Execution Environment
SGX	Software Guard Extensions
IoT	Internet of Things
CAN	Controller Area Network
LIN	Local Interconnect Network
MOST	Media Oriented System Transport
SOTA	Software Over the air
TUF	The Update Framework
REE	Rich Execution Environment
SoC	System on Chip
DRAM	Dynamic Random Access Memory
EPID	Enhanced Privacy ID
IAS	Intel Attestation Service
DCAP	Data Center Attestation Primitives
PCS	Provisioning Certification Service
EPC	enclave Page Cache
PRM	Processor Reserved Memory
RoT	Root of Trust
SME	Secure Memory Encryption

SEV	Secure Encrypted Virtualization
SSD	Solid State Drive
UART	Secure Memory Encryption
VIN	Vehicle Identification Number
URL	Uniform Resource Locator
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
PAL	Platform Adaption Layer
TOML	Tom's Obvious Minimal Language
gRPC	Google Remote Procedure Call

Chapter 1

Introduction

Vehicles are quickly transforming from a simple means of transportation into complex computers on wheels, that are increasingly smart and deeply integrated into our digital lifestyles. Growing increasingly connected and computer-like, vehicles now have capabilities to sync with mobile phones, provide navigation updates, and communicate safety information to other vehicles and surrounding infrastructure.

One great advantage is the user experience often comes down to how quickly remote updates can be made, by downloading an updated code from a cloud server provided by the Original Equipment Manufacturer (OEM) instead of physically visiting a technician. Also, with Over The Air (OTA) updates, it is possible to enhance the consumer experience by allowing drivers to add new features [1]. In terms of security, OTA updates can also maintain a secure drive by handle the patching of security gaps in a quick and efficient way.

The update process on the Electronic Control Unit (ECU) is composed of an intelligent and innovative architecture, giving specific roles for each type of ECU. The Primary ECU manage the outside vehicle communication by downloading and verifying the package and it is responsible of supplying this update to its Secondaries ECUs which should not communicate with outside for security and efficiency purposes [2-4].

1.1 Motivation

The concept software OTA updates brings along not only automation and efficiency benefits, but also increased exposure to Cybersecurity threats.

Hackers tend to use attacks via updates by the fact that this mechanism gives them power over the system once it is compromised. Not only this problem is worrying for the automotive industry but also for the other technological industries in which attack scenarios have already occurred multiple times [5–9].

An example happened a few years ago in South Korea [10], when attackers exploited a software update mechanism to spread malware. Government officials estimated that the attack had cost three-quarters of a billion US dollars in economic damage. In the case of vehicles, the impact is far more serious because human lives are at stake. This is commonly done by compromising the repository that serves software users and inserting malicious updates [11].

In automotive industries, various scenarios of OTA updates attacks occurred as well [12–15]. Therefore, it is important to operate repositories that serve software to vehicles while assuming that attackers can control communications anywhere between the repository, where the software updates are contained, and the ECUs [16].

Uptane [16], has become the first software update framework for automobiles that addresses a comprehensive and broad threat model and gives effective solutions to provide protection against multiple security threats using repository roles in order to distribute responsibilities and enhance security guarantees during the update [16–18]. For example, previous update systems were not capable of adding and validating new types of signed metadata to improve resilience to attacks.

However as all security solutions, Uptane does not give full safe guarantees for the ECUs [16, 17]. The Primary ECU, that has a sophisticated verification process (Full verification) which verifies and caches the timestamp, release, root, and targets metadata, is vulnerable to some security attacks, depending on the attacker capabilities, such as having compromised a set of keys used to sign updates [16]. When compromised, the impact can be huge, and the Secondaries ECUs must perform their own verification process to avoid to worst-case scenarios. Still, some Secondary ECUs have less hardware resources than others and those particular ECUs perform a basic verification process (Partial verification) which verifies and caches only the director metadata. The Partial verification method is susceptible to more attacks (like Arbitrary software and Rollback attacks) than to those that perform Full verification [16, 17, 19].

Therefore, the security of the Secondary ECUs that perform Partial verification should be taken into consideration. Within this context, our main goal in this dissertation is to

give this Secondaries ECUs the security guarantees provided by the Primary ECU (Full Verification), resulting in an improvement on their security process. We will use Intel SGX to isolate and execute the Full verification method and the Secondaries ECUs will rely on remote attestation provided by the Primary ECU.

1.2 Thesis Goals

To achieve our goal, we redesign the way that Primary ECUs which perform Full verification, providing their verification guarantees to the Secondary ECUs, which some only perform Partial verification. In particular, we will go through the following steps:

- Refactor Uptane implementation particularly the Primary ECU for better understanding of critical components (which are methods of metadata verification) and non-critical components in the client-side.
- Design an API by creating a Verification Service which is going to contain the critical components of full verification process and communicate between the non-trusted components of the client-side.
- Implement critical components of Full verification within the TEE.
- Implement a remote attestation interface on the Uptane process, the Primary ECU generates and sends the Quote to the Secondaries ECUs which corresponds to attesting that the Full verification process was performed correctly inside the enclave.

1.3 Document Structure

In this thesis, we will clarify the development of this research plan.

Chapter 2 will provide some required background on general OTA [20], Uptane [16, 17, 19] and TEEs [21, 22]. In the end of this chapter, we will describe some related work on the use of OTA, Uptane and/or TEEs.

Chapter 3 will clarify with more detail our thesis goals, step by step. We describe our TEE-enabled secure update framework, break down the relationship between the Uptane standard and a client implementation, and define concretely which parts of the client implementation shall be executed inside a TEE according to our framework.

Chapter 4 describe the design and the implementation of our modified Uptane justifying why we use certain types of technologies.

Chapter 5 show the results of our proposed solution, the evaluation of the performance of our modified Uptane comparing to the native Uptane. Finally we evaluate the security of our solution explaining what additional benefits it brought.

Finally, Chapter 6 presents the conclusions and discusses possible improvements left for future work.

Chapter 2

Background and Related Work

This chapter describes the relevant background for this dissertation. This includes an overview of the Uptane Framework. Also, this chapter will explain the concepts related to Trusted Execution Environments, followed by a description of some types of TEEs that are commonly used by manufacturers of constrained chipsets and Internet of Things (IoT) devices. To finish this section, we cover a brief description of some of the most relevant related work.

2.1 Automotive context

Nowadays, modern vehicles are made up of mechanical components (e.g., brakes, engine, battery, etc...) that are controlled via software or firmware on small devices called ECUs. These units are responsible for controlling a specific function of the car. For example, if an accident happens, the airbag ECU would choose which airbags to deploy, depending on the location of the passengers, or an automatic emergency braking ECU would receive inputs from the sensors that detect an approaching obstacle [23].

Inside the vehicle, the data flow of the ECUs is through multiple types of communication buses [24] which include the Controller Area Network (CAN), Local Interconnect Network (LIN), Media Oriented System Transport (MOST), Ethernet and FlexRay. Among these in-vehicle communication buses, the most popular is CAN [25] since this bus is the main communication network used in automotive applications [26].

These buses have their own local gateway and to be able to communicate with each other, there is a node called the central gateway that acts as a communication controller. Figure 2.1 shows an in-vehicle network architecture topology where multiple ECUs are

installed in each data bus and the buses are connected through the central gateway which converts data from one bus format to another bus format.

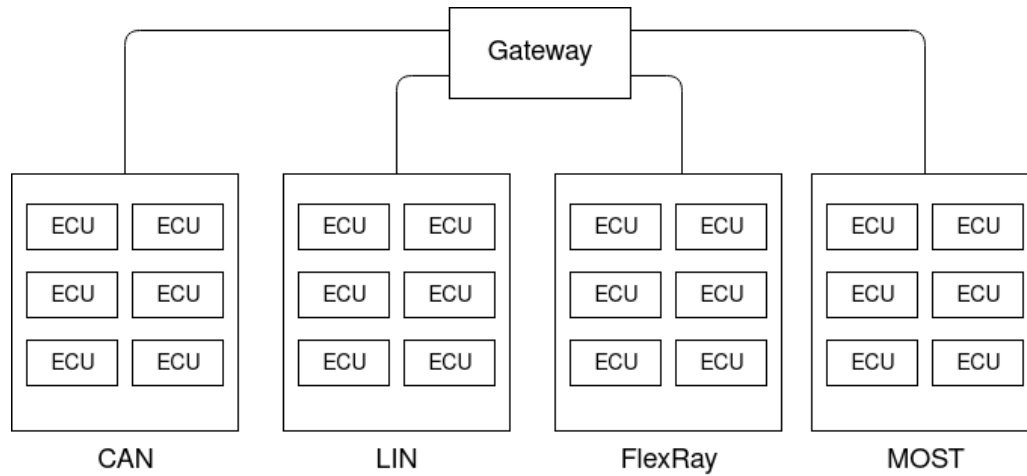


FIGURE 2.1: In-vehicle networks

2.1.1 OTA process

The OTA update mechanism refers to the method of delivering the update wirelessly, i.e., “over the air,” and sent directly to the device. OTA updates have been in the software industry a long time ago for the deployment of functional enhancements and correct deficiencies to both the operating systems and application programs [27]. With regard to the automotive industry, OTA updates for cars are expected to grow rapidly as a result of the expansion of integrated vehicles. The OEMs (Original Equipment Manufacturers) are increasingly adopting this technology for delivering updates to embedded devices that are part of the vehicle [2, 20, 28].

The following elements should be considered in the ECU software update process:

- Ensure the integrity of software update.
- Reduce software update time.
- Secure software update.

Two main types of OTA updates are used in practice, Software Over-The-Air (SOTA) and Firmware Over-The-Air (FOTA). SOTA updates generally refer to downloading software components meant for any system in a device, not restricted to firmware for ECUs. On the contrary, FOTA updates are more specific, as the term refers to downloading an update that involves replacing a specific firmware image in a embedded device. In other

words, the process of updating firmware concerns the updating of the main system software that controls an specific hardware [20, 29, 30].

The following update scenario proposed by Kim and Park [2] and Onuma et al. [3] gives us an overview of how OTA updates on an ECU are processed. Before performing an ECU software update, the vehicle must be parked for safety purposes. The OEM server generates new update information, which it transmits to the OTA Master of the vehicle. The role of the OTA master is to check which target ECU is mentioned in the update process and to forward the update towards to it. The OTA Master will store the information received from the server and wait for approval from the ECU to update via a Human Machine Interface (HMI). If the driver agrees, the package from the OTA Master will be sent and the ECU will proceed with the software update process. After the driver selects the ECU to update, the vehicle will restart with the new updated ECU installed.

2.1.2 TUF

Before we talk about Uptane, we have to explain how and why it emerged. It all started with a project that was designed from the scratch called TUF (The Update Framework). TUF is an open-source secure OTA framework designed to protect users of software repositories by focusing on protection from key compromise attacks [31, 32].

The security-aware system has four key principals:

- **Separate the duties:** TUF uses multiple keys and roles to expand the verification process by distributing responsibilities and increase compromise resilience. Four main roles are used in this framework: Root, Timestamp, Target and Snapshot [31].
- **Optional use of a threshold number of signatures:** Thresholds were added to each role to increase security, meaning that you must sign a minimum number of t out of n keys to compromise the a role.
- **Process to revoke keys:** To verify a metadata file, public keys can be revoked by including an expiration date in the metadata file which helps to protect clients from trusting outdated keys.
- **Use offline keys, or private keys to sign the most sensitive roles:** Even if the attackers controls the repository, they can not be able to sign malicious versions of sensitive metadata.

At the same time, industrial partnerships were seeking OTA update solutions for the automotive, and explored in particular the use of TUF as a secure update framework. One thing that they realized is that TUF only offers one secure view to a repository and cannot provide a customized view based on the client's needs. Other discoveries pointed that this framework has vulnerabilities in some critical security attacks. For example, for deny updates attacks, Freeze attacks are easily performed against Primary and Secondaries ECUs. Also, the attacker can perform deny functionality attacks like endless data attacks and mixed bundle attacks on the Secondaries ECUs with no need to compromise any keys roles [16].

2.1.3 Uptane

Uptane has then emerged as an adaptation of TUF for the automotive, targeting the secure updating of ECUs. Since its inception, Uptane [16] has become a popular open source secure software update framework that has achieved very positive adoption in the automotive industry in order to prevent large-scale updates attacks – the team of researchers behind its development worked in close collaboration with representatives of OEMs that manufacture 78% of all cars on US roads [17]. Not only is Uptane present in the automobile industry, but it can also be employed in other vehicles such as trains [33] or drones [34] or other constrained IoT devices. The original Uptane implementation was written in Python but also another project named Aktualizr [35, 36] was designed and implemented in C++ the OTA client-side functionality according to the Uptane framework mechanisms.

2.1.3.1 Uptane process

We now briefly describe how Uptane works. The standard implementation of the Uptane framework is composed by a secure communication channel between the secure server representing the OEM and the client representing the Primary ECU.

The secure server implementation has three core components (i.e. the Image Repository, the Director Repository, and the Time Server) to provide security through the validation of images before downloading. The Director repository has the function of instructing ECUs about which images should be installed, by producing a mandatory signed metadata. The Image repository stores every image currently deployed, along with the metadata files that have been securely signed by OEM offline keys, proving their authenticity. Finally, the Timeserver repository generates timestamp tokens that enable checking

if the most recent package that was updated is being delivered to the client. Figure 2.2 illustrates an overview of the elements involved in the Uptane update process and how the system behaves. The right box contains a Primary ECU which is the gateway ECU in the vehicle, and communicates with the external cloud and the internal Secondary ECUs. On the left side is the external cloud where the repositories are located. If the ECU requires an update, the Primary ECU generates and sends its vehicle version manifest with vehicle metadata to the Director repository. The function of the Director is to choose which images should be installed next. A set of new images and metadata for these selected images is then sent to the Primary, from both the Director Repository and Image Repository. After receiving the package, the Primary ECU will run a verification process to check if the images and metadata are genuine. If no anomalies are found during the verification process, the Primary ECU will transmit the update package to the Secondary ECUs. Also, the Secondary ECUs will perform a verification process which can be Full or Partial Verification depending on the resources and importance of the ECU. The Secondary ECU will install the new images if the verification succeeds and the vehicle version manifest will be updated.

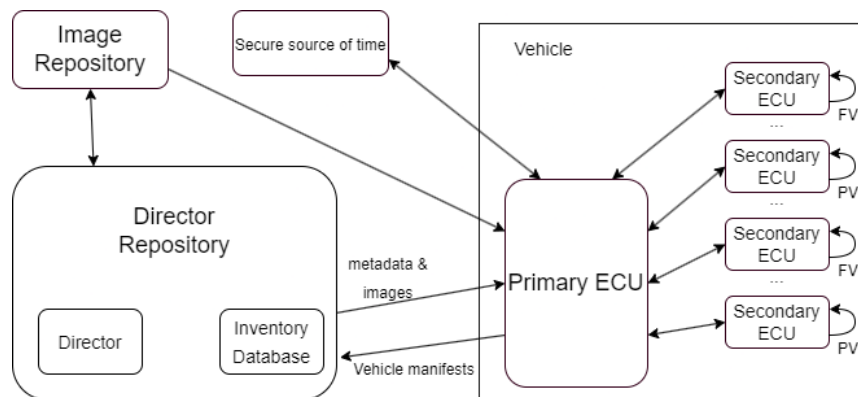


FIGURE 2.2: Uptane workflow [17, 18].

2.2 Trusted Execution Environment

Trusted Execution Environments (TEEs) are special modules of modern CPUs that allow the isolated execution of security-critical applications. They offer added security, in the sense that sensitive data from such applications is not known to the rest of the system.

In summary, according to the Confidential Computing Consortium [37], a TEE provides a level of assurance characterized by the following three properties [38]:

- **Data confidentiality:** Unauthorized entities cannot view data while in use within the TEE.
- **Data integrity:** Unauthorized entities cannot add, remove, or alter data while it is in use within the TEE.
- **Code integrity:** Unauthorized entities cannot add, remove, or alter code executing inside the TEE.

A TEE establishes a secure execution environment by creating a safe, isolated space using hardware and software primitives. Most TEE threat models consider a powerful adversary controlling user-space applications, the operating system, or even the hypervisor, trying to influence the execution of applications in the trusted environment. To achieve the protection needed for this threat model, TEEs require strong isolation between the TEE and the Rich Execution Environment (REE).

The innovative and appealing idea of shifting security critical applications into a trusted execution environment is already adapted by major vendors like Intel, ARM, and Apple. Common TEE designs use hardware extensions like Intel SGX [22] and ARM TrustZone [39] to create a virtual secure processor in the main application processor. Other technologies like Google's Titan [40] and Apple's T2 [41] mount a dedicated security processor next to the main CPU [42].

2.2.1 ARM Trustzone

The ARM System on Chip (SoC) processors created a way to keep data and code secure at its highest level. This is done via separation of computation into two separate worlds, the secure world and the normal world [21]. The separation is accomplished by a Secure Configuration Register bit called monitor mode [43, 44], the non-secure bit is set to 1 and a bit set to 0 means that is secure. This isolation between those two worlds actually affects parts of the infrastructure like the main memory, system bus, peripheral devices and the interrupt configuration.

Diagram 2.3 illustrates the division between the two separate sections. The REE Normal World, which contains generic applications and possibly application with security, communicates with the trusted applications in the TEE via the TEE Client API [44] and is managed by the secure monitor. The secure monitor then communicates with the secure operating system/kernel and with different secure standalone applications or services.

Client Applications running in REE need to connect to the TEE Secure World before connecting to Trusted Application. To open a session, the Universally Unique Resource Identifier (UUID) of the target TA needs to be specified. At this time, the library will then look for a trusted application with that UUID, and pass the whole trusted application image to the TEE without the client application’s knowledge of how the actual communication happens.

The ARM TrustZone, unlike Intel SGX, does not provide a native mechanism for remote attestation, preventing relying parties from proving that the execution took place inside the TEE remotely [45, 46]. Many protocols provides implementation of additional end-to-end security solutions, likewise, remote attestation [47–49]. However, these protocols require the availability of extra hardware with additional security features [50].

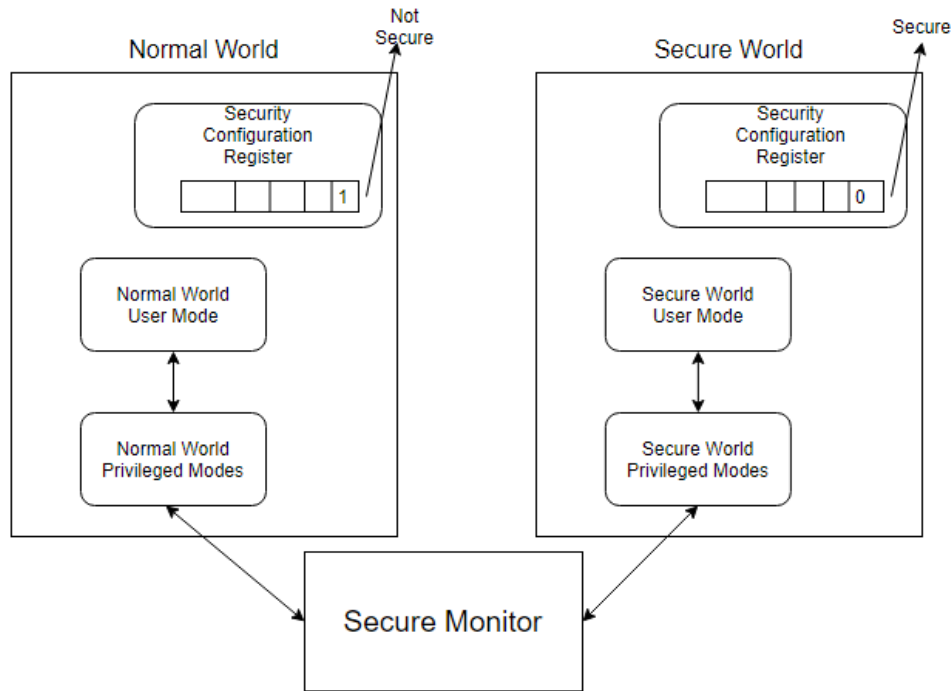


FIGURE 2.3: Concept of a TrustZone supported execution environment

2.2.2 Intel SGX

Unlike ARM Trust Zone, where two separate sections of execution – or two worlds – are defined, Intel SGX provides a mechanism of creating secure enclaves. These secure enclaves are essentially containerized sections of memory, a place where data and code can be loaded or executed securely and while being verified by cryptography attestation keys. This trusted memory space, called the Enclave Page Cache (EPC), is divided into

4Kb pages that are encrypted in Dynamic Random Access Memory (DRAM) using Memory Encryption Engine which sits at the edge of CPU, connected to Memory Controller [51]. The decryption of the pages only occurs inside the physical processor core.

The application code is attested by a remote entity and calculated during enclave creation. The verification procedures are based in two measurement registers: the MRENCLAVE that provides an identity of the enclave code and data; and the MRSIGNER that provides an identity of an authority over the enclave. This feature gives the relying party increased certainty that the software is running inside an enclave on a fully updated system at the latest security level [52, 53]. If an enclave proves its identity to a remote party, an encrypted communication channel can be established between the two. The remote host creates a cryptographic report (Quote) that is signed by a hardware-specific key. A Quote contains information about the enclave that signed it.

Intel SGX currently supports two types of remote attestation [52, 54, 55]:

- **Enhanced Privacy ID (EPID)** enables signing objects anonymously by dividing signers to groups (i.e. EPID groups). The EPID groups create signatures with their own secret keys, and this signatures can only be verified by a public key of the group they belong to. With this method, it is possible to check if the signer belongs to the right group; however, the EPID scheme does not uniquely identify the signer [54].
- **Data Center Attestation Primitives (DCAP)** provide an architecture to benefit from remote attestation without contacting Intel services to validate the attestations at runtime. Thus, a data center can create its attestation infrastructure using the Elliptic Curve Digital Signature Algorithm (ECDSA) algorithm [55].

Looking at the enclave setup, a part of memory is reserved as Processor Reserved Memory (PRM) and the CPU has to protect this memory structure from external memory accesses. The PRM controls the enclave page cache, storing critical enclave information such as code and data [56].

2.2.3 Other solutions

In 2019, Google launched OpenTitan, an open-source secure chip design project [57]. The objective of the OpenTitan project is to make the execution of Root of Trust (RoT) which is defined as “system element that provides services, including verification of system, software/data integrity and confidentiality, and data (software and information) integrity

attestation between other trusted devices in a system or network” [58]. The RoT shall provide the methods for maintaining and verifying the security and integrity of the embedded boot code and firmware, as well as cryptographic material.

In 2016, AMD introduced a new extension called Secure Memory Encryption (SME) [59]. This extension defines a simple and efficient architectural capability for main memory encryption with a purpose to protect security-relevant assets that reside there. It uses an encryption key that is randomly generated by the AMD secure processor and is loaded into the memory controller at boot time to encrypt the memory. In order to make the memory encryption more transparent and be able to run with any operating system, Secure Encrypted Virtualization (SEV) [60, 61] extends SME to AMD-V, allowing individual Virtual machines to run SME using their own secure keys.

In 2018, the Apple T2 chip was introduced [41]. The T2 processor is used to implement the “Secure Boot” feature. Compared to its predecessor (Apple T1 chip) the T2 chip has better performance handling more tasks such as early boot tasks. It is securing data storage at-rest by encrypting it on the Solid State Drive (SSD) using dedicated hardware that has a 256-bit key. This processor also has a built-in “Secure Enclave” that can store and/or process critical information like user’s fingerprint data [62].

2.3 Related work

Due to several possible types of threats and the importance of OTA updates, research efforts have been made to achieve various levels of protection against these attacks. In this section, we will visit some previous work related to OTA updates, with or without TEEs, and other alternatives besides Uptane. Then, to end this section, we will provide a brief overview of a project that combined Uptane with TEE [63], which will also serve as a comparison with our proposed solution.

Dhobi et al. [64] propose to use Secure Firmware Update OTA of embedded devices, leveraging ARM TrustZone technology to check integrity, authenticity and security of firmware updates. During the update verification process, the RSA algorithm is used to verify the image and signature to authenticate the firmware and ensure security. Their empirical analysis showed some improvements on the integrity of the system. Some tests were made based on security parameters (i.e. Load Trusted Application with fake UUID, Loading a fake Trusted Application from non-secure world); other measured time consumed by the process based on size of the firmware and the key.

The article written by Al Blooshi and Han [34] applies Uptane framework in Drone Environments. They consider a “drone swarm” which consists in multiple drones, like vehicles which possess multiple ECUs. Based on the Uptane process in vehicles, the “drone swarm” has a CH (Cluster Head) which has the same function as the Primary ECU who communicates with the repositories, downloads and verifies the package. If no issue is found during the verification process, the package will be transmitted to each following Drone, who is a member of drone swarm controlled by CH, which in an automotive perspective, they are the Secondary ECUs [34]. The GS (Ground Station), which is a drone operator who has information of public keys of drones. The GS has the function of establishing a drone swarm, uploading the updates to the update server and defining the URL of the update server. They conclude that the proposed design is suitable for the drone environments while supporting the fundamentals of Uptane.

Nilsson et al. [65] present a framework for self-verification of firmware updates over the air. To deploy the firmware, they have chosen Microkernel Architecture as the reasonable candidate platform for memory isolation. The device architecture was split into two parts: a portal that contains the new firmware and is considered to be trusted by the vehicle, and a vehicle that has a wireless gateway and an in-vehicle network consisting of interconnected ECUs. They developed a prototype system based on the virtual machine monitor SPUMONE [66]. SPUMONE provides memory isolation and communication between the control and functional system, where the control system is based on the microkernel and it handles the flashing and verification procedure of the functional system. In their analysis, they ensure that the verification code in the control system cannot be modified by an attacker because of the effectiveness of memory isolation.

Qureshi et al. [67] mention that the traditional Uptane Framework has vulnerabilities against control attacks, so they proposed a framework called eUF (enhanced Uptane framework) where they add a new layer in the traditional Uptane Framework based on DL (deep learning) techniques in order to detect control attacks launched during OTA software update. As ECUs under Partial Verification are more susceptible to attacks, they aimed to detect control attack in those ECUs. The result from training those DL models was stored in secondary ECUs. They also report an improvement on compromise resilience and detection of malicious software updates at Secondary ECUs when the partial verification method is employed.

Asokan et al. [68] proposes a secure firmware update framework called ASSURED for

large-scale IoT. ASSURED provides end-to-end security based on the TUF framework in order to improve the functionality of update constraints IoT devices in terms of deployability and performance. For more security guarantees, the authors leverage two low-end security architectures: ARM TrustZone-M [69] which isolates critical resources (e.g memory and interrupt lines) from the non-trusted code, and HYDRA [70] which checks if the code in the secure world has been correctly performed by hybrid remote attestation. The results conclude that this solution enhances the current update mechanisms in terms of performance and deployability in realistic settings.

Mukherjee et al. [63] proposed to isolate and deploy Uptane entirely inside a TEE using ARM TrustZone in order to protect the integrity of (BEV/EVSE) and avert any external or internal system vulnerabilities. The Uptane framework was decoupled into a secure server and a modified client-side implementation (which consists only communication channel and a metadata staging area, and services client-specific functionalities) to run as trusted application(s) within the secure world inside TEE. They used the Universal Asynchronous Receiver-Transmitter (UART) [71] for communication and to transfer data and between the server and the client-side. The OTA update requests were triggered periodically and the subsequent responses with image and metadata download from the server through a secure UART communication. They also included a software analysis and used a verification tool (SAWScript [72]) which detects any logical and programming correctness errors of the client application which runs inside the TEE in order to verify its reliability against any security vulnerabilities.

Chapter 3

Extending the Uptane trust-chain using TEE

In this chapter, at first we will describe Uptane implementation and its relation with the standard Uptane. Then, we present the overview of the proposed approach, a detailed description of each phase and relevant activities. Finally, we explain the main challenges of the proposed solution.

3.1 Standard Uptane client

This section provides a detailed description of how a concrete Uptane client should be implemented. We are going to describe the latest standard version released (V2.0.0) [19].

3.1.1 Roles on repositories

- **Root role:** The Root role is responsible for certificating the authority of the repository by distributing and revoking public keys used to verify the root, timestamp, release, and targets role metadata.
- **Targets role:** The Target role has the function of signing metadata such as the cryptographic hashes and file sizes of images that verify the image.
- **Snapshot role:** The Snapshot Role is responsible for signing metadata about all Targets metadata that the repository releases, including the current version number of the top-level Targets metadata, and the version numbers of all delegated Targets metadata.

- **Timestamp role:** The Timestamp role indicates whether there is any new image or metadata available on the repository.

3.1.2 Types of metadata

To ensure security, the Uptane Framework relies on a properly created metadata having a designated structure. The four Uptane roles described before share a common structure. They SHALL sign a payload of metadata and contain an attribute which stores the signature(s) of the payload. The following Table 3.1 provides a description of the four different types of metadata and some common characteristics shared between them, each corresponding to a metadata role.

3.1.3 Server implementation

The Uptane implementation SHALL make the following repositories available in order to communicate with the Primary ECU:

- **Director repository:** The Director repository has the function of instructing ECUs as to which images should be installed by producing a required signed metadata. This repository also possesses a private inventory database containing critical information about the vehicle (contains a unique identifier called VIN (Vehicle Identification Number)) and its ECUs (contains ECU type, key and unique identifier). Also, this inventory should record a hardware identifier for each ECU to prevent installation of incompatible firmware. The interface should be public so that the Primary ECU uploads its vehicle version manifest. The implemented storage permits an automated service to change generated metadata files.
- **Image repository:** The Image repository stores every image currently deployed, along with the metadata files that are securely signed by a private key, proving their authenticity. The interface should be public, permitting the Primary ECU to download the image and metadata. It shall provide a method that allows authorized users to upload images and their associated metadata. It also verifies if the images are trustworthy by checking the chain of delegations.
- **Timeserver repository:** Finally, the timeserver repository generates timestamp tokens that check and validate if the most recent package that was updated is being delivered to the client.

Metadata type	Description
Root Metadata	<ul style="list-style-type: none"> • The repository's Root metadata has the responsibility to distribute the public keys of top-level Root, Targets, Snapshot, and Timestamp roles. • It SHALL contain two attributes: <ol style="list-style-type: none"> 1. A representation of the public keys for all four roles. The public keys SHALL have a unique identifier. 2. An attribute mapping each role to its public key, and the threshold of signatures required for that role.
Targets Metadata	<ul style="list-style-type: none"> • The repository's Target metadata contains all the information about the images to be installed on ECUs (i.e., filename, file sizes, hashes). • Can also contain metadata about delegations, enabling one Target role to delegate its authority to another.
Snapshot Metadata	<ul style="list-style-type: none"> • Lists version numbers and filenames of all Targets metadata files, ensuring protection against mix-and-match attacks. • Lists filename and version number of the Root metadata in order to provide backward compatibility.
Timestamp Metadata	<ul style="list-style-type: none"> • Timestamp Metadata SHALL contain the filename and version number of the latest Snapshot metadata and one or more hashes of the Snapshot metadata file besides the hashing function used.

TABLE 3.1: Description of each type of metadata used by the Uptane Framework.

3.1.4 Client-side

In this subsection we are going to describe the procedures made by a client side implementation, with a particular emphasis on distinguishing the Primary and Secondary ECU operations. In figure 3.1 represents the respective procedures of each ECU.

3.1.4.1 What the Primary ECU does

The role of Primary ECU is to download, verify, and transmit the images and metadata to the Secondary ECUs.

The Primary SHALL execute the following steps:

1. **Construct and send vehicle version manifest:** Before starting any download process, the Primary ECU needs to build a vehicle version manifest and send it to the Director Repository. The Director checks the manifest and determines an appropriate update package for the vehicle.
2. **Download and check current time:** For secure attestation of a sufficiently recent time.
3. **Download and verify metadata:** After downloading the metadata, the Primary ECU MUST perform Full Verification. The Full verification method will be specified in Section 3.1.4.2.
4. **Download and verify images:** The Primary ECU will verify and validate that the latest image matches the latest metadata.
5. **Send the update package to the Secondaries ECUs:** If the Primary verification method is performed without failure, ensuring that the downloaded package was legitimate, the metadata and image are sent to the associated Secondaries and SHALL include the necessary content required for another verification process. As explained before, the Secondary ECUs have two types of verification depending on their resources and how security critical they may be (i.e. Full verification (Section 3.1.4.2) and Partial verification).
6. **Install image** If all pre-conditions are correctly checked, an ECU SHALL attempt the installation process. If the installation process fails, we need to ensure that the ECU has a backup of its current image and metadata.

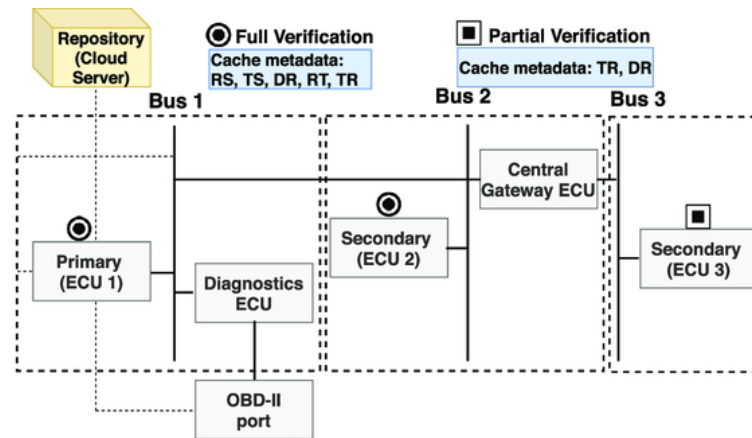


FIGURE 3.1: An illustration of full and partial verification methods. [67]

3.1.4.2 Full Verification

The Full verification process MUST have the full set of metadata from Director and Image repository, which is comprised of Root, Targets, Snapshot, and Timestamp. The files are obtain from the primary, which are already verified. To defend against security attacks, the ECU checks all of these metadata following the TUF protocols. During the process, Full Verification must verify if hashes and version numbers of the downloaded firmware image match those of the file in both the Director and Targets metadata.

To perform full verification, the ECU SHALL perform the following steps:

1. Load and verify the current time or the most recent securely attested time.
2. Download and check the Root metadata file from the Director repository.
3. Download and check the Timestamp metadata file from the Director repository.
4. Download the Snapshot metadata file from the Directory, and check if the hashes and version numbers of that file don't match the hashes and version number listed in the new Timestamp metadata. Otherwise, there are no new updates and the verification process SHALL be stopped and considered complete.
5. Download and check the Targets metadata file from the Director repository.
6. Download the Root metadata file from the Image repository, and check if the Targets metadata from the Directory repository reveals that there are new targets that are not currently installed. Otherwise, there are no new updates and the verification process SHALL be stopped and considered complete.

7. Download and check the Timestamp metadata file from the Image repository.
8. Download the Snapshot metadata file from the Image repository, and check if the hashes and version number of that file do not match the hashes and version number listed in the new Timestamp metadata. Otherwise, skip to the last step.
9. Download and check the top-level Targets metadata file from the Image repository.
10. This step depends on which ECU is running the verification process:
 - **Primary ECU** SHALL verify that the Targets metadata from the Director and Image repositories matches for all images listed in the Targets metadata file from the Director repository downloaded in step 6.
 - **Secondary ECU** SHALL verify that the Targets metadata from the Director and Image repositories matches only on the metadata for the image it will install.

If any steps described fails, the ECU MUST return an error message indicating the failure. In case during the verification process, if a security attack is detected, the ECU SHOULD return a error message that indicate the type of attack.

3.1.4.3 Partial Verification

The partial verification is performed by following this steps:

1. Load and verify the current time or the most recent securely attested time.
2. Download and check the Targets metadata file from the Director repository.

3.1.5 Uptane implementation

The Uptane reference implementation is written in Python based on official Uptane repository [73]. The optional alternative Aktualizr written in C++ was not selected due to be an outdated and complex implementation. In this implementation we assume that all essential (MUST) requirements are implemented, and some optional (SHOULD/SHALL) requirements are implemented.

3.1.5.1 Protocols

When the Primary ECU communicates with the Director repository, Image repository, and time server, it uses HTTP POST protocol and requests over XML-RPC. During the

communication, the Primary acts as a XML-RPC client and the server side (the Director repository, Image repository, and time server) acts as XML-RPC server. When it comes the communication between the Primary and the Secondaries ECUs, the Primary acts as XML-RPC server and the Secondaries acts as XML-RPC client. The requests supported by XML-RPC servers mentioned are described in the figure 3.2. The XML-RPC traffic is transmitted over TCP/IP and does not support CAN or other types of network.

3.1.5.2 Message Handler

The messages are sent using XML-RPC with the function names in the Request column and data from the Data column. In addition to the table 3.2, both the Director and Image repository hosts the Root Metadata, Targets Metadata, Snapshot Metadata, and Timestamp Metadata, and the Director repository hosts these metadata using XML-RPC.

Request	Sender	Receiver	Data	Response	Specification Reference
submit_vehicle_manifest	Primary ECU	Director Repository	VehicleVersionManifest		https://uptane.github.io/uptane-standard/uptane-standard.html#director_repository
register_ecu_serial	Primary ECU	Director Repository	ecu_serial Identifier, ecu_public_key PublicKey, vin Identifier, is_primary BOOLEAN		
get_signed_time	Primary ECU	Timeserver	SequenceOfTokens	CurrentTime	
submit_ecu_manifest	Secondary ECU	Primary ECU	vin Identifier, ecu_serial Identifier, nonce, ECUVersionManifestSigned		https://uptane.github.io/uptane-standard/uptane-standard.html#create_version_report
register_new_secondary	Secondary ECU	Primary ECU	ecu_serial Identifier		
get_time_attestation_for_ecu	Secondary ECU	Primary ECU	ecu_serial Identifier		https://uptane.github.io/uptane-standard/uptane-standard.html#send_time_primary
get_image	Secondary ECU	Primary ECU	ecu_serial Identifier		https://uptane.github.io/uptane-standard/uptane-standard.html#send_images_primary
get_metadata	Secondary ECU	Primary ECU	ecu_serial Identifier, is_partial_verification BOOLEAN		https://uptane.github.io/uptane-standard/uptane-standard.html#send_metadata_primary

FIGURE 3.2: Message Handler Table [74]

3.1.5.3 Operations

The following operations are part of the implementation:

- The Primary ECU as described in standard is the only ECU who communicates with the Director repository, Image repository, and time server;

- The Primary ECU sends the time server nonce tokens from each ECU that produces a time attestation for each update cycle;
- The implementation supports asymmetric keys and uses the same keys for encrypting and signing
- The Director repository and Image repository have public interfaces for communication with ECUs and stores the metadata in a file system that is hosted to form the public interface.
- The Primary ECU identifies to the Director repository using a vehicle version manifest. As described in the standard, the Director verifies the vehicle version manifest, however it does not make any additional checks.

3.1.5.4 Metadata

Some metadata features that are optional in the standard are supported in the existing implementation. The details of the metadata features are the follow:

- When the metadata updates, the version number will increment.
- Root metadata supports mappings of others roles using a map file (described in table 3.1)
- The ECUs contains encrypted images. As the Image repository does not know which ECU should deliver the specific image, it is necessary that both Image and Director repositories contain information about the unencrypted image in Targets metadata.
- On the Director repository, the Target metadata supports a functionality that encrypts images per ECU. This functionality also contains a release counter and an id from both the Director and Image repositories. This implementation the Director repository cannot add a download URL to the custom field of Targets metadata, it is not supported.
- The custom field of Target metadata on the Director repository is not compared to the one that is on the Image repository, meaning that this verify functionality described in the standard is not implemented.

- The implementation of the Snapshot metadata does not contain the Root filename or version.

3.1.5.5 Usage

Instead of using key management and image generation described in the standard, the implementation handles online keys for all roles for demonstration purpose.

3.1.5.6 Data

In the table 3.2 it is described the list of keys that are involved on each ECU, and the respective accesses to certain data according to the implementation.

Location	Data
Primary ECU	ECU private key * Timeserver public key * Currently installed version * Secondary's Vehicle Version Manifests * The most recent Root, Timestamp, Targets, and Snapshot metadata
Full verification Secondary ECU	ECU private key * Timeserver public key * Currently installed version * The most recent Root, Timestamp, Targets, and Snapshot metadata (for a new installation, just the known Root metadata) from both the Image and Director repositories
Partial verification Secondary ECU	ECU private key * Timeserver public key * Currently installed version * Director's Targets metadata public key
Director Repository	ECU public keys * Metadata about images * Inventory database * Online metadata private Director metadata keys * Metadata signed by offline Director metadata key
Image Repository	ECU public keys * Metadata about images * Images * Online metadata private image metadata keys * Metadata signed by offline image metadata keys
Timeserver	Timeserver private key * Current time

TABLE 3.2: Data table representing the required devices that are expected to have at least the following data [74].

3.2 Extending Uptane security model using TEE

The power and information contained in ECUs creates interest for hackers to attack and to steal information or even control a particular ECU. The attacker may want to read updates

to learn the contents of software updates in order to reverse-engineer ECU firmware, deny updates to prevent vehicles from updating and fixing current software problems or deny functionality to disturb the functioning of the internal system. Finally and perhaps the most dangerous, is when the attackers gains control of the ECU and modify the vehicle’s performance and correct behaviour.

The security mechanisms put in place by Uptane are processed in different ways depending on the type of ECU, its resources, and how security critical it may be [16]. The Primary ECU is typically the one that is more capable in terms of computation and storage capacity, and is the only one with connectivity to the outside of the vehicle. While the Secondaries ECUs some have more capacity than others and they must not establish communication outside the car, it depends on the Primary ECU for receiving and installing software updates. The Primary ECU was designed to download the metadata and firmware images from Director and Image repositories, verify the package by performing a full verification process 3.1.4.2. On the other hand, not all Secondaries ECUs have the same capabilities, some performs full or partial 3.1.4.3 verification of the image against the metadata.

Despite Uptane being a sophisticated security framework with the ability to protect against critical security attacks (listed in figure 3.3), it does not cover all possible security flaws.










Icon	Security attack
	Eavesdrop attack
	Drop-request attack
	Freeze attack
	Partial bundle installation attack
	Rollback attack
	Endless data attack
	Mixed-bundles attack
	Mix-and-match attack
	Arbitrary software attack

FIGURE 3.3: Types of security attacks [16].

Some attacks have different effects and some are more alarming than others. If the attacker wants the information of software updates, an eavesdrop attack may be performed. With this action, the attacker can read unencrypted updates sent from the repository to

the vehicle. Other attacks can deny updates, preventing the vehicle from fixing software issues. Drop-request attack, Freeze attack and Partial bundle installation attack can be some of these kind of attacks. Not only the attacker can deny updates, but also deny functionality. If the attacker would prefer to cause fail function of the vehicle, they can perform Rollback attack, Endless data attack, Mixed-bundles attack and Mix-and-match attack. Finally, and most severe of all, it is possible for the attacker install malware on the ECU making possible to arbitrarily modify the vehicle's performance.

Particularly relevant for this thesis, there is a possibility that a Primary can be compromised. In Figure 3.4, we can see a list of possible scenarios that affect the Primary ECU, depending on an attacker's capabilities.

Attacker capabilities	Attacks on the primary				
MitM					
MitM + DR					
MitM + TS RS DR					
MitM + TS RS DR SP					
MitM + TS RS DR TR					
MitM + RT					

FIGURE 3.4: Security attacks 3.3 that affect the primary, organized by attacker capabilities. DR, TS, RS, SP, TR, and RT denote the director, timestamp, release, supplier, targets, and root keys respectively. Red keys are easier to compromise than blue keys. The * symbol denotes that an attack is limited to ECUs signed by the given roles. The # symbol denotes that an attack is limited till the earliest expiration timestamp. The @ symbol denotes that an attack can be detected, if not prevented [16]

In the scenario shown in Figure 3.5, we assume that the Primary ECU has been corrupted. The Primary will broadcast the malicious package to Secondary ECUs, and the ones that perform Partial Verification are the most vulnerable and exposed to more types of attacks because, unlike the Full verification, it uses less roles (verifies and caches only the director metadata) that are necessary to distribute responsibilities and increase compromise resilience.

Figure 3.6 depicts a list of attacks that affect a Secondary ECU if the Primary is compromised. As we can see, there is a considerable difference in terms of vulnerability between Partial and Full Verification. For example, if the attacker controls only the Director Key, the ECUs with Partial Verification will be the target of more deny functionality security attacks (Rollback attack, Arbitrary software attack) compared to the Full Verification ones.

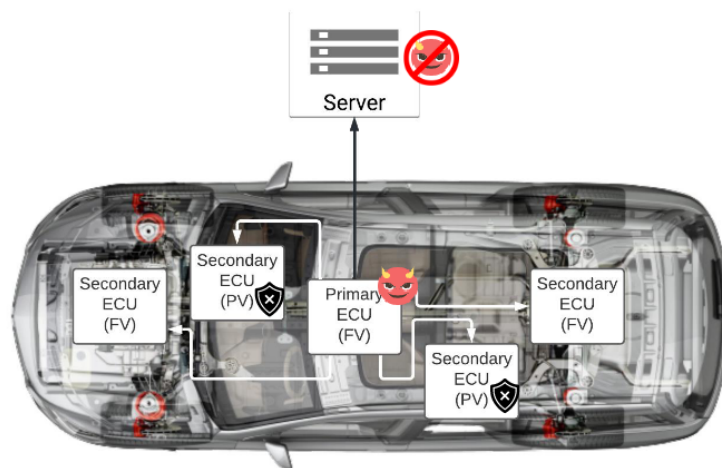


FIGURE 3.5: Problem under discussion, Secondaries with PV (Partial Verification) does not provide FV (Full Verification) guarantees, especially if the Primary is corrupted. We assume that the Server is trusted.

Attacker capabilities	Type of secondary	
	Full verification	Partial verification
MitM		
MitM + DR		
MitM + TS RS DR		
MitM + TS RS DR SP		
MitM + TS RS DR TR		
MitM + RT		

FIGURE 3.6: Security attacks that affect Secondaries if attackers have compromised the Primary. [16]

To avert that kind of vulnerabilities, we propose a customized Uptane framework that leverage a TEE in order to strengthen the security of Secondary ECUs that perform Partial Verification by giving them the same guarantees of Full Verification provided by Primary ECU. The concrete steps will be the following:

1. Customize the Client-side, focusing the Primary ECU, by refactoring the Uptane implementation in order to isolate the critical components constituted in the Full verification process.
2. Design an API for communication between the created enclave and non-trusted components.
3. After the refactoring, the critical Primary components that perform Full Verification are going to be isolated within the enclave.

4. Finally, we provide remote attestation of the Primary Full Verification so that it can be verified during Secondary Partial Verification.

3.3 Extending Uptane client using TEE

In this section, we discuss the process of extending an Uptane client using a TEE. Along the way, we compare the differences between the standard and our proposed approach.

3.3.1 Refactoring Uptane

In our proposed approach, we will not run all the client code of the Primary ECU inside a TEE, unlike the approach followed in [63]. Assuming that the client's code is portable to the TEE, such an approach can be generally more convenient in terms of enhancing security for the Primary ECU. However, it has natural issues such as having the TEE itself becoming responsible for handling communication with the outer servers and the Secondary ECUs.

Instead, we will separate the Full Verification process of a Primary ECU into two parts: a standard, non-trusted one that interacts with the other actors in system and orchestrates the general client-side steps expected of a client, just as per the standard; and a new trusted one, which we will call the Verification Service (VS) that performs the core verification functions, inside the TEE. The intuition is that the VS will run all the critical functions and pieces of code that are part of verification of metadata, and the regular client will perform the downloading of the metadata and retain control of the control flow of the client process.

After the client downloads the packages from the repositories, the information that was received is serialized to the VS where it is going to be verified. If the verification succeeds, the VS will send a positive message to client and the communication between these two agents goes on until all metadata has been verified. Nonetheless, if the verification fails, the client will receive an error message from the VS and the update cycle will be terminated. The steps presented in diagram 3.7 explain how the Full verification of the Primary ECU is realized with the assistance of the Verification Service:

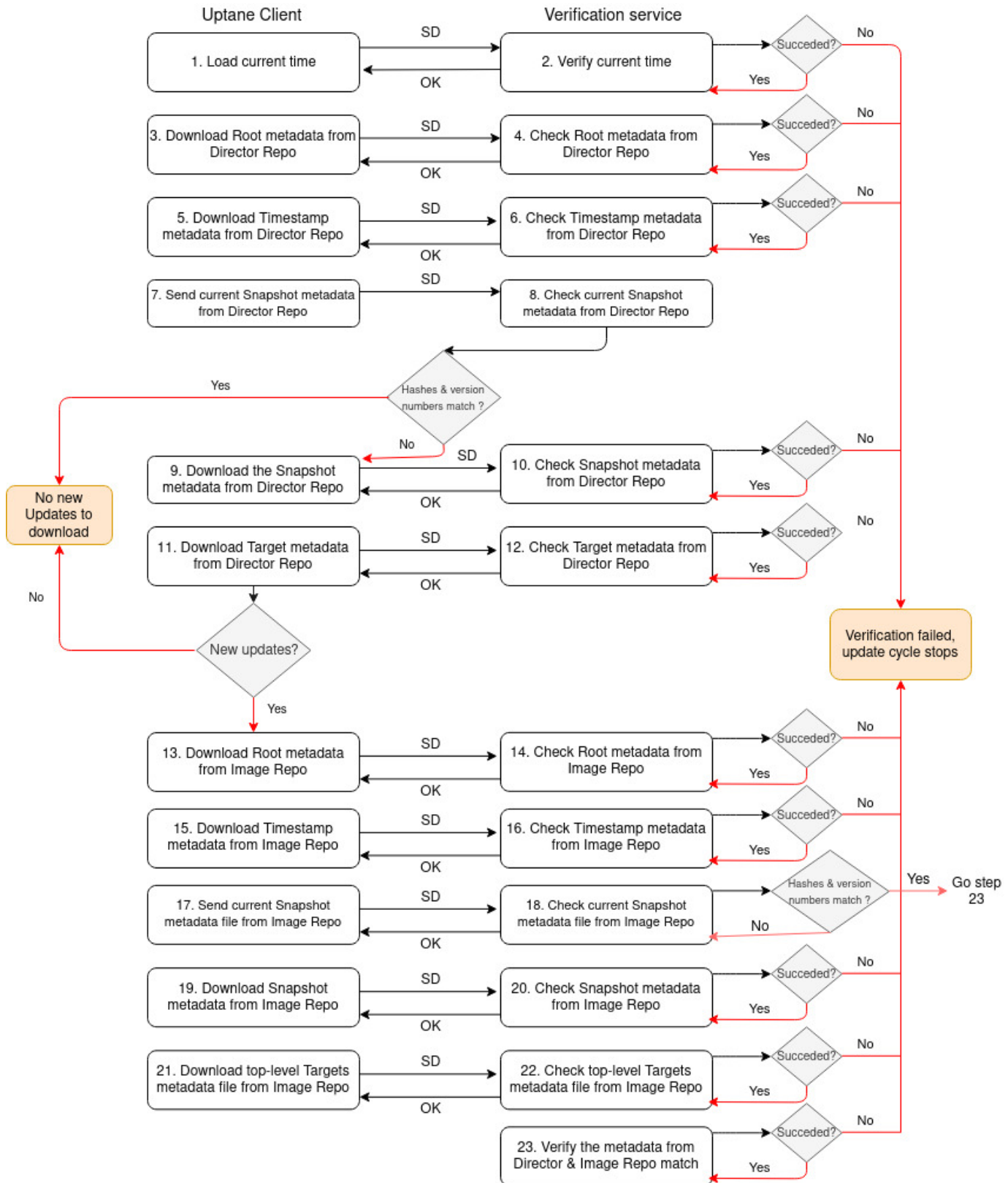


FIGURE 3.7: Modified Full verification process and the communication between client and the VS. The client SD (Serialize Data) to the VS and the service responds with an OK if the verification succeeded. Otherwise, the update cycle stops

3.3.2 Isolate the VS within TEE

With the Verification service built and connected to Client-side ECU via API, the following step is to isolate the verification components within the enclave.

In the Primary ECU, when the Full verification process starts executing, the enclave is created and the Uptane client will send the information yet to be checked to the trusted verification function located in the enclave. Inside the enclave, the metadata is going to be analyzed by the implemented verification methods. If the verification fails, the enclave will send a error message to the Uptane client and the update process will be aborted. Otherwise, the process continues until full verification is finished, after verification is complete, the enclave is automatically destroyed. The following diagram 3.8 shows an overview of how client behaves with SGX.

It is important to remark that, if the client is compromised, it may control the steps performed by the VS and the inputs passed on to the VS. In all cases, the additional guarantee that the VS will provide over the standard Full Verification process, as we will see in Section 4.1. is that the concrete verification trace that took place inside the VS will be attestable by other parties installing the updates.

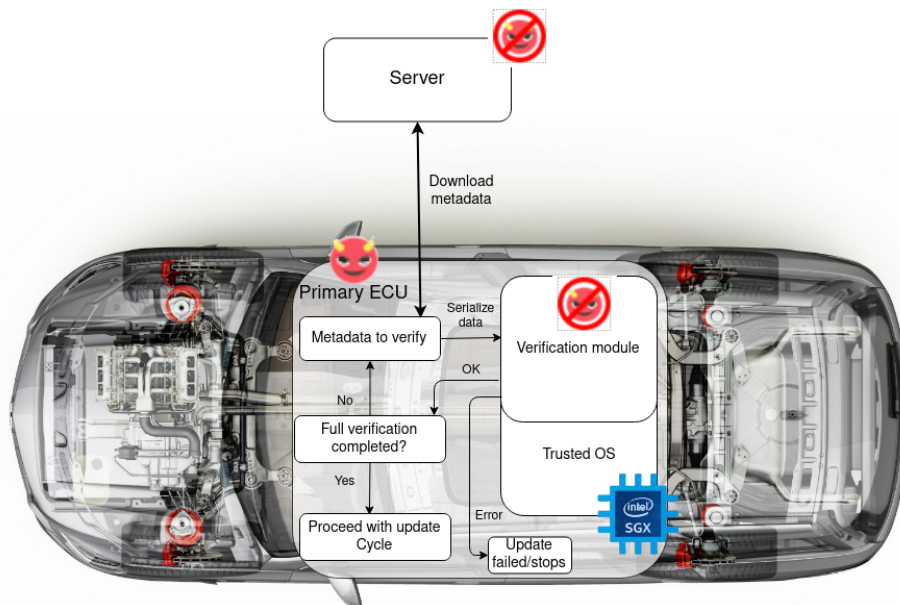


FIGURE 3.8: Communication between Client-side and Enclave in a Primary ECU performing Full Verification, according to our proposed approach.

3.3.3 Integrate remote attestation into Secondaries ECUs

To complete our proposed solution, during the Full verification process, the Uptane client submits an attestation request to the Verification service with a nonce. Then, the Verification Service prepares quote that will be used as evidence to prove its authenticity and send it to the client-side. After that, in the Primary ECU, the quote received by the enclave is sent to the Secondary ECUs that will be updated. Finally, the Secondary needs to validate the process occurred in the enclave by using Quote Verifier Service which returns back whether that the quote received from the Primary ECU can be trusted.

The figure 3.9 shows the complete process of our modified Uptane architecture relying on an SGX enclave with support for remote attestation.

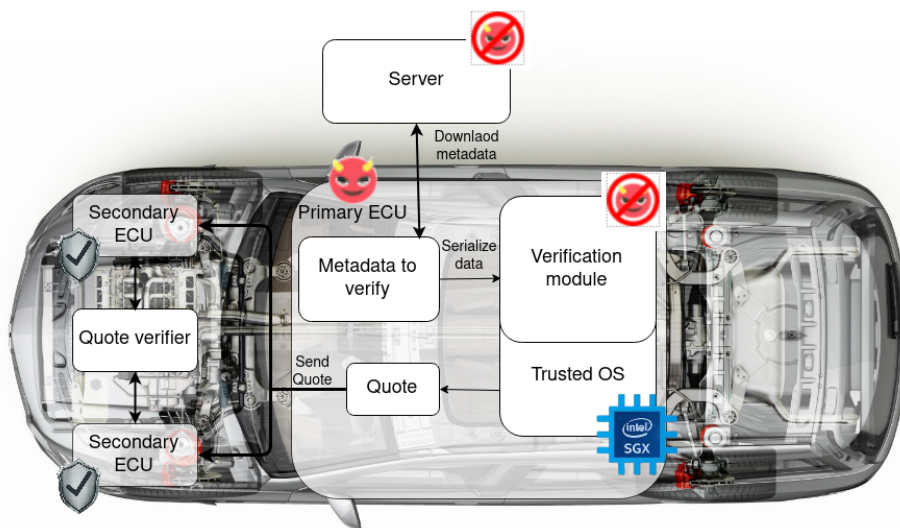


FIGURE 3.9: The remote attestation flow of Intel SGX in our proposed uptane.

Chapter 4

Implementation

In this chapter, we explain the implementation details of the proposed approach. During our implementation, we used specific technical tools like Intel SGX, Gramine and GRPC to achieve our objective. We will also explain the rationale for their use.

4.1 Why SGX?

To implement our proposed approach, we leveraged Intel SGX as our TEE support. Before we explain the reason why we chose this technology, consider the framework of Mukherjee et al. [63], where they implemented the Uptane framework entirely inside a TEE using ARM Trustzone. The following diagram 4.1 is an overview of their proposed update architecture. This proposal proved to be more effective in terms of security, compared to the traditional Uptane, by running the whole Uptane client inside the TEE/Trusted OS.

Nonetheless, their framework only considered the case of a single ECU performing Full Verification. Their scenario did not consider remote attestation of the Full Verification process to another entity, such as a Secondary ECU, and therefore security for Secondary ECUs would impose that they run TEE-enabled Full Verification, as for Primary ECUs.

On the other hand, since Intel SGX provides remote attestation services, it will be possible to transmit the result of a TEE-enabled Full verification process to the targets ECUs, preserving the trustworthiness of the verification process.

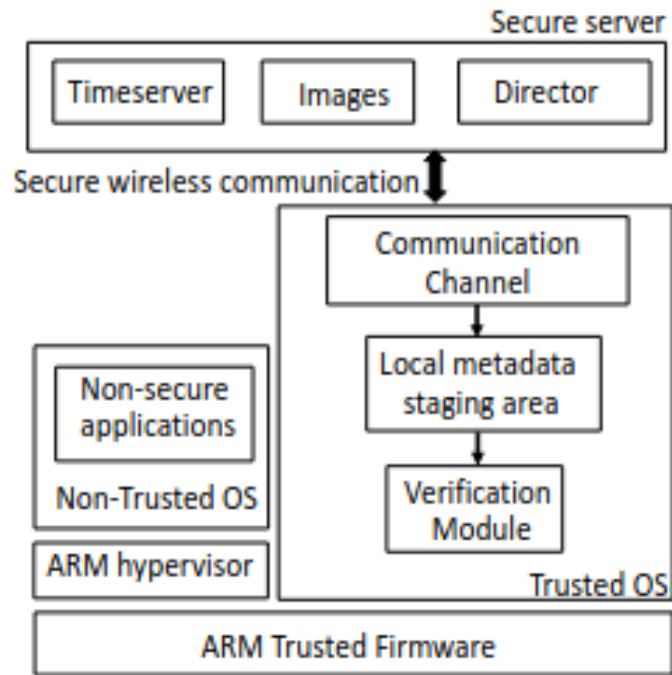


FIGURE 4.1: Modified update model proposed by Mukherjee et al. [63] which is divided into a secure Uptane server and a modified Uptane client-side implementation where it can run trusted applications within the trusted OS inside TEE.

4.2 Why we used Gramine?

The main technical challenge that we faced during this journey was to decide how to isolate the VS (Verification service) inside an enclave. The standard SGX API is written in C/C++, and the Uptane client that we adapted is written in Python. Translating all VS function code to C/C++ would require non-trivial effort, and could also introduce errors in the implementation. To avoid the limitations of rewriting we used a security isolation LibOS with purpose of code-reuse approach.

Gramine [75, 76] (formerly Graphene [77]) is a Library OS that facilitates the execution of existing unmodified applications in SGX enclaves. Gramine also performs cryptographic verification at the untrusted host interface in order to increase security. A Gramine application is accompanied with a manifest file that describes the its security configuration and isolation policies while running inside Gramine.

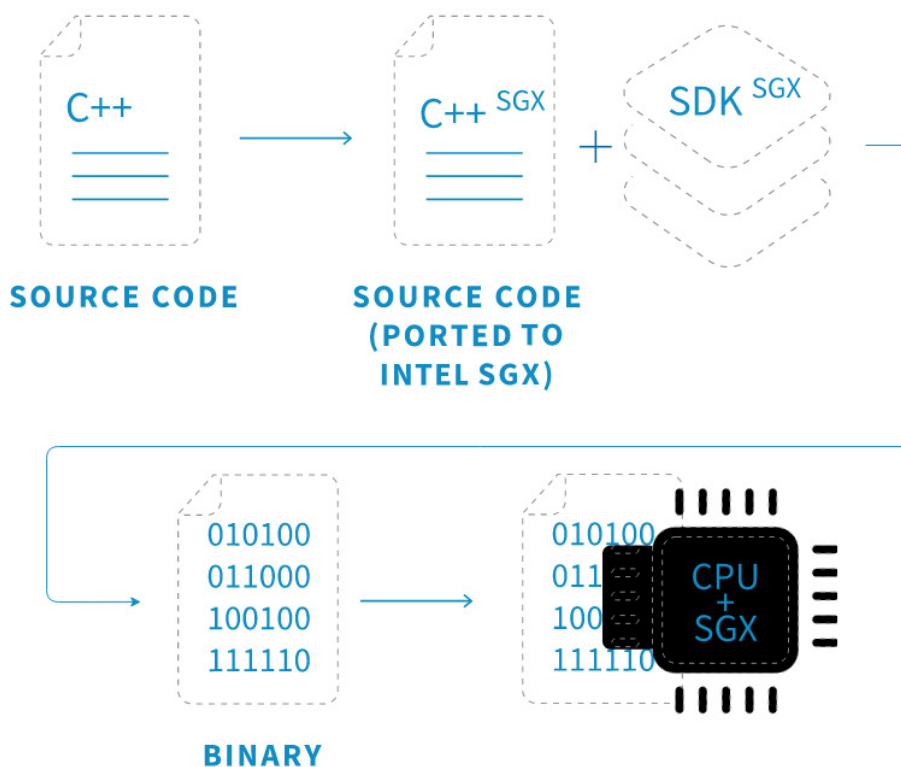


FIGURE 4.2: Regular integration of Intel SGX [75]

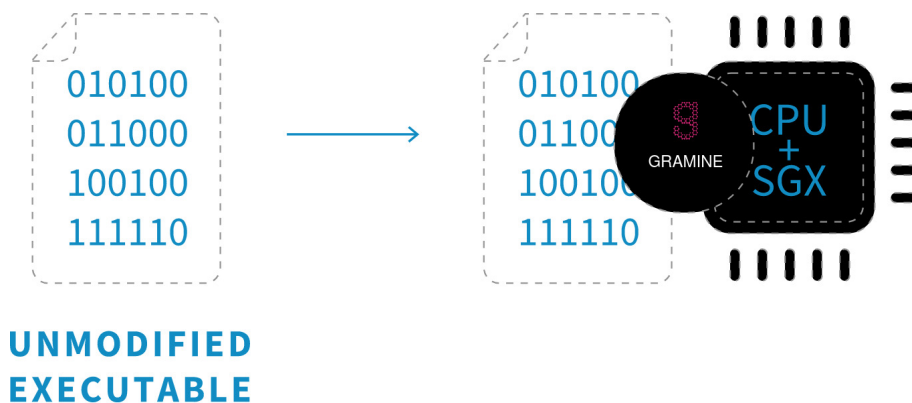


FIGURE 4.3: Integration of Intel SGX with Gramine [75]

Before using Gramine, it is necessary to generate an RSA 3072 key suitable for signing SGX enclaves and stores in a special pseudo-filesystem (`HOME/.config/gramine/enclave-key.pem`), in order to guarantee the integrity of the outputs produced by the application inside the enclave.

To invoke Gramine with a Python application, it is necessary to configure the manifest file which is a simple configuration file written in the standard TOML format [78] that

specifies all the additional dependencies that need to be trusted and the file access permissions assigned to the Gramine executable. We provide an example in 4.5 . Once provided all this information, to make sure the interactions between the enclave and the untrusted application are safe, Gramine intercepts requests (system calls) by calling Platform Adaption Layer (PAL), which in turn calls SGX drivers to initialize the enclave. These requests that involves talking to the host OS, uses read and write operations for extra security precautions. These operations apply real-time encryption, so that the data stored on the host cannot be read without an encryption key [79].

To execute Gramine application we have two alternative modes: `gramine-direct` (without SGX) just for ease of debugging and `gramine-sgx` (with SGX) that loads the program onto SGX and runs Gramine inside the enclave.

```
# generate Gramine- and SGX-specific files
$ gramine-manifest helloworld.manifest.template helloworld.manifest
$ gramine-sgx-sign --key private.pem --manifest helloworld.manifest \
  --output helloworld.manifest.sgx
$ gramine-sgx-get-token --sig helloworld.sig --output helloworld.token
# run Gramine in direct mode (non-SGX, simulation)
$ gramine-direct helloworld
Hello, world
# run Gramine in SGX mode
$ gramine-sgx helloworld
Hello, world
```

FIGURE 4.4: Example of running application with Gramine [80]

```
libos.entrypoint = "helloworld" — Executable to load into SGX enclave

loader.env.LD_LIBRARY_PATH = "/lib" — Envvars are overwritten/propagated

fs.mount.lib.type = "chroot"
fs.mount.lib.path = "/lib" — Restrict mount points to a small subset of
fs.mount.lib.uri = "file:/usr/local/lib/gramine" host-OS directories

sgx.enclave_size = "1024M" — SGX-specific limits like the maximum enclave size and
sgx.thread_num = 8 number of threads

sgx.trusted_files = [ "file:/usr/local/lib/gramine/libc.so.6" ]
...
SGX-specific trusted files (securely hashed and verified
at load time
```

FIGURE 4.5: Security policies in manifest file of Gramine [80]

4.3 Remote Attestation

Gramine provides all functionality required for ensuring end-to-end protection of workloads, including attestation. As explained before, attestation is a security mechanism that allows verifying if the outputs produced by the enclave are trustworthy. In addition to this assurance, a Secure Channel needs to be created for trusted communication between the Primary ECU and the remote TEE. In many cases, the Primary ECU also wants Secret Provisioning to deliver safely the secret keys and other sensitive data to the remote TEE.

The following three levels of attestation flows are provided by Gramine:

1. **Local Attestation and Remote Attestation** are exposed to the application via a special pseudo-filesystem (`/dev/attestation`). SGX local attestation in Gramine generates a SGX Report as an attestation evidence. Gramine remote attestation uses the Intel SGX PSW's AESM service and EPID service (uses Intel Attestation Service (IAS)) or ECDSA/DCAP service (provided by special DCAP libraries).
2. **Secure Channel** are communication channels for trusted transmission of arbitrary data between a TEE and a remote trusted party.
3. **Secret Provisioning** is a mechanism to deliver secrets contents (such as encryption keys, passwords, etc.) from a remote trusted party inside a TEE.

The diagram 4.6 shows how the EPID attestation cycle runs under Gramine. Additionally, the descriptions of each stages are as follows:

1. The cycle starts with the enclavized user application opening the special file in order to write the report (`/dev/attestation/user_report_data`).
2. Gramine, running under SGX, uses the EREPORT hardware instruction to generate a SGX Report.
3. After the SGX report is generated, the application opens a special file `/dev/attestation/quote`.
4. Under the SGX, Gramine request for the SGX quote to Quoting Enclave.
5. Only on the first deployment, the Quoting Enclave request fo EPID key from the Provisioning Enclave.

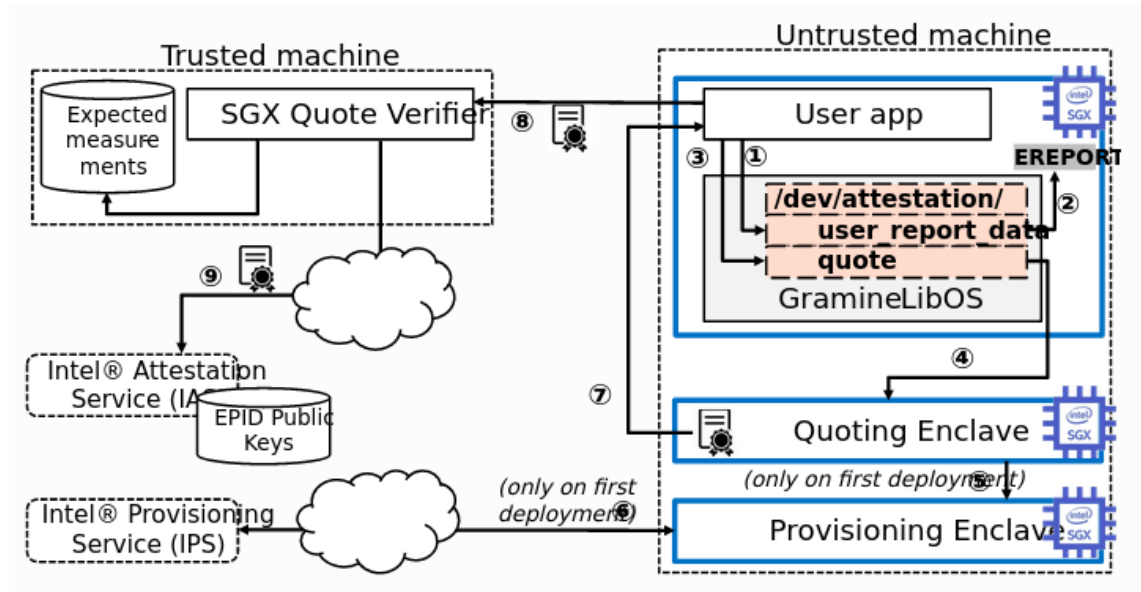


FIGURE 4.6: Gramine EPID remote attestation. [81]

6. The Provisioning Enclave requests the EPID key related with this SGX machine from the IAS (Intel Provisioning Service).
7. The Quoting Enclave generates the SGX quote and transmits to the enclavized user application.
8. The application stores this SGX quote in its enclave memory to be verified later.
9. The remote user consults the IAS to verify if the quote is trustworthy.
10. Finally, if this verification procedure was successful, the remote user can trust the SGX enclave on the untrusted environment.

The diagram 4.7 represents DCAP based remote attestation under Gramine. DCAP flows are very similar to EPID flows, but the big difference is that DCAP uses the classic PKI with X.509 certificate chains to verify the Quote instead of consulting the Intel Attestation Service. The steps 1-4 are the same as the EPID flows. However, the Quoting Enclave communicates with the Provisioning Certification Enclave (PCE) rather than the Provisioning Enclave (step 5), and in step 6, instead of consulting IAS, the PCE uses another Intel service called Intel Provisioning Certification Service (PCS) to obtain the attestation certificates. The Primary ECU periodically fetches and caches the DCAP attestation certificates on a local machine (step 0) rather than consulting a web service from Intel each time a new SGX quote arrives. Finally in step 9, when the Primary ECU receives

the SGX quote, it compares the certificates embedded in the quote against these cached DCAP attestation certificates.

We could not implement DCAP remote attestation due to execution issues with Gramine. EPID attestation could not be an option either since the relying party verifies the Quote at runtime by contacting Intel Attestation Service (IAS) and according to the Uptane standard, the Secondaries should not communicate outside the vehicle. Unlike DCAP where internet-based services are not required at runtime, is the only option that favours our proposed solution.

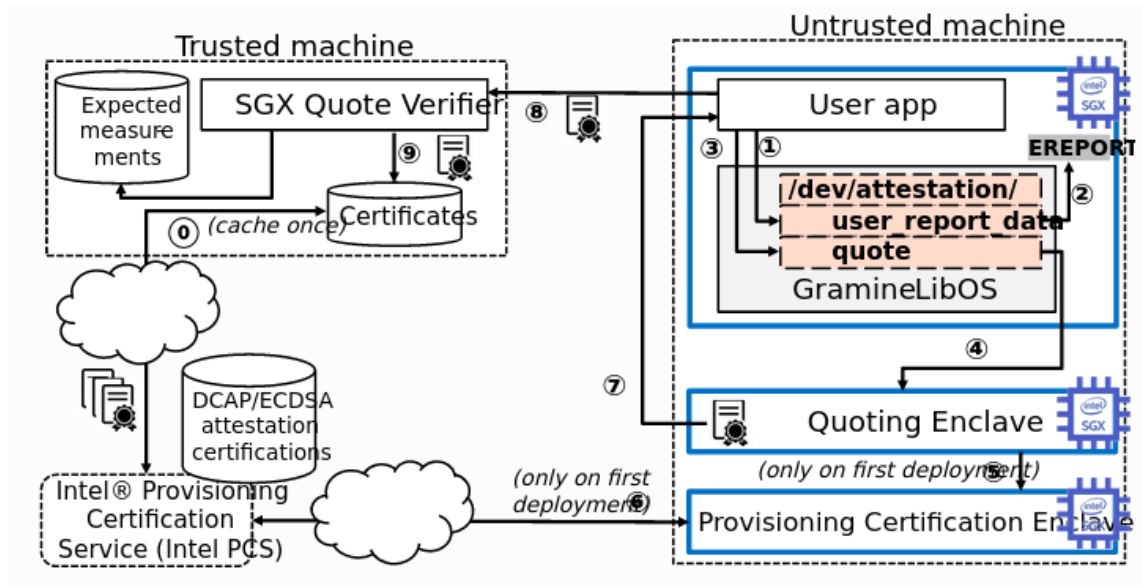


FIGURE 4.7: Gramine DCAP based remote attestation. [81]

4.4 Why we used GRPC?

Traditionally, the code to be run in a SGX enclave must be statically compiled to guarantee code integrity. However, looking back to our proposed Uptane 3.7, the Verification service (VS) is designed to interactively exchange information with Primary ECU.

For that purpose, our enclave runs a statically-defined interactive server application that listens to requests from the Uptane client, executes the respective verification, and replies with the outputs of verification.

In order to design such an interactive server application, we employ gRPC as our communication service. In gRPC, a client application can directly call a method on a server application on a different machine by using the protocol buffer (i.e., Proto Request/Response), creating then distributed applications and services without much effort.

The Protocol buffer is a binary format that efficiently serializes structured data with minimal overhead. The service interface definition is specified in a proto file (file.proto). The proto file defines which method parameters and message formats are going to be used when the method is invoked.

The following code 4.1 and 4.2 shows the service's interface used in Uptane from a file.proto. The first figure represents two types of messages: the Request which contains a collection of data structures called fields that the Primary ECU will serialize and the Response that has two fields, an ID and a result message (response_data) of the service done in the Verification service.

```
1 service GRPCDemo {
2
3 message Request {
4     int64 client_id = 1;
5     string role_name = 2;
6     map<string, string> mymap = 3;
7     bytes file_object = 4;
8     int64 length = 5;
9     string metadata_dict1 = 6;
10    string metadata_dict2 = 7;
11    string metadata_dict3 = 8;
12    string metadata_dict4 = 9;
13    string repository_name = 10;
14    bool check = 11;
15    string metadata_directory_c = 12;
16    string metadata_directory_p = 13;
17    string metadata_file = 14;
18    string remote_filename =15;
19    int64 version = 16;
20    string filepath = 17;
21    string filename = 18;
22    string version_cur = 19;
23    string version_new = 20;
24    string current_repository_metadata = 21;
25    string previous_repository_metadata = 21;
26 }
```

LISTING 4.1: Definition of the message format/type of client and server.

The second figure represents a service (GRPCDemo) which consists a collection of services with RPC method parameters. These services takes an input of type Request and returns a Response.

```

1 service GRPCDemo {
2
3     rpc server_check_hashes (stream Request) returns (Response);
4
5     rpc server_check_file_length (Request) returns (stream Response);
6
7     rpc _server_ensure_not_expired (stream Request) returns (Response);
8
9     rpc _server_verify_metadata_file (stream Request) returns (Response);

```

LISTING 4.2: Example of some remote methods for verification the data transmitted by the client (Request) to the Verification service and the verification service returns with a result (Response).

As previously analyzed in 3.7, we implemented a bi-directional streaming method where the client (Primary ECU) and the Verification service use a read-write stream to send a message sequence. In 4.3 shows an example method where the Primary ECU builds the the message format which contains the information that needs to by analysed in the Verification service. In the end of the method, the client ECU will wait for the result from the Verification Service.

```

1 def client_streaming_method(self, stub, byte, file_hashes):
2     print("-----Call check_hashes Begin-----")
3     # create a generator
4     def request_messages():
5         request = demo_pb2.Request(
6             client_id=CLIENT_ID,
7             file_object=byte,
8             mymap=file_hashes
9         )
10        yield request
11    #response from the server
12    response = stub.ClientStreamingMethod(request_messages())
13    print("resp from server(%d), the message=%s" %
14          (response.server_id, response.response_data))
15    print("-----Call check_hashes Over-----")

```

LISTING 4.3: Example of a gRPC method from client (Primary ECU)

The Verification Service, whose interface is demonstrated in 4.4, will receive input data from the client and is responsible for executing an specific verification method according to the client ECU needs. If no errors occurred, the Verification service transmits a positive result to the Primary ECU.

```
1  # In a single call, the client can transfer data to the server
2  def server_check_hashes(self, request_iterator, context):
3      print("Verification Service called by client...")
4      #print data from client Primary ECU
5      for request in request_iterator:
6          print("recv from client(%d), FILE_OBJECT--> %s ___ TRUSTED_HASHES=
7              %s" %
8                  (request.client_id, request.file_object, request.mymap))
9
10         bytess = io.BufferedReader(io.BytesIO())
11         bytess.write(request.file_object)
12
13         flag_check_hash = self.check_hashes(bytess, request.mymap)
14         #send the result to the Client
15         response = demo_pb2.Response(
16             server_id=SERVER_ID,
17             response_data="VERIFICATION RESULT= %s" % (flag_check_hash))
18     return response
```

LISTING 4.4: Example of a gRPC method from server (Verification service)

Chapter 5

Evaluation

The current chapter provides a preliminary evaluation of our proposed Uptane framework. In the first section, we compare the performance of our solution with the original Uptane client implementation. In the second section, we discuss and compare the security aspects of our proposal.

5.1 Performance Evaluation

To evaluate the performance of our proposal, we will measure the time that it takes for the Primary ECU to fully verify a sample update with a size of 15 bytes. We will evaluate our Uptane proposal under four different scenarios:

- Running the original Uptane client with no separation between client and Verification Service;
- Running the Verification Service as a native Python application;
- Running the Verification Service without SGX support, using `gramine-direct`;
- Running the Verification Service with SGX support, using `gramine-sgx`.

These four scenarios seek to evaluate the gradual performance impact of using gRPC, gramine, and of using SGX. Although preliminary, and far from an exhaustive performance evaluation, this measure is important to provide a general intuition for the performance impact of our design, since we must bear in mind that efficiency is always a critical aspect for end-to-end solutions for OTA updates.

5.1.1 Uptane with native client

In traditional Uptane, the implementation of the Full verification process executed in the client-side performed by the Primary ECU, it consists of three main stages. Each stage is composed by methods that are part of the Full Verification process. Below we provide a description of each stage:

1. `refresh toplevel metadata`: Refreshes client's metadata for the top-level roles: root, targets, snapshot, and timestamp.
2. `get validated target info`: This method returns trustworthy target information for the given target file (specified by its file path), from the Director and validated against the Image Repository.
3. `download target`: After we possess the target information for all targets listed by both the Director and the Image Repository, this call performs the actual download and verification of the specified target and only keep each if it matches the verified target information.

We measured three stages that perform retrieval and validation of metadata and data provided by Director repository.

5.1.2 Uptane with Verification Service

Regarding the use of a remote service, this will affect directly the execution time, as the Primary ECU does significant number of message exchanges with the Verification service, the execution time may be longer, which can be noticed during the verification process. That's why we elected gRPC, it uses Http/2 [82], which is considered a faster binary serialization protocol resulting a positive efficiency of message-passing, we expect that this RPC framework is a suitable solution so that there is no significant performance decrease.

5.1.3 Uptane with Verification Service inside Gramine

We can conclude that the performance impact of the Verification service is not significantly high compared to Native Uptane, and Verification service under `gramine-direct` mode consumes almost the same time as Verification service without `gramine`.

5.1.4 Uptane with Verification Service inside Gramine and SGX

Finally, the most significant performance test consists in executing the Uptane client plus the Verification service inside the Enclave with `gramine-sgx` support. The results of this experiment are crucial because SGX imposes a heavy performance penalty upon switching between the client application and the verification functions running under enclave, ranging from 10,000 to 18,000 cycles per call [83] depending on the call mechanism used. This penalty affects applications efficiency running under SGX [83].

In our case of study, we recommend that the Primary ECU should have a minimum EPC (Enclave page Cache) size required (depending on the CPU type), in order to not impair significantly the performance. Swapping EPC pages is expensive, costing hundreds of thousands of cycles per swapping operation [83]. If the program consumes more than the allowed EPC, the EPC will oversubscribing and Linux driver ends up paging the EPC, causing an additional overhead, which increases depending on the number of threads running inside the enclave taking more time to build an enclave and disrupt efficiency [84].

Scenarios	refresh toplevel metadata	get validated target info	download target
Native Uptane	0.02945	1.00098	0.00762
Uptane+VS	0.18728	1.09946	0.02071
Uptane+VS under <code>gramine-direct</code>	0.19471	1.10729	0.02214
Uptane+VS under <code>gramine-sgx</code>	0.51394	1.41043	0.06555

TABLE 5.1: Time performance results (in seconds) of traditional Uptane verification stages compared to our final solution

5.1.5 Remote Attestation

As previously mentioned, our implemented does not support remote attestation attestation, hence, we cannot evaluate with valid results of our modified Uptane with remote attestation. Remote attestation with EPID would not satisfy our proposed solution, because it is not recommended that the Secondaries ECUs communicate outside the vehicle, and the verification of the Quote, in this type of attestation, requires a relying party to have internet access. On the other hand, DCAP allows the ECU to build and deliver their

own attestation service rather than using the remote attestation service provided by Intel. Furthermore, we expect an improvement of availability and performance compared to EPID [53, 55, 85, 86]. In addition, it improves privacy by making trust decisions in-house [53, 55, 85].

5.1.6 Summary

In table 5.1 we are able to examine the performance impact of each scenario. When utilizing gRPC, we concluded that the performance compared to the Native Uptane dropped around 26%. When running Uptane with Verification service under gramine-direct, the performance was almost the same as the running without gramine, the execution time drooped around 1.3%. Comparing the Uptane with Verification service running under gramine-sgx we can conclude an increase of around 52% in execution time. Finally in our proposed solution, there was an increase of around 91% compared to the the Native Uptane Full verification process.

In terms of remote attestation, the runtime execution of Uptane with Verification service running under gramine-sgx would grow if the remote attestation was performed. According to Gramine’s support team [87], the execution time of remote attestation does not depend on the target size, in other words, the Quote generation and verification time is a constant. The dissertation written by Reis [88] which implement a secure system based on Intel-SGX using an unmodified application (REDIS), the remote attestation process was analyzed with SCONE [89] mechanism using DCAP. The presented results, referenced in the table 5.2, conclude that the attestation mechanism to a Proxy instance container induces in $\approx 1,05$ extra seconds, while for the Redis $\approx 1,23$ seconds.

	No Attestation	Attestation
Redis	0,14s	1,37s
Proxy	62,07s	63,12

TABLE 5.2: Attestation Impact upon boot [88].

Another research made by Yulianti [90], the authors implemented a flexible and secure environment using Intel-SGX where multiple users can collaborate in sharing their data. In this project, they evaluate Occlum [91] as a memory-safe Library Operating System (OS) that enables secure and efficient multitasking on Intel SGX. During the evaluation

of the attestation process, three types of frameworks were measured: baseline Linux, Occlum, and Graphene-SGX. The analyses done on baseline Linux, the generation of the Quote is not performed (because it occurs outside the enclave), so it only checks the runtime execution of Quote verification using X.509 extension fields. The results represented in table 5.3, we can conclude that takes ≈ 0.01301 seconds to perform the verification of the Quote.

Value: Frameworks	Linux (ms)	Occlum (ms)	Graphene-SGX (ms)
Minimum	12.791	144.224	4966.162
First Quartile	13.199	286.966	5464.675
Median	13.362	419.42	5494.199
Third Quartile	13.635	594.7	5520.352
Maximum	22.922	1888.293	5718.175
Mean	13.591	465.25	5485.093
Std. Deviation (SD)	0.856	229.849	60.753
Mean after SD	13.575	461.662	5483.648

TABLE 5.3: Execution Time of Attestation Process [90].

To summarize, despite our solution decrease significantly the performance of the Primary ECU, at least, we believe that it is possible save computational resources from the Secondaries ECU by only verify the Quote instead of performing the traditional Full Verification. According to the results obtained in table 5.3 and assuming that the estimated time of Quote verification (≈ 0.01301 seconds) is the same as our solution, we conclude that it is faster than the execution time of the Full Verification process (≈ 1.03805) seconds. Not forgetting that during our evaluation, the size of the analyzed target was basic, if it were more complex, the execution time of the Full Verification would increase while the Quote verification, as mentioned before, would be constant.

5.2 Security Evaluation

Remembering Chapter 3.2, we cited potential update attacks that traditional Uptane may not be capable to protect in certain scenarios. We remarked that the Secondary ECUs that perform Partial verification are the most vulnerable, being susceptible to more security attacks in comparison to ECUs that perform Full Verification. Our objective, described before, is to bring the guarantees of the Full verification method to all Secondary ECUs,

thus being able to avoid update attacks that Partial verification cannot particularly defend against.

5.2.1 Security flaws in traditional Uptane

In this subsection, we will focus on security flaws that can affect the security of software updates in Secondary ECUs that perform Partial verification. Some types of attacks like physical attacks, compromise of the build system, remote exploits to compromise an ECU and random failures are considered outside the scope.

Figure 3.6 has previously documented the security attacks to which secondaries are vulnerable if attackers have compromised the Primary. The most notable difference between the Secondaries that perform the Full verification and the Secondaries that perform Partial verification is that even if attackers have compromised the director keys, then attackers are able to execute rollback and arbitrary software attacks on all partial verification secondaries on all vehicles. This is because these secondaries depend upon the Primary verification method (which is compromised) to prevent these attacks by comparing the director to the targets metadata. In contrast, the Full verification Secondaries only are affected by this attacks described above if the attacker compromises the root key which is supposed to be the hardest to compromise.

5.2.2 Additional protection in our proposed Uptane

Finally, we state the security goals that we believe are achievable with our proposed solution. Unlike our modified Uptane, the solution proposed by Mukherjee et al. [63] offers better guarantees for the Primary ECU itself, since the entire Client-side runs inside the TEE. In the case of our solution we also provide extra security to the Primary itself, we enhance the security of the Full verification process by isolating the core functions inside the enclave. Although, any internal or external attack on the rest of the software or system can affect the integrity of our modified framework, our Verification service receives inputs from (and returns outputs to) the untrusted environment, where the attacker always has capabilities of performing Denial of Service attacks. This inconvenience may occur in our solution, on contrary to the Uptane with ARM TrustZone proposal which despite running networking within the TEE, thus losing security guarantees, still it is more effective against these flaws. However, the solution proposed by Mukherjee et al. [63] does

not give additional guarantees to the secondaries that only perform partial verification justifying the same initial problem that we sought to explore and mitigate.

As explained and described in Figure 3.9, the Primary ECU sends the downloaded metadata to the verification service under SGX, and this service will perform the verification methods pertaining to the Full verification process. In order to enhance confidence that the intended software is securely running inside an enclave, Intel SGX provides an attestation mechanism.

At the end of the verification process, the Primary ECU requests for attestation to the verification service, and the verification service requests that its SGX enclave produces an attestation. In SGX, this happens in a two-part process: the verification service sends a local attestation from its enclave to a Quoting Enclave; the Quoting enclave then verifies the local attestation and converts it into a Quote by signing the local attestation. The Quote is returned to the verification service and, finally, forwarded to the Primary ECU. The Primary ECU then transmits the Quote to the Secondaries ECUs where it will perform a Quote verification.

With this Quote validated, we can guarantee that the verification process that took place in the primary was successful. The final step is to send the Quote to the Secondaries so that they can attest that the full verification was correctly performed, in which case they can skip partial verification and start the installation of the software update.

5.2.3 Summary

Concluding this assessment, with our solution, all Secondaries ECUs will now have the same Security attacks that affect the Primary ECU, regardless of whether the Primary is compromised or not (figure 3.4). The notable advantage over the traditional Uptane is that the rollback and arbitrary software attack are not going to affect any Secondary ECU (in case the attacker just compromise the director key).

Not only our solution is effective in securing the Secondaries, it also improved the security of the Primary (not as much as Mukherjee et al. [63] proposal). For example, an arbitrary software attack can only attack in the untrusted world, since the code running inside the enclave is fixed, and with Verification service support, eavesdrop attack is slightly more limited, the attacker can only see Input and Output of the Verification service, but cannot inspect internal operations of the verification process.

With our solution, we expect that it is possible to give all Secondary ECUs full verification guarantees without having to execute the Full verification process, thus reducing the number of security attacks.

Chapter 6

Conclusion

In this thesis, we explored the combined use of the Uptane framework with Trusted execution environment technologies, in order to enhance security for the more vulnerable secondaries ECUs. We showed that, despite the usage of Intel SGX in our modified Uptane requires a more significant amount of resources and induces a decrease in terms of performance, it is feasible to mitigate some typical security attacks to the most vulnerable Secondaries ECUs.

During the initial research phase of this thesis, we invested a lot of effort into framing the Intel SGX technology with the Uptane framework. As explained before in Section 4.2, manually translating all critical verification methods to C/C++ would be a laborious and error-prone process. We found a compromise solution using the lightweight library OS Gramine [75] which was the most effective solution to prototype our framework [92]. Thanks to Gramine, we managed to get further in this investigation making possible to develop a preliminary prototype that instantiates our idealized design. However, Gramine was used only for experimental purposes, it is not recommended executing it in real world due to the negative impact in terms of performance. Our main objective was to focus on proving that it is possible to give Full verification guarantees for all Secondary ECUs.

Although Gramine has facilitated the combination of these technologies, we faced new challenges that brought some obstacles to the realization of this thesis. The main difficulty was performing remote attestation with Gramine. Due to problems during the installation of the DCAP infrastructure, we could not test the practical use of this type of remote attestation, which would be the most suitable for Secondary ECU verification.

6.1 Future Work

In terms of future work, it would be interesting to overcome some of the limitations identified before, like performing the DCAP remote attestation method with our solution.

Another possibility for a more realistic implementation would be to go through the effort of implementing an Uptane client in C/C++. As mentioned in Background and Related Work (chapter 2), Aktualizr [35, 36] could be a suitable solution in this case since the implementation is written in C++. Using a lower-level language would likely bring significant performance gains over an interpreted language such as Python. Moreover, despite the convenience of Gramine, removing that layer of indirection would also likely translate into significant performance gains. However, this project wasn't as satisfactory as the traditional Uptane, and for that reason we didn't leverage Aktualizr.

To conclude this section, we believe that if we merge the benefits of our solution with the contributions of Mukherjee et al. [63] modified Uptane, would be the best of two worlds. In other words, a modified Uptane which the client-side of Primary ECU is isolated entirely inside a TEE and uses remote attestation to provide Full verification guarantees for the Secondaries ECU, would significantly enhance the security of all ECUs.

Bibliography

- [1] A. Birnie and T. v. Roermund, "A multi-layer vehicle security framework," *white paper*, Date of release: May, 2016. [Cited on page 1.]
- [2] B. Kim and S. Park, "Ecu software updating scenario using ota technology through mobile communication network," pp. 67–72, 2018. [Cited on pages 1, 6, and 7.]
- [3] Y. Onuma, Y. Terashima, S. Nakamura, and R. Kiyohara, "A method of ecu software updating," pp. 298–303, 2018. [Cited on page 7.]
- [4] Z. Wu, T. Liu, X. Jia, and C. Sun, "Security design of ota upgrade for intelligent connected vehicle," in *Proceedings of the 2021 International Conference on Control and Intelligent Robotics*, ser. ICCIR 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 736–739. [Online]. Available: <https://doi.org/10.1145/3473714.3473851> [Cited on page 1.]
- [5] T. Spring, "Supply chain update software unknowingly used in attacks," 2017. [Online]. Available: <https://threatpost.com/supply-chain-update-software-unknowingly-used-in-attacks/125483/> [Cited on page 2.]
- [6] M. Nestor, "New debian gnu/linux 11 "bullseye" kernel security update fixes 9 vulnerabilities - 9to5linux," 2022. [Online]. Available: <https://9to5linux.com/new-debian-gnu-linux-11-bullseye-kernel-security-update-fixes-9-vulnerabilities>
- [7] R. Montti, "Cloudflare names ovh and hetzner as origins of ddos attack," 2022. [Online]. Available: <https://www.searchenginejournal.com/cloudflare-names-ovh-hetzner-origins-of-ddos-attack/447991/>
- [8] M. Schulze, "Debian investigation report after server compromises, 02.12. 2003," 2003.

- [9] J. Ness, "Flame malware collision attack explained," Tech. Rep., Jun. 2012. [Online]. Available: <http://blogs.technet.com/b/srd...>, Tech. Rep., 2012. [Cited on page 2.]
- [10] A. Hern, "North korean 'cyberwarfare'said to have cost south korea£ 500m," *The Guardian*, vol. 16, 2013. [Cited on page 2.]
- [11] J. Samuel, N. Mathewson, J. Cappos, and R. Dingleline, "Survivable key compromise in software update systems," p. 61–72, 2010. [Cited on page 2.]
- [12] A. Greenberg, "Hackers remotely kill a jeep on the highway—with me in it," *Wired*, vol. 7, no. 2, pp. 21–22, 2015. [Cited on page 2.]
- [13] C. Miller and C. Valasek, "Adventures in automotive networks and control units," *Def Con*, vol. 21, no. 260-264, pp. 15–31, 2013.
- [14] M. Casey, "Fiat chrysler recalls 1.4m cars after hack revelations - cbs news," 2015. [Online]. Available: <https://www.cbsnews.com/news/fiat-chrysler-recall-after-jeep-hack/>
- [15] J. Crosbie, "Tesla thieves are on a high-tech model s stealing spree in europe," 2017. [Online]. Available: <https://www.inverse.com/article/34278-tesla-thieves-model-s-stealing-spress> [Cited on page 2.]
- [16] 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. [Cited on pages [xiii](#), [2](#), [3](#), [8](#), [26](#), [27](#), and [28](#).]
- [17] 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. [Cited on pages [xiii](#), [2](#), [3](#), [8](#), and [9](#).]
- [18] "Uptane design." [Online]. Available: <https://uptane.github.io/design.html> [Cited on pages [xiii](#), [2](#), and [9](#).]
- [19] Uptane Community, "Uptane standard for design and implementation 2.0.0." [Online]. Available: https://uptane.github.io/papers/uptane-standard.2.0.0.html#full_verification [Cited on pages [2](#), [3](#), and [17](#).]

- [20] T. Kim and S. Park, "Compare of vehicle management over the air and on-board diagnostics," pp. 1–2, 2019. [Cited on pages 3, 6, and 7.]
- [21] A. Bindal, S. Mann, B. Ahmed, and L. Raimundo, "An undergraduate system-on-chip (soc) course for computer engineering students," *IEEE Transactions on Education*, vol. 48, no. 2, pp. 279–289, 2005. [Cited on pages 3 and 10.]
- [22] Intel Corporation .2019, "Intel® 64 and ia-32 architectures software developer manuals," 2019. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html> [Cited on pages 3 and 10.]
- [23] C. Miller and C. Valasek, "Adventures in automotive networks and control units," *Def Con*, vol. 21, no. 260-264, pp. 15–31, 2013. [Cited on page 5.]
- [24] S. Poledna, W. Ettlmayr, and M. Novak, "Communication bus for automotive applications," in *Proceedings of the 27th European Solid-State Circuits Conference*, 2001, pp. 482–485. [Cited on page 5.]
- [25] R. Bosch *et al.*, "Can specification version 2.0," *Rober Bousch GmbH, Postfach*, vol. 300240, p. 72, 1991. [Cited on page 5.]
- [26] O. Henniger, "Evita: E-safety vehicle intrusion protected applications," *tech. rep., EVITA*, 2011. [Cited on page 5.]
- [27] T. Chowdhury, E. Lesiuta, K. Rikley, C.-W. Lin, E. Kang, B. Kim, S. Shiraishi, M. Lawford, and A. Wassyng, "Safe and secure automotive over-the-air updates," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2018, pp. 172–187. [Cited on page 6.]
- [28] H. Dakroub and R. Cadena, "Analysis of software update in connected vehicles," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 7, no. 2014-01-0256, pp. 411–417, 2014. [Cited on page 6.]
- [29] T. Hampton, "Know the term: Sota/fota," 2017. [Online]. Available: <https://www.business.att.com/learn/tech-advice/know-the-term--sota-fota.html#> [Cited on page 7.]
- [30] O. Weinmann, "Software updates in the iot: an introduction to sota." [Online]. Available: <https://blog.bosch-si.com/bosch-iot-suite/software-updates-in-the-iot-an-introduction-to-sota/> [Cited on page 7.]

- [31] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," pp. 567–581, 2016. [Cited on page 7.]
- [32] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," pp. 61–72, 2010. [Cited on page 7.]
- [33] T. Galibus, "Securing software updates for trains," in *International Conference on Critical Information Infrastructures Security*. Springer, 2019, pp. 137–148. [Cited on page 8.]
- [34] S. Al Blooshi and K. Han, "A study on employing uptane for secure software update ota in drone environments," in *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE Computer Society, 2022, pp. 1–6. [Cited on pages 8 and 14.]
- [35] G. Cve, "uptane/aktualizr: C++ uptane client," 2022. [Online]. Available: <https://github.com/uptane/aktualizr> [Cited on pages 8 and 52.]
- [36] A. S. Interaction, "Aktualizr documentation." [Online]. Available: <https://advancedtelematic.github.io/aktualizr/index.html> [Cited on pages 8 and 52.]
- [37] Confidential Computing Consortium, "Confidential computing consortium." 2020. [Online]. Available: <https://confidentialcomputing.io/> [Cited on page 9.]
- [38] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," vol. 1, pp. 57–64, 2015. [Cited on page 9.]
- [39] A. A. 2009., "Building a secure system using trustzone technology," 2009. [Cited on page 10.]
- [40] S. Johnson, D. Rizzo, P. Ranganathan, J. McCune, and R. Ho, "Titan: enabling a transparent silicon root of trust for cloud," in *Hot Chips: A Symposium on High Performance Chips*, vol. 194, 2018. [Cited on page 10.]
- [41] A. 2020., "Mac models with the apple t2 security chip," 2020. [Online]. Available: <https://support.apple.com/en-us/HT208862> [Cited on pages 10 and 13.]
- [42] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 187–199. [Cited on page 10.]

- [43] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surv.*, jan 2019. [Cited on page 10.]
- [44] H. Yang and M. Lee, "Demystifying arm trustzone tee client api using op-tee," p. 325–328, 2020. [Online]. Available: <https://doi.org/10.1145/3426020.3426113> [Cited on page 10.]
- [45] M. M. Quaresma, "Trustzone based attestation in secure runtime verification for embedded systems," 2020. [Online]. Available: <https://mquaresma.github.io/assets/dissertation.pdf> [Cited on page 11.]
- [46] M. Chmiel, M. Korona, F. Koziół, K. Szczypiorski, and M. Rawski, "Discussion on iot security recommendations against the state-of-the-art solutions," *Electronics*, vol. 10, no. 15, 2021. [Cited on page 11.]
- [47] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, 2015, pp. 75–88. [Cited on page 11.]
- [48] J. Ahn, I.-G. Lee, and M. Kim, "Design and implementation of hardware-based remote attestation for a secure internet of things," *Wireless personal communications*, vol. 114, no. 1, pp. 295–327, 2020.
- [49] C. Shepherd, R. N. Akram, and K. Markantonakis, "Establishing mutually trusted channels for remote sensing devices with trusted execution environments," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–10. [Cited on page 11.]
- [50] J. Ménétrey, C. Göttel, M. Pasin, P. Felber, and V. Schiavoni, "An exploratory study of attestation mechanisms for trusted execution environments," *arXiv preprint arXiv:2204.06790*, 2022. [Cited on page 11.]
- [51] A. Adamski, "Overview of intel sgx - part 1, sgx internals," 2018. [Online]. Available: <https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html> [Cited on page 12.]
- [52] Intel Corporation, "Strengthen enclave trust with attestation." [Online]. Available: <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html> [Cited on page 12.]

- [53] M. U. Sardar, R. Faqeh, and C. Fetzer, "Formal foundations for intel sgx data center attestation primitives," pp. 268–283, 2020. [Cited on pages 12 and 46.]
- [54] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: Epid provisioning and attestation services," *White Paper*, vol. 1, no. 1-10, p. 119, 2016. [Cited on page 12.]
- [55] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting third party attestation for intel sgx with intel data center attestation primitives," *White paper*, 2018. [Cited on pages 12 and 46.]
- [56] P. INTEL, "Intel software guard extensions programming reference," 2014. [Cited on page 12.]
- [57] Z. Whittaker, "Google launches opentitan, an open-source secure chip design project — techcrunch," 2019. [Online]. Available: <https://techcrunch.com/2019/11/05/google-opentitan-secure-chip/> [Cited on page 12.]
- [58] W. D. Casper and S. M. Papa, *Root of Trust*. Boston, MA: Springer US, 2011, pp. 1057–1060. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_789 [Cited on page 13.]
- [59] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016. [Cited on page 13.]
- [60] D. Kaplan, "Protecting vm register state with sev-es," *White paper*, 2017. [Cited on page 13.]
- [61] R. Brown and W. Ng, "Secure encrypted virtualization," Jun. 14 2018, uS Patent App. 15/375,593. [Cited on page 13.]
- [62] D. Sladović, D. Topolčić, and D. Delija, "Overview of mac system security and its impact on digital forensics process," in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2020, pp. 1236–1241. [Cited on page 13.]
- [63] A. Mukherjee, R. Gerdes, and T. Chantem, "Trusted verification of over-the-air (ota) secure software updates on cots embedded systems," in *Workshop on Automotive and Autonomous Vehicle Security (AutoSec)*, vol. 2021, 2021, p. 25. [Cited on pages xiii, 13, 15, 29, 33, 34, 48, 49, and 52.]

- [64] R. Dhobi, S. Gajjar, D. Parmar, and T. Vaghela, "Secure firmware update over the air using trustzone," vol. 1, pp. 1–4, 2019. [Cited on page 13.]
- [65] D. K. Nilsson, L. Sun, and T. Nakajima, "A framework for self-verification of firmware updates over the air in vehicle ecus," in *2008 IEEE Globecom Workshops*, 2008, pp. 1–5. [Cited on page 14.]
- [66] W. Kanda, Y. Yumura, Y. Kinebuchi, K. Makijima, and T. Nakajima, "Spumone: Lightweight cpu virtualization layer for embedded systems," in *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, vol. 1. IEEE, 2008, pp. 144–151. [Cited on page 14.]
- [67] A. Qureshi, M. Marvi, J. A. Shamsi, and A. Aijaz, "Euf: A framework for detecting over-the-air malicious updates in autonomous vehicles," *Journal of King Saud University-Computer and Information Sciences*, 2021. [Cited on pages xiii, 14, and 21.]
- [68] N. Asokan, T. Nyman, N. Rattanaivanon, A.-R. Sadeghi, and G. Tsudik, "Assured: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2290–2300, 2018. [Cited on page 14.]
- [69] arm Developer, "Trustzone for cortex-m." [Online]. Available: <https://developer.arm.com/Processors/TrustZone%20for%20Cortex-M> [Cited on page 15.]
- [70] K. Eldefrawy, N. Rattanaivanon, and G. Tsudik, "Hydra: hybrid design for remote attestation (using a formally verified microkernel)," in *Proceedings of the 10th ACM Conference on Security and Privacy in wireless and Mobile Networks*, 2017, pp. 99–110. [Cited on page 15.]
- [71] Wikipedia contributors, "Universal asynchronous receiver-transmitter — Wikipedia, the free encyclopedia," 2022, [Online; accessed 30-August-2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver-transmitter&oldid=1102493830 [Cited on page 15.]
- [72] Galois, Inc., "Saw - tutorial." [Online]. Available: <https://saw.galois.com/tutorial.html> [Cited on page 15.]

- [73] “uptane-obsolete-reference-implementation.” [Online]. Available: <https://github.com/uptane/obsolete-reference-implementation#1-starting-the-demo> [Cited on page 22.]
- [74] M. Moore, “Uptane reference implementation pouf,” 2019. [Online]. Available: <https://uptane.github.io/reference.pouf.html> [Cited on pages xiii, 23, and 25.]
- [75] Gramine Project., “Gramine – a library os for unmodified applications,” 2022. [Online]. Available: <https://gramineproject.io/> [Cited on pages xiii, 34, 35, and 51.]
- [76] D. Kuvaiskii, G. Kumar, and M. Vij, “Computation offloading to hardware accelerators in intel sgx and gramine library os,” *arXiv preprint arXiv:2203.01813*, 2022. [Cited on page 34.]
- [77] C.-C. Tsai, D. E. Porter, and M. Vij, “{Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX},” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 645–658. [Cited on page 34.]
- [78] “Toml: Tom’s obvious minimal.” [Online]. Available: <https://tom.io/> [Cited on page 35.]
- [79] P. Marczewski, “Blog – gramine,” 2022. [Online]. Available: <https://gramineproject.io/blog/> [Cited on page 36.]
- [80] C.-C. Tsai, D. Porter, and M. Vij, “Introduction to gramine,” February 2022. [Online]. Available: <https://confidentialcomputing.io/webinar-gramine/> [Cited on pages xiii and 36.]
- [81] Gramine Project., “Attestation and secret provisioning,” 2022. [Online]. Available: <https://gramineproject.io/> [Cited on pages xiii, 38, and 39.]
- [82] M. Nagarajan, “A beginner’s guide to http/2 and its importance — by madhavan nagarajan — the startup — medium,” 2019. [Online]. Available: <https://medium.com/swlh/a-beginners-guide-to-http-2-and-its-importance-700f619bbfe7> [Cited on page 44.]
- [83] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont, “Everything you should know about intel sgx performance on virtualized systems,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–21, 2019. [Cited on page 45.]

- [84] T. Pol and A. Badescu, "Preserving the privacy of data in autonomous cars iot using intel sgx," *19th SC@ RUG 2021-2022*, p. 27. [Cited on page 45.]
- [85] S. Yulianti, "Confidential computing in public clouds: Confidential data translations in hardware-based tees: Intel sgx with occlum support," 2021. [Cited on page 46.]
- [86] A. Dhar, I. Puddu, K. Kostianen, and S. Capkun, "Proximatee: Hardened sgx attestation by proximity verification," ser. CODASPY '20, 2020, p. 5–16. [Cited on page 46.]
- [87] "Gramine users." [Online]. Available: <https://groups.google.com/g/gramine-users> [Cited on page 46.]
- [88] J. C. C. Reis, "Tredis—a trusted full-fledged sgx-enabled redis solution," Ph.D. dissertation, 2021. [Cited on page 46.]
- [89] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumar, D. O'keeffe, M. L. Stillwell *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 689–703. [Cited on page 46.]
- [90] S. Yulianti, "Confidential computing in public clouds: Confidential data translations in hardware-based tees: Intel sgx with occlum support," 2021. [Cited on pages 46 and 47.]
- [91] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970. [Cited on page 46.]
- [92] "Is it possible to write an untrusted application in python* and enclave in c++?" 2022. [Online]. Available: <https://www.intel.sg/content/www/xa/en/support/articles/000090322/software/intel-security-products.html> [Cited on page 51.]