Informatics

TECHNISCHE UNIVERSITÄT DARMSTADT

Privacy-Preserving Remote Attestation Protocol

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

IT-Sicherheit

by

Dominik Roy George, BSc
Registration Number 2841746

to the Department of Computer Science

at the TU Darmstadt

Advisor:	Prof. Dr. Michael Waidner
Assistance:	Prof. Dr. Christoph Krauß
	Michael Eckel, M. Sc.

at the TU Wien

Advisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Edgar Weippl
Assistance:	Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Vienna, 24th June, 2021

Dominik Roy George   Michael Waidner

Double Degree IT Security between Technische Universität Darmstadt and Technische Universität Wien
www.tu-darmstadt.de ▪ www.tuwien.at

Informatics

TECHNISCHE UNIVERSITÄT DARMSTADT

Privacy-Preserving Remote Attestation Protocol

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Dominik Roy George, BSc
Registration Number 01525091

to the Faculty of Informatics

at the TU Wien

Advisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Edgar Weippl
Assistance:	Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

at the TU Darmstadt

Advisor:	Prof. Dr. Michael Waidner
Assistance:	Prof. Dr. Christoph Krauß
	Michael Eckel, M. Sc.

Vienna, 24th June, 2021

Dominik Roy George   Edgar Weippl

Double Degree IT Security between Technische Universität Darmstadt and Technische Universität Wien
www.tu-darmstadt.de ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dominik Roy George, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. Juni 2021

Dominik Roy George

Acknowledgements

I would like to thank my supervisor at TU Darmstadt Prof. Dr. Michael Waidner and my co-supervisors at Fraunhofer Institute for Secure Information Technology (SIT) Prof. Dr. Christoph Krauß and Michael Eckel, M. Sc.. Further, I would like to thank my supervisor at TU Wien Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Edgar Weippl and my co-supervisor Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik. Due to all of my supervisors and co-supervisors of both Institutions, I could finish this double degree master program. Moreover, special thanks for giving helpful advice while writing my thesis goes to Dr. Björn Grohmann.

I would like to express my deepest gratitude to my parents and my family and my friends for supporting me in all the years of my studies as well as listing to my thoughts and also while writing this thesis.

Abstract

Binary Attestation (BA) is a security mechanism which has existed for decades. The goal of BA is to guarantee that the correct software is loaded on a computer system. This is accomplished by applying concepts of Trusted Computing technology. Trusted Computing technologies are: Trusted Platform Module (TPM), measured/trusted boot, and the Linux Integrity Measurement Architecture (IMA), to ensure that a system has booted correctly and authentic software is running. Further, to provide evidence that the system's operational state is trustworthy, a log is generated which contains file names and hash sums of software components. The trustworthiness of the evidence is tied to a TPM. Later on, a remote party is able to verify the evidence if the operational state of the target system is trustworthy, in a process known as Remote Attestation (RA). Cyber-physical systems (such as power grids or nuclear power plants), and vehicular systems (such as railway- or automotive-transportation) are running multiple applications from various vendors. In the case of binary remote attestation, all log entries (such as file names and hash sums of software components) of all running software from all vendors are exposed to the remote party. Thus, the problem of binary remote attestation is that privacy is not preserved.

This thesis resolves the privacy deficiency by designing a privacy-preserving remote attestation approach. The core idea of the approach is to combine the Trusted Computing technology sector with the privacy-enhancing technology area. The approach preserves privacy by blinding all log entries. Hence, the approach reveals the associated (blinded) log entries to the verifier by applying the Schnorr Non-Interactive Zero-Knowledge (NIZK) proof over an elliptic curve (Schnorr Signature). This allows the attester to convince the verifier that the blinded entry is the actual running software on the attester's system without exposing the other entries.

To demonstrate feasibility and practicability of the approach, this thesis implemented a Proof-of-Concept (PoC). This work evaluated the performance and privacy of the PoC. The results show that privacy against the remote party is guaranteed while ensuring integrity and authenticity of the operational state of the attester's system (target system). However, the resource utilization increased during remote attestation while preserving privacy.

Kurzfassung

Binary Attestation (BA) ist ein Sicherheitsmechanismus, der schon lange zur Sicherstellung der Integrität der auszuführenden Software auf einem Computerssystem verwendet wird. Dies wird durch die Anwendung der Konzepte von Trusted Computing Technologie erreicht. Die Konzepte der Trusted Computing Technologie sind: Trusted Platform Module (TPM), measured/trusted boot und Linux Integrity Measurement Architecture (IMA). Damit die Vertrauenswürdigkeit des Betriebszustandes des Systems überprüft werden kann, wird eine Logdatei erstellt. Diese enthält den Dateinamen und die Hash-Summe der Softwarekomponenten, welche an das TPM gebunden wird. Zu einem späteren Zeitpunkt kann eine entfernte Partei (Verifier) die Evidenz verifizieren, um zu sehen, ob der Betriebszustand des Systems vertrauenswürdig ist. Dies wird als Remote Attestation (RA) bezeichnet.
Auf den Systemen im Bereich der Cyber-physische Systeme (z. B.: Energie Netze und Kernkraftwerke) und im Bereich der Verkehrsmittel (z. B.: Züge und Automobil Fahrzeuge) laufen mehrere Anwendungen von verschiedenen Herstellern. Bei bisherigen Ansätzen der BA, werden alle Einträge der Logdatei von allen laufenden Anwendungen aller Hersteller während der RA an die entfernte Partei preisgegeben. Daher gewährleistet die Binary Remote Attestation keine Privatheit.
Diese Masterarbeit adressiert die genannte Schwäche von der Binary Remote Attestation indem es eine Privacy-Preserving Remote Attestation Ansatz erstellt. Im Kern der Arbeit steht die Konzeptionierung eines Ansatzes, welches den Trusted Computing Sektor mit Privacy-Enhancing Technologie verknüpft. Der Ansatz bewahrt die Privatheit, indem alle Logeinträge verschleiert werden. Dabei werden dem Verifier nur die jeweilig zugehörigen Einträge offengelegt. Dadurch kann der Attester dem Verifier durch die Anwendung des Schnorr Non-Interactive Zero-Knowledge Proofs über eine elliptische Kurve (Schnorr-Signatur) beweisen, dass es sich bei den ausgeblendeten Einträgen um die tatsächlich laufende Anwendung auf dem System des Attesters handelt, ohne die anderen Einträge preiszugeben. Dieser Ansatz wird in dieser Arbeit als Proof-of-Concept umgesetzt. Die Leistung und die Privatheit des Proof-of-Concepts wurden in dieser Arbeit analysiert. Hierdruch wird gezeigt, dass die Privatheit gegenüber der entfernten Partei gewährleistet wird, während die Integrität und Authentizität des Betriebszustands des Systems vom Attester sichergestellt wird. Die Gewährleistung der Privatheit geht allerdings mit einem Anstieg der benötigten Ressourcen einher.

Contents

CHAPTER 1

Introduction

The first chapter of this thesis motivates the work, explains the problem statement and the aim of the work and finally outlines the following chapters.

1.1 Motivation & Problem Statement

The Internet of Things (IoT) as well as E-Vehicles and power grids are using a hardware security module called Trusted Platform Module (TPM) to enhance the security state of the devices or platforms. The TPM is capable of securely generating and storing keys. These private keys never leave the TPM, due to its resistance against attacks. One of the major purposes of the TPM is to check the software integrity of the executed components starting from the bootloader to the operating system, during the boot process of a platform by using the generated private keys.

Software integrity is an integral part of the system security domain. Remote Attestation (RA) is a process which enables a third party to assess the operational state of a platform. The operational state is defined as all executed software on a platform. The TPM is used to measure the executed software. It uses the Platform Configuration Registers (PCRs) and stores the measurements into logs, called Stored Measurement Log (SML). Furthermore, the TPM has the responsibility to provide the integrity of the SML. The executed software on a platform is measured using the TPM, which stores the measurements into the SML.

During remote attestation the target system, the attester, sends to the third party the SML and a signature from the TPM, constituting the operational state of a platform. The verifier checks the logs against a whitelist. The verifier receives the entire SML thus, disclosing information to the verifier about all running software on the attester since the last boot. In many use cases, privacy plays an important role. Thus, it may be desirable not to reveal the entire SML to the third-party verifier.

The concept of privacy-preserving remote attestation prevents a malicious verifier from knowing what other programs are running on a platform. The adversary is incapable of observing whether unpatched software is running on the platform. A naïve solution presents itself: instead of receiving the entire SML, the verifier only requests the SML entries, that are associated with the verifier. However, this solution breaks the ability of the verifier to verify the TPM anchored integrity and authenticity of the SML.

Solutions to this problem have been presented in the past based on property-based attestation from Sadeghi et al. [55] and Chen et al. [19]. However, these approaches have the following disadvantages. First, Sadeghi et al. [55] use a similar approach, which this work will present, but in their concept, a Trusted Third Party (TTP) is necessary to map the system state (binary configuration) to a property and the TTP needs to publish the properties. Secondly, Chen et al. [19] designed a protocol, which uses a signature and commitment scheme, but the protocol needs more computational operations as well as it relies on how many properties are mapped. Otherwise, the verifier can guess which property is mapped to configurations (system state) and compromise privacy. Nonetheless, this thesis intends to improve these approaches by applying Schnorr signatures, which reduce the computational overhead. Additionally, this new scheme will not require a trusted mapping/transformation service for properties. Hence, this thesis solves the privacy deficiency based on binary attestation. Furthermore, a proof-of-concept implementation will be developed and evaluated. Therefore, this work focuses on how to limit the exposure of the SML to a defined subset without losing the chain of trust.

1.2 Aim of the Work

The aim of the work is to develop a RA concept while preserving privacy. In order to develop a Privacy-Preserving Remote Attestation (PPRA) protocol, it is necessary not to reveal the entire SML. This work seeks to use zero-knowledge proofs for proving the integrity of the received subset, to ensure the chain of trust remains intact. Therefore, this thesis tries to answer the following research questions about developing a PPRA protocol:

- How can the TPM-based log integrity be preserved, while revealing only parts of the SML?
- Can the privacy-preserving property be realized with a zero-knowledge proof? How can a solution be designed?
- How could such an approach be used in a scenario with multiple verifiers in order to prove to an interested third party that all entries of the SML are verified?

Currently, RA approaches reveal everything about the system state of the platform.

First, it is necessary to define what kind of information should be sent to the verifier. In the currently existing approaches, the attester signs the aggregated hash of the SML and sends the signed accumulated hash with the entire SML to the verifier. As mentioned in the previous section the attester sends a subset of the SML, which breaks the integrity of the chain of trust. To remedy the chain of trust, we could additionally send hashes of the non-revealed programs. However, this allows an attack in which the adversary creates a reverse lookup table from the hashes to programs. Therefore, a malicious verifier is capable of observing what kind of software is running on the system. Our approach randomizes the hash, eliminating the attack. The idea is to "hide" the hash and send only the column of the SML containing the scrambled hash and the accumulated hash which is signed by the TPM. During the verification process, the verifier is re-aggregating the hashes and checks if the output is the same as the signed accumulated hash from the attester.

Secondly, the randomization of the hash is designed in a way, which allows us to prove, in zero-knowledge, that the randomized hash corresponds to a revealed program hash. In our case, we use non-interactive zero-knowledge proofs to implement the privacy-preserving RA scheme.

To conclude, we design a concept, which addresses the privacy deficiency of binary attestation.

1.3 Outline

This thesis is organized in coherent structure. Chapter 2 presents the prerequisites and fundamentals, which are used during this work. Further, it elaborates on the RA process and the non-interactive zero-knowledge proof. The fundamentals are explained to understand how these two research sectors are combined in this work. In Chapter 3, an overview is given of related work about property-based attestation, and it delimits between binary attestation and property-based attestation. Moreover, it presents similar methods to preserve privacy in the remote attestation process. Chapter 4 provides real-world use cases, where privacy is an integral part. Four use cases are presented in the chapter and explained how privacy is a necessity. Chapter 5 does a requirement analysis for establishing the PPRA approach. Afterward, Chapter 6 presents in detail the theoretical design approach of the PPRA protocol. The theory of the previous chapter is realized into a proof-of-concept, which dispenses the implementation and the algorithms in Chapter 7. Next, the proof-of-concept and the PPRA approach are evaluated in Chapter 8. Further, Chapter 8 discusses the achieved security and privacy aspects of the PPRA approach. In the end, Chapter 9 concludes the work of this thesis and gives an outlook for future work.

CHAPTER 2
Background

Remote Attestation (RA) protocols are used to provide authenticity and integrity of the running software on the attester side. The attester sends the attestation of the system state of the attester's platform to the verifier, where the verifier validates the attestation. Therefore, this chapter will elaborate the peculiarities to understand the approach of designing a Privacy-Preserving Remote Attestation (PPRA) protocol.

2.1 Trusted Platform Module

Trusted Platform Module (TPM) is a hardware chip, which is directly or indirectly integrated into a computer system. The concept and architecture of the TPM is specified by the Trusted Computing Group (TCG). The TCG states, to establish trust in a platform, it is necessary to identify the expected behavior of the platform's hardware and software [64]. Hence, the TPM provides functions for collecting and reporting these behaviors. Therefore, the TPM is used in a computer system to determine the expected behavior to establish trust.

Furthermore, the TPM provides cryptographic primitives, credential protection, secure storage and remote attestation [64]. In other words, the TPM is a small crypto engine. Additionally, the TPM is protected against physical attacks, so it is infeasible for an adversary to extract any secrets. In the case of this thesis, the TPM in version 2.0 is used. As mentioned, the TPM provides cryptographic primitives. Therefore, after manufacturing the TPM 2.0, it is shipped with unique cryptographic seeds. These cryptographic seeds are used to derive key pairs. The TPM 2.0 provide three major key hierarchies, each has a primary key or root key, which is derived from its own primary seed [41, 64]:

- **Endorsement Key:** The Endorsment Key (EK), which is derived from endorsement primary seed (EPS). The EK is controlled by the privacy administrator. The

EK is used to prove legitimacy to the user.

- **Platform Key:** The Platform Key (PK), which is derived from platform primary seed (PPS). The PK is managed by the platform firmware.
- **Storage Key:** The Storage Key (SK), which is derived from storage primary seed (SPS) and controlled by the platform owner. The SK is used to protect user data.

All the introduced keys provide a private and a public part, where the private part never leaves the TPM. Hence, the user can derive new keys under the hierarchy. For instance, a new key can be derived from the EK.

2.1.1 Attestation Key

The Attestation Key (AK) is an essential part during the remote attestation process. The TPM can accommodate multiple attestation keys. The major purpose of these keys are to sign the internal state of the TPM. To receive the internal state of the TPM a specific operation called *TPM Quote* needs to be executed. Moreover, the EK can be used for signing the internal state of the TPM as well, if the *TPM Quote* operation is explicitly specified in the *restricted* attribute of the EK. Furthermore, the key pair of the AK leaves the TPM, but the private part of the key pair is encrypted by the parent key (e. g. EK or SK).

2.1.2 Platform Configuration Registers

The Platform Configuration Register (PCR) is another integral part of the remote attestation process. The TPM provides PCRs which are part of its volatile secure storage facilities. The PCRs are used to generate a chain of trust, which will be explained in the next section. The task of the PCR is to generate folding hashes which are saved in the TPM. The TPM does not allow saving data directly, thus the TPM allows for an operation called *PCR extend*. *PCR extend* is an operation, which modifies the value of a specific PCR and anchors in the TPM to maintain the chain of trust. The *PCR extend* operation is defined as [64, 56, 72]:

$$extend(i, d) := H(PCR_i \| d) \tag{2.1}$$

Where i defines the PCR number or PCR index, H is a cryptographic hash function and d is the data which needs to be extended into a specific PCR value. PCR_i represents the current value of the PCR number i. After, applying the *PCR extend* operation the $PCR_{i'}$ contains the newly extended value. The $PCR_{i'}$ is computed as :

$$PCR_{i'} := extend(i, d) \tag{2.2}$$

Furthermore, the PCR banks in TPM 2.0 allow for each PCR number to have multiple values. These values are folding hashes applied with different hashing algorithms which are anchored in PCRs within the TPM.

In every boot sequence, the TPM sets all PCR values to zero which are the initial values, except for some special PCRs, the initial values are set to one. Moreover, the PCRs are useful for keeping the folding hash of log entries. During the remote attestation, the folding hash can be recomputed by verifying the integrity of the whole log.

2.2 Trusted Boot

The TCG defined the trusted boot process otherwise known as "Measured Boot", which takes the integrity measurement during the system boot sequence. This thesis uses the term *measurements* throughout the work. This term is referred to the computed hashes over software applications/binaries/software components.

The difference between secure boot and trusted boot is that secure boot measures components and verifies them, if the verification fails the boot sequence is aborted [41]. However, trusted boot only measures the components without verifying them in place. After measuring each software component, the trusted boot extends the PCRs inside the TPM. Further, it keeps a log of all measured components, the log is called *Boot Log*.

In addition, a chain of trust is an essential part of the whole trust establishment process defined by the TCG. The root of the chain is an immutable Core Root of Trust for Measurement (CRTM). The CRTM is the first entity in the trusted boot, which becomes active after the system is powered on. The CRTM measures itself and extends a PCR by storing the measurement into the TPM. Next, the next entity will be measured and the measurement will be stored, and the entity is executed. The measurement steps for maintaining the chain of trust are [56, 72]:

1. measure next entity in the chain,
2. store the measurement in the TPM by extending a PCR and
3. pass control to the next entity in the chain.

After, measuring all components in the trusted boot (see Figure 2.1), the chain of trust reaches the operating system. In Linux-based operating systems the Integrity Measurement Architecture (IMA) takes over with the same procedure of the trusted boot, where it continuously measures all executed software applications. Therefore, it brings the same conduct of the trusted boot in the operating system.

2.3 Integrity Measurement Architecture

The IMA is a part of the Linux kernel and is a trusted computing component [38, 56, 72]. After the trusted boot process reached the operating system, the IMA is invoked to continue the measurement of all software loaded into the memory. The IMA establishes a log file, the Stored Measurement Log (SML) and supports various log formats known

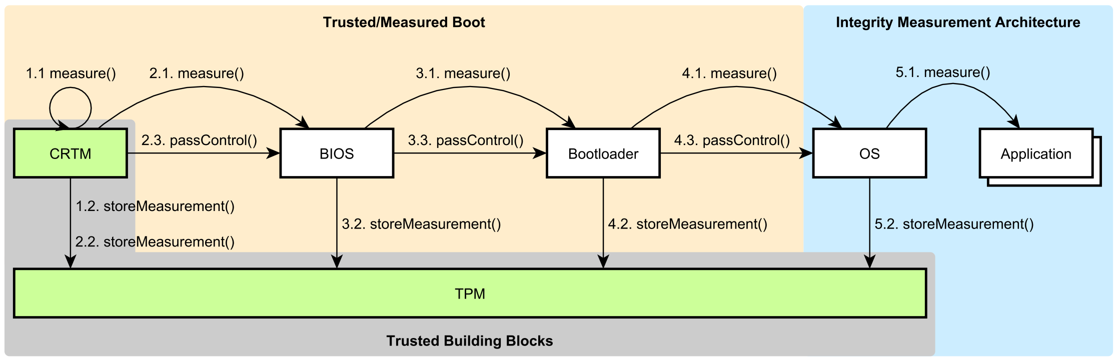

Figure 2.1: Chain of Trust [25].

as IMA templates. The measuring process takes place before the software component is loaded into memory. This process is atomic, which executes in the following three steps with no interruption [38, 56, 72]:

1. **Measurement of software applications:** The hash value from the software application file is computed and then the software application is loaded into memory. The hash algorithm varies due to the used IMA template.

2. **Insert entry into the SML:** After the measurement, the hash, and information will be added into the SML. Before, it adds the entry to the SML it checks if it already exists. If the entry exists, it checks if the file content (the measurement) changed, if not, it will not be added. The SML is used to ensure the integrity of an SML entry. The computed *template hash* is a part of the SML entry.

3. **Anchor entry into the TPM:** After measuring and adding the measurement into the SML a PCR inside the TPM will be extended with the *template hash*. The PCR extend operation is done by the TPM to anchor an *accumulative hash* over all SML entries. An example is given for PCR extend operation over all SML entries (n is the number of SML entries, e represents the *template hash* of an SML entry) [56, 72]:

$$\begin{aligned}
PCR_i^0 &:= extend(i, e_0) \\
PCR_i^1 &:= extend(i, e_1) \\
&\vdots \\
PCR_i^n &:= extend(i, e_n) \\
PCR_i^n &:= extend(extend(extend(extend(i, e_0), e_1), \ldots), e_n)
\end{aligned}$$

The last line shows how the folding hash is generated with the extend operation.

PCR Index	Template Hash	IMA Template	File Hash	File Path
10	c4456...331	ima	65727...12b	boot_aggregate
10	73c9b...eff	ima	153f3...c95	/lib64/ld-linux-x86-64.so.2
10	9b0ed...c09	ima	4ebaf...c9a	/lib/x86_64-linux-gnu/libc.so.6
10	fef56...71b	ima	64743...f69	/bin/dash
⋮	⋮	⋮	⋮	⋮
10	ee599...667	ima	fc66b...cef	/bin/mkdir
10	089c4...4bf	ima	2f43e...699	/bin/ln
10	78bd2...14c	ima	e46fd...b48	/bin/mount
⋮	⋮	⋮	⋮	⋮

Table 2.1: Traditional SML_{ima} [26].

Moreover, IMA provides an IMA policy, which defines the set of software and binaries/files that needs to be measured. This policy is configurable by hard-coding it into the Linux kernel or over an character device in the OS in */sys/kernel/security/ima/policy*.

2.4 Stored Measurement Log

The SML is generated by the IMA and has the format of a specified IMA template. The existing IMA templates are *ima, ima-ng, ima-sig, ima-modsig* and *ima-buf* [38]. The default IMA template is hard-coded into the Linux kernel, however, it can be configured. The Table 2.1 illustrates the log format of the IMA template *ima*. The column *PCR Index* defines the number of the PCR, in which the *template hash* of an entry has to be extended into the TPM. The IMA uses by default the PCR Index 10 for all entries. The first entry in the SML is different from the others, since it is the concatenation of all PCRs from the trusted boot process(0-7): $boot\_aggregate := H(PCR_i \| PCR_{i+1} \| \dots \| PCR_n)$.

Template Hash The *template hash* is the concatenation of an entry (column) *File Hash* with the (column) *File Path*. $H(\text{File Hash}\|\text{File Path})$, where H is the default hash algorithm used in the IMA. The IMA supports as default SHA-1 for generating the template hash. The *IMA template* column defines, which log format is used.

2.5 Reference Integrity Measurements

The Reference Integrity Measurements (RIMs) represent the intended operational platform configuration state. Moreover, RIMs serve as a whitelist for the verifier to compare it against the SML. The structure of the RIMs may be the same as the SML. Furthermore, RIMs are used during the measurement verification process to find the matching entry of the SML. If and only if all the entries of the SML are found in the RIMs, the verification result will be valid.

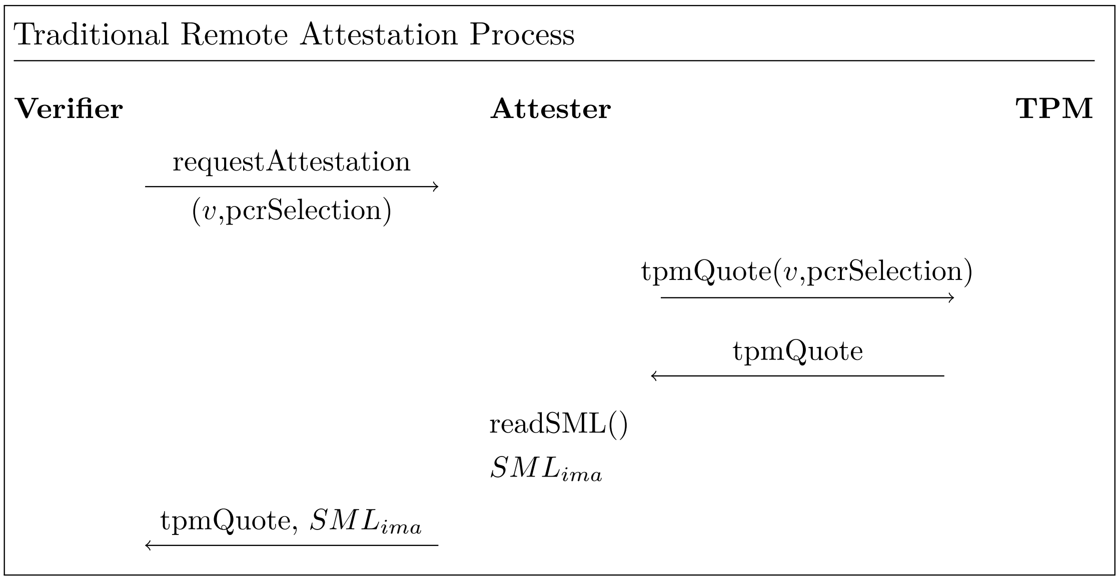

Figure 2.2: Remote Attestation process between verifier and attester.

2.6 Remote Attestation

RA is the process for reporting TPM-produced cryptographically signed attestation evidence about the attester's platform operational state [26]. This state will be transferred to the requested remote party, the verifier, which will verify the received state. The AK is a necessity for the RA. The TPM has full access to the AK, which is required to cryptographically sign the attester's platform operational state known as *TPM Quote*. The verifier has access to the public key of the AK in order to verify the signed data/state. Figure 2.2 embellishes the whole traditional RA process. Further, the v in the figure represents a nonce, which is used to prevent replay attacks and to provide freshness/recentness.

2.6.1 TPM Quote

A *TPM Quote* represents the current platform operational state. The *TPM Quote* contains the following data, which will be verified by the verifier [38, 56, 72]:

- The selected PCRs $(s_0, \ldots, s_n)$.
- External Data, which is normally a nonce provided by the verifier.
- A signature signed by the TPM's AK over the PCR selection and external data: $sig(H(\{extDat, PCR_{s0} \| PCR_{s1} \| \ldots \| PCR_{sn}\}))_{AK'_{priv}}$, where H is a cryptographic hash algorithm.

Additionally, the TPM 2.0 includes the information about the internal clock of the TPM. The attestation process is triggered remotely by the verifier. Therefore, the verifier sends a nonce, for protection against replay attacks, as external data and a selection of PCRs, which are then included in the *TPM Quote*.

2.6.2 Verification

The verification process of the operational state of the attester's platform is a crucial part in the RA process. Since it has multiple verification parts. The whole process of verifying the operational state is split into *integrity verification* and *measurement verification*.

Integrity Verification Integrity verification ensures authenticity and integrity of the received data. Further, integrity verification is a prerequisite for measurement verification. The integrity verification is split into the followings steps [26]:

1. Verification of the TPM signature by using the public key (AK_{pub}) of the AK.
2. Verifying whether the external data (nonce) matches with the previously sent nonce to the attester, in order to trigger a remote attestation.
3. The integrity of the SML is verified. The verification is divided into:
    3.1. Verifying the whole SML by recomputing the accumulative hash $PCR_i^{n}\prime$ over all entries in the SML (see previous Section 2.3 and Section 2.4). The accumulated value will be compared to the actual PCR value from the *TPM Quote*.
    3.2. Verification of each entry in the SML by recomputing the *template hash*. The recomputed values for each entry will be checked whether it matches the actual values from the received SML.

    Each of the sub integrity verification steps needs to return a positive value, otherwise, the whole integrity verification will fail.

Measurement Verification Measurement verification is a process which takes the received SML from the attester and checks against the RIMs. If the comparison yields a positive value, then the measurement verification is valid.

The operational state of an attester's platform is only valid, if and only if all subparts of the verification process yield positive values, otherwise the operational state is corrupted and the attester's platform is not trustworthy.

2.7 Abstract Algebra

The fundamental basis of public-key cryptography is based on the theory of abstract algebra and algebraic structures. Hence, this work recalls some basics of algebraic structures. First, the definition of the groups and fields will be elaborated.

2.7.1 Groups

Groups are algebraic structures. Let G be a non-empty set, which is associated with a binary operation $\circ$. G is mapped as $G \times G \rightarrow G$, which means that each element $a, b \in G$ results to an element $a \circ b$. Therefore, the pair of the set G and binary operation $\circ$ is known as a binary algebraic structure $(G, \circ)$. $(G, \circ)$ will be defined as a *group* if the following axioms hold for the binary operation [61, 54]:

- The binary operation $\circ$ takes two elements $a, b \in G$ from the group as input and returns as output an element $a \circ b \in G$. If the resulting element is in the group then the group is closed under the binary operation $\circ$.
- For the elements $a, b, c \in G$ the associativity should hold for the group, if $(a \circ b) \circ c = a \circ (b \circ c)$ applies.
- There exists an identity element $e \in G$, if $e \circ a = a \circ e = a$ for all $a \in G$ applies.
- For all elements in the group $a \in G$ exists an inverse element $a\prime \in G$, if $a \circ a\prime = a\prime \circ a = e$ holds. The element e is the identity element of the previous axiom.

If all these axioms hold, then the set G with a binary operation is a *group*. Further, a *group* is an *abelian group*, if commutativity axiom holds, which defines if all the elements in a group $a, b \in G$ applies to $a \circ b = b \circ a$ [61]. A group is a finite group if it contains a finite number of elements [61]. The group has an order, which defines the number of elements the group contains. If no order is defined the group has an infinite number of elements.

The cyclic group is an integral part of the design of the PPRA. A group is a cyclic group if a single element generates the group. Let $\cdot$ be a multiplicative binary operation and the group is defined as $(G, \cdot)$. With the binary operation $\cdot$ exponentiation can be defined as a repeated operator of $\cdot$. For instance, g is an element of G and $g^3 = g \cdot g \cdot g$ [61]. Moreover, the identity element is defined as $g^0 = e$ and the inverse element is $g^{-n} = (g\prime)^n$. Therefore, if all the elements in G can be generated by g^r, where $r \in \mathbb{Z}$ and g is a fixed element in G, then the group is cyclic [61]. g becomes the generator of the group $(G, \cdot)$, since g^r generates all the elements in the group.

2.7.2 Fields

Fields are defined with two binary operations such as $(F, +, \cdot)$, where F is the field. However, in cryptography, the field is slightly different defined such as F_p, where p is a prime number. F_p^* means a finite field without the element 0. Further, this definition includes integer modulo p [54]. Moreover, the field is a finite field, which has finite elements.

2.8 Elliptic Curve Cryptography

The Elliptic Curve Cryptrography (ECC) was introduced by N. Koblitz and S.V.Miller in the 19th century [39]. The ECC is more preferred than the RSA cryptosystem due to the smaller key size, and it provides the same security. The computation in the cryptosystem plays as well as a major role. The ECC is more efficient than the RSA cryptosystem. Therefore, ECC is preferred in the embedded system domain.

An elliptic curve is characterized by an equation with two variables (x, y) and coefficients. Further, the two variables are the coordinates, which satisfy the equation. For this work an elliptic curve $M_{A,B}$ over finite field F_p in Montgomery form is used, which is defined by the following equation [54]:

$$M_{A,B} : By^2 = x^3 + Ax^2 + x \tag{2.3}$$

The parameters A, B in Equation (2.3) are constants for defining specific curves (see Chapter 7) [54]. Moreover, to satisfy the equation, following conditions for parameters A, B should be met [54]:

- $A, B \epsilon F_p$
- $B(A^2 - 4) \neq 0 \implies B \neq 0, A \neq -2, A \neq 2$

Further the elliptic curve $M_{A,B}(F_p)$ defines the set of points, which satisfy the Equation (2.3).

Elliptic curves are defined over a finite field, which has a group structure of an abelian group [47]. The group structure of the elliptic curve is used to build cryptosystems [39]. Besides, two major operations on the elliptic curve will be used in the cryptosystem and these are point addition and point multiplication.

2.8.1 Point Addition

As above stated, let $M_{A,B}$ be an elliptic curve over F_p. Let P and Q be points on $M_{A,B}$. A point on the elliptic curve is defined as $P(x, y)$, where x and y are coordinates. Next, this work defines the point addition, negation, and doubling. The negation of point P is defined as $-P(x, -y)$. If two points are selected such as P and Q, then the point addition is $R(x_3, y_3) = P(x_1, y_1) + Q(x_2, y_2)$ where [48, 20]:

$$\begin{aligned} x_3 &= (B\lambda^2 - A - x_1 - x_2) \\ y_3 &= (2x_1 + x_2 + A)\lambda - B\lambda^3 - y_1 = \lambda(x_1 - x_3) - y_1 \end{aligned} \tag{2.4}$$

with

$$\lambda = \begin{cases} (y_2 - y_1)/(x_2 - x_1) & \text{if } P \neq Q \text{ or } -Q, \\ (3x_1^2 + 2Ax_1 + 1)/(2By_1) & \text{if } P = Q; \end{cases}$$

The case defines which condition is taken for λ. If point P is equal to point Q on the curve $M_{A,B}$, then it is point doubling $P + P = 2P(x, y)$. Therefore, the point doubling needs a different case in the point addition.

2.8.2 Point Multiplication

The point multiplication is defined as [39, 54]:

$$Q = k \cdot P \tag{2.5}$$

P is a point on the elliptic curve and k is an integer value. The point multiplication is not the same as the integer multiplication. Because the scalar multiplication takes P and add itself k times [20]:

$$Q = k \cdot P = \underbrace{P + P + ... + P}_{\text{k times}} \tag{2.6}$$

Q is a point which is on the elliptic curve. The interesting part of this operation is the inverse operation, which retrieves the integer k, when the points P and Q are given. To retrieve k is a "hard problem" known as the **Elliptic Curve Discrete Logarithm Problem (ECDLP)** [39]. To solve the ECDL problem for a properly chosen elliptic curve group, with an algorithm in sub-exponential time is not possible, since it is assumed such an algorithm does not exist (it is not known/not proven that such kind of algorithms exists). The ECDLP is the fundamental basement for using the Elliptic Curve (EC) as a public-key cryptosystem.
Furthermore, both group operations (point addition and point multiplication) are used for designing the concept of privacy-preserving remote attestation, in Chapter 6.

2.9 Zero-Knowledge Proof

The zero-knowledge proof system was introduced by Goldwasser, Micali, and Rackoff [11]. Zero-knowledge proofs are applied for constructing cryptographic secure protocols [69]. Furthermore, the method of zero-knowledge proofs are practiced in the domain of cryptocurrencies such as Ethereum and Zerocash, to provide anonymity in transactions [1, 68]. The zero-knowledge proof is an interactive proof system with two parties (prover, verifier). The prover wants to prove an information without revealing the information to the verifier. In other words, the prover sends the verifier a proof of knowledge, which proves to the verifier that the prover has the knowledge of the verifier's seeking information without revealing it. In this case, this work is focused on non-interactive zero-knowledge proofs.

Non-Interactive Zero-Knowledge Proof

The prover sends only one message to the verifier, which is defined as a non-interactive zero-knowledge proof system. Meanwhile, the Non-Interactive Zero-Knowledge (NIZK) and Zero-Knowledge (ZK) proof need to hold the following properties to be accepted as a proof system [69, 11]:

- **Completeness:** The prover is capable of convincing the verifier that the statement is correct.
- **Soundness:** The prover is capable of convincing the verifier that the statement is incorrect.
- **Zero-Knowledge:** The prover does not reveal any extra information, except the correctness of the statement.

An interactive ZK proof system can be transformed into a NIZK proof system by applying the Fiat Shamir technique. The Fiat Shamir technique is used to transform the three-phase Schnorr identification scheme to a non-interactive variant [33]. This work builds on top of the Schnorr non-interactive zero-knowledge proof. Nevertheless, this work will explain the applied cryptography of the Schnorr non-interactive zero-knowledge proof over an elliptic curve.

Schnorr NIZK proof The Schnorr NIZK proof is used as a sub-protocol for the design of the privacy-preserving protocol. The Fiat Shamir technique is using a random oracle in the theory of cryptography. In the case of the thesis and applied cryptography the random oracle is replaced by a cryptographic hash function. Hash functions are deterministic. Further, it is not possible to draw any relation from the output of a cryptographic hash function, since it appears to be random. Now this work explains the Schnorr NIZK proof over an elliptic curve. Let $M_{A,B}(F_p)$ be an elliptic curve over a finite field F_p (see Section 2.7 and Section 2.8). The NIZK computation steps are based on the RFC standard 8235 [33].

The first phase is the setup phase, the prover generates the private key r where r is chosen uniformly random from prime order group L of the elliptic curve $M_{A,B}(F_p)$. Next, the prover generates the public key $R = g^r$, where g is the generator of the subgroup of the curve $M_{A,B}(F_p)$ with the prime group L (note: g^r is another notation for the point multiplication $r \cdot g$).

After generating the private and public key, the public key will be published to the verifier. Next, the prover chooses uniformly random v from the range between 0 and $L - 1$. The prover computes $V = g^v$. The next computation step is to compute the challenge c by applying the Fiat Shamir technique : $c = H(g \| V \| R)$, where H is a cryptographic hash function. Further, the prover computes $s = v - r \cdot c \mod L$. The prover sends the pair (c, s) to the verifier. The pair (c, s) is necessary for the verifier to validate the proof of knowledge.

The verifier verifies if the public key R is a valid point on the elliptic curve. Afterward, the verifier validates if the proof of knowledge is valid by computing: $V\prime = g^s \cdot R^c$. The

Prover **Verifier**

$r \leftarrow\$ \{0, L-1\}$

$R \leftarrow g^r$

$v \leftarrow\$ \{0, L-1\}$

$V \leftarrow g^v$

$c \leftarrow H(g \| V \| R)$

$s \leftarrow v - r \cdot c \quad \mod L$

$$\xrightarrow{(c, s), R}$$

$V' \leftarrow g^s \cdot R^c$

$= g^{v-r\cdot c} \cdot R^c$

$c' \leftarrow H(g \| V\prime \| R)$

if $c == c'$

return 1

Figure 2.3: Schnorr non-interactive zero-knowledge proof over elliptic curve [33, 11].

computing step of $V\prime$ is a point addition since g^s and R^c are points on the elliptic curve. The verifier computes $c\prime = H(g\|V\prime\|R)$. Last the validity check is, if and only if $c\prime == c$, then the proof of knowledge is valid.

Figure 2.3 illustrates the computation steps described above in mathematical notation (note: v, r, c are scalars therefore the scalar arithmetic is applied. For g^s, R^c are points on the elliptic curve. Therefore, point operations are applied).

CHAPTER 3

Related Work

Research on remote attestation has existed for decades, this chapter references some related work. Moreover, the chapter introduces another technology branch of remote attestation, the property-based attestation. Further, it delineates the delimitation of the research between property-based attestation and binary attestation. In addition, this chapter provides different related approaches to achieve privacy.

3.1 Property-based Attestation

Sadeghi et al. [55] address the privacy deficiency of binary attestation and design a new attestation scheme called Property-based Attestation (PBA). The idea of PBA is to determine in the attestation process if the configuration of a platform or the application has a desired property [18]. A property describes the behavior of a certain configuration or application. A configuration or a set of configurations is mapped to a property. The property will be revealed by the attester, but the configurations or applications will not. Further, Sadeghi et al. [55] suggest a PBA protocol with a similar idea as the one pursued in this thesis. They construct a PBA protocol to prove membership, where the attester proves that the blinded configuration is in the set of configurations (the sets are platform configurations related to a desired property), which is published by a trusted third party, and proves that the Trusted Platform Module (TPM)'s attestation signature is valid. The blinding strategy is an encryption operation on a platform configuration. In other words, the whole state (or platform configurations) of the system is one of the published states, which fulfills the property, but the verifier does not know which of them, while the attester only proves that the valid state is among the published set.

This thesis uses a blinding strategy as well, but it does not use an encryption operation. In this work, the verifier defines a set of binary configurations for which it is responsible and informs the attester. Next, the TPM will attest the blinded configurations and sign them. Further, the verifier aggregates the blinded values and validates them. Afterwards, the attester reveals only the agreed configurations or applications to the verifier and proves that the revealed configurations are among the blinded set. The thesis resolves the privacy deficiency of binary attestation without mapping the configurations into properties.

Chen et al. [19] extend the protocol by Sadeghi et al.[55] with a detailed design in theory. They apply ring signatures and commitments. The computations of the ring signature are not as efficient as the one used in this work. The reason is that the Schnorr signatures use fewer computation steps to generate a signature and do not depend on a commitment scheme. Furthermore, if the size of the configuration set is too small, then the verifier can guess the configuration and thereby compromise privacy [19, 70]. Next, the verifier can run the protocol multiple times with a different set of configurations to find an intersection, which leads to compromising privacy as well [19, 70]. In the case of this thesis once the verifier defines the responsibility of the configuration it cannot be changed.

Sadeghi et al. and Chen et al. only suggested designs on how such an approach could be defined, but they did not implement a proof-of-concept. This thesis will illustrate a practical approach by implementing a privacy-preserving attestation mechanism based on binary attestation.

3.2 Privacy-Preserving Methods

The following works explain how to preserve privacy without applying zero-knowledge proofs and signature schemes. Luo et al. [42] discuss a concept about preserving privacy and reducing attestation overhead by implementing a partial attestation scheme. This approach is applied to virtualized platforms. In the paper the authors state that instead of sending every information to the verifier, they will enable the verifier only to attest to specific requirements of the remote platform. The privacy is preserved by using the *seal* function of the TPM, by sealing secret information which identifies the chain of trust. The verifier has to check only if the secret is accessible instead of receiving the whole configuration of the chain of trust.

Zhang et al. [71] focus on the existing problem of binary attestation, which is privacy. Therefore, the authors introduced an extended hash algorithm based on Merkle Hash Tree. This concept should provide the privacy-preserving property to the remote attestation scheme. The integrity of the program is preserved in the nodes of the Merkle Hash Tree. The verifier has to obtain the encrypted hash value while obtaining the node from Merkle Hash Tree. The privacy of the approach is given due to the nodes of the Merkle Hash Tree, which resembles a file of a program. At last, the encryption is introduced as xor and cycleshift(bit shift) operations.

Next, Luo et al. [43] designed an architecture for privacy-preserving integrity measurement for containers. For each container, they measure all the processes and generate a secret for each measurement ($PCR.secret$). These secret values are saved into individual event logs for each container. Next, each entry of the event log of a container is “XOR’ed” with the corresponding *PCR.secret*. This is done for each entry. Afterwards, all the “XOR’ed” values will be folded to a folding hash. The folded hash is extended into a specific Platform Configuration Register (PCR) of the TPM. This process is done for each container. When the verifier triggers the remote attestation, it receives the *PCR.secrets* for the affiliated container and the accumulated hash of a specific PCR bank signed by the TPM. The verifier applies the same procedure as described above and compares the actual value with the received one. The state of the other containers and the host system is in the accumulated hash signed by the TPM. However, the verifier can only check the value of the affiliated container. By applying the *XOR* operation, they preserve privacy.

The concepts of these papers illustrate one way for solving the privacy problems of binary remote attestation for virtualized platforms. As the majority of these papers rely on the *XOR* operation, we will be using in this work another approach to focus on privacy, based on non-interactive zero-knowledge proofs. Further our approach tries to focus to be applicable in the cyber-physical and automotive domain.

In conclusion, this chapter presents a related work, where it uses a non-interactive zero-knowledge proof. Hamadeh et al. [31] introduce an implementation of privacy-preserving data provenance based on Physical Unclonable Function (PUF) and non-interactive zero-knowledge proof. This work establishes a protocol, which provides more security in the Internet of Things (IoT) ecosystem. The PUF in case of this work has similar tasks as the TPM and the non-interactive zero-knowledge proof is used in the protocol for the authentication stage, where the IoT device proves to the server (verifier) without revealing any identity of the IoT device, which ensures privacy. After, the successful verification the next stage is to establish an encrypted channel. The last stage of the protocol is to transmit the data and verification of the IoT data provenance. The work demonstrated the technique of non-interactive zero-knowledge proof provides benefits to ensure privacy.

CHAPTER 4

Use Cases

Remote Attestation (RA) is a resourceful security procedure, which can be applied in different areas. Therefore, this chapter introduces four use cases. Each use case describes an overview on how remote attestation can be applied and why privacy property is necessary.

Private User Domain The private user domain addresses the domain of devices such as smartphones and laptops. These devices contain application information which the user uses daily. The manufacturer of one of the installed applications on the smartphone wants to attest the integrity and authenticity. The traditional RA would reveal all the running applications on the device to the manufacturer. However, if the Privacy-Preserving Remote Attestation (PPRA) is applied, the manufacturer only knows the affiliated application and the other applications are not revealed. This provides the integrity and authenticity of the operational state of the device.

The banking sector can be taken as a concrete example. The bank wants to attest the banking application on the customer's smartphone. The customer does not wish to expose the other applications (which could contain other banking apps) to the bank, except the one necessary banking app. Hence, the privacy of the customer will be contained. Consequently, this thesis introduces a design concept and proof-of-concept implementation which addresses the privacy problem.

Health Care The integrity and authenticity need to be ensured of the Electronic Medical Devices (EMD) and the software applications in a hospital [62]. The hospital has various departments for different patient cases. Hence, each department uses various software applications from different software distributors. For instance, the software distributor of the cardiology department wants to attest integrity and authenticity of the software and the operational state of the hospital. However, the software distributor of the cardiology does not need to know about the other running applications in the other departments. Therefore, a design for PPRA is necessary, which divulges only the affiliated software of the software distributor.

Cyber Physical Systems Power-grid has multiple resource facilities, where the electricity is generated by windmills, hydro-power plants, nuclear power plants or solar grids. Each of the plants have various software applications running on the system. The facilities are checked by different authorities to see if plants maintain the standards and regulations. A concrete example would be the International Atomic Energy Agency (IAEA), which conducts safeguard inspections at nuclear power plants [34]. One of their standards is to collect the recording of the safeguard surveillance videos [34]. The software for the surveillance system is a part of the nuclear power plant. If the IAEA wants to attest the reactor facility, it does not need to learn about other running software at the nuclear power plant. Therefore, the information of the surveillance system will be revealed, while the rest of the system is blinded, and the integrity and authenticity are assured.

Automotive In the automotive context the privacy plays an important role since a car has multiple manufacturers for different components, such as breaks, sensors, etc.. Therefore, the manufacturer for the breaks wants to attest the car. In the case of the traditional RA all the other running software applications from other manufacturers are visible. The manufacturer for the breaks will know what kind of firmware/software is used for the tire sensor or what kind of applications are installed on the onboard computer. To prevent this knowledge a design for the PPRA is introduced in this work.

In conclusion, the presented four use cases underline the importance of the PPRA. In the following chapter requirements analysis is done to define requirements for the privacy-preserving remote attestation scheme.

CHAPTER 5

Requirements Analysis

Requirements analysis defines requirements for designing the Privacy-Preserving Remote Attestation (PPRA) approach. Further, this chapter lists the functional, security and privacy requirements. These requirements were derived based on the use cases of the previous chapter and the threat model (see Section 6.1).

5.1 Functional Requirements

Functional requirements are the fundamentals to design the PPRA protocol. The requirements are derived from the previously explained use cases. The design of the PPRA approach has multiple components and process stages to provide security as well as privacy. Therefore, requirements are necessary to realize the concept.

The first step to realize the approach is to use or simulate the operations of the Integrity Measurement Architecture (IMA) (Linux IMA), which computes the measurements (*template hash*) of applications/software components before they are loaded into the memory (F_1). Next, these measurements need to be blinded to promise privacy of the attester's system (F_2). In the following, the blinded measurements need to be persisted in a log file (i. e. Stored Measurement Log (SML)) (F_3). The logging of the blinded measurements keeps trace of all loaded software components into the memory of the attester's system.

Moreover, to anchor the blinded measurements into the Trusted Platform Module (TPM) and to retrieve the operational state of the attester's system, a communication interface has to be established. This interface provides access to the functionalities/operations of the TPM (F_4).

The remote party wants to attest the operational state of the target system. Therefore, it is necessary to provide or implement a component which provides the essential functionalities to trigger a remote attestation and to verify the response (F_5). Due to the remote

ID	Functional Requirements
F_1	Simulation/Usage of the integrity measurement architecture
F_2	Blinding the measurements
F_3	Logging the measurements of the software components/applications
F_4	Communication interface for utilizing the functionalities of the TPM
F_5	Provide functions for establishing a remote attestation
F_6	Standardize communication structure between attester and verifier
F_7	One communication round between verifier and attester
F_8	The attester reveals only the measurement entries associated to the verifier
F_9	The verifier validates the trustworthiness of the target system (attester)
F_{10}	Resource-restricted devices needs efficient operations
$F_{10.1}$	Applying cryptographic operation on various measurements, which utilizes the same amount of resources
$F_{10.2}$	Suitable resource consumption should be met to preserve privacy

Table 5.1: Lists all functional requirements.

attestation, a communication between the attester and the verifier is established. Hence, it is necessary to have a standardized communication structure, which the verifier and the attester understand (F_6).

The traditional Remote Attestation (RA) has only one round of the attestation request-response scheme. In other words, the verifier sends an attestation request and the attester in return sends an attestation response. Hence, the whole process is stateless, due to the one-round request-response scheme. Therefore, the PPRA must provide one communication round (F_7).

Furthermore, to preserve privacy during the remote attestation, the attester discloses only the associated information to the verifier (F_8). However, these information needs to be verified by the verifier. Whether the verification results in a positive outcome, then the trustworthiness of the attester's system is given (F_9).

The four use cases are introduced for different domains and each domain has various resource-restricted devices. Therefore, it is necessary to observe and analyze the performance of the PPRA approach (F_{10}). Besides, various measurements are blinded by applying cryptographic operation. However, the outcome of the cryptographic operation for different measurements should utilize the same amount of resources ($F_{10.1}$). The traditional RA has less resource consumption. Nevertheless, by achieving privacy, a suitable trade-off between privacy and resource consumption should be met ($F_{10.2}$).

5.2 Security Requirements

This thesis introduces a privacy-preserving RA approach. Therefore, the security requirements need to be fulfilled to be resistant against adversarial attacks. The integrity of the

ID	Security Requirements
S_1	Integrity of the operational state of the target system
S_2	Authenticity of the operational state of the target system
S_3	Communication between attester and verifier has to be encrypted
S_4	Mutual authentication between attester and verifier
S_5	Trustworthiness of the attestation information
S_6	Selection of a secure and efficient elliptic curve

Table 5.2: Lists all security requirements.

operational state of the target system has to be assured. Therefore, the measurements need to be anchored into secure volatile storage of a TPM (S_1). Further, the authenticity is guaranteed if the trusted platform module or trusted hardware security module signs the operational state (S_2).

During the remote attestation, a securely encrypted communication between the attester and the verifier has to be established to prevent *passive man-in-the-middle* attack (S_3). Nevertheless, an encrypted channel prevents the adversary from listing. However, the verifier needs to authenticate to the attester and the attester to the verifier. Hence, mutual authentication has to be established to guarantee the identity of each party (S_4).

Besides, the target system provides trustworthiness, while providing attestation information to the verifier. The verifier validates the attestation information if and only if all the attestation information is valid, then the target system is trustworthy (S_5).

To preserve privacy, it is necessary to select a suitable elliptic curve for the PPRA protocol. Thus, the selection of an elliptic curve guarantees that it does not leak any information to an adversary. Further, it needs to be efficient and provides security properties, which are advantageous for integrating it into the PPRA approach (S_6).

5.3 Privacy Requirements

This section continues with privacy requirements, which need to be fulfilled to preserve privacy. First, a suitable privacy-enhancing technology must be selected. Further, the privacy-enhancing technology needs to be adapted for designing a PPRA approach (P_1).

ID	Privacy Requirements
P_1	Applying a tailored privacy-enhancing technology
P_2	*"Constrained disclosure"* [30]
P_3	Verifiers do not communicate with each other

Table 5.3: Lists all privacy requirements.

During the remote attestation, the attester only discloses information, which is affiliated to the verifier. Nonetheless, the attester does not reveal any further configurations/running programs, while providing the integrity and authenticity of its system's operational state (P_2).

Multiple verifiers (e. g. vendors) can request the operational state of the same target system. Therefore, the verifiers do not communicate with each other, nor do they know each other (P_3). Otherwise, if the verifiers communicate with each other, privacy cannot be preserved. If the adversary compromises more than one verifier, it has the power to unite the constrained disclosures, which expose the privacy of the attester's system.

CHAPTER 6

Privacy-Preserving Remote Attestation

This chapter explains, first the threat model of the traditional remote attestation. Secondly, it gives a broad overview of the privacy-preserving remote attestation concept, which resolves one of the open challenges of binary remote attestation. Further, it elaborates in-depth, how to limit exposure of the Stored Measurement Log (SML) to a defined subset while preserving the ability of a remote party to verify integrity and authenticity of the SML (the operational state of the target platform). The original idea for the privacy-preserving remote attestation stems from an unpublished work [26] by M. Eckel and B. Grohmann. They provided the main idea and this thesis extends this theoretical foundation on how to integrate it in the measurement process, remote attestation process, and designs a threat model.

6.1 Threat Model

In Chapter 1 this work identified the problem with the traditional Remote Attestation (RA) scheme. The existing method of binary attestation provides the verifier the entire entries of the SML from the target platform (attester). Therefore, the verifier knows all software running on the attester's system.

The first attack vector is, an adversary listens to the communication channel between attester and verifier. Hence, the adversary is capable of recording the transferred SML. This attack is identified as the *passive man-in-the-middle* attack. However, this attack can be prevented by using an encrypted channel between attester and verifier. Nonetheless, the encrypted connection does not prevent further attacks. In the following attack vector a securely encrypted channel is used.

Moreover, the next attack vector is that the verifier is compromised by the adversary. Therefore, the adversary has the power to execute a remote attestation process and to gain the knowledge of all software running on the target system as well as the content of the SML. Even though, if only the *template hashes* are sent to the verifier, privacy is still not guaranteed (see definition of *template hashes* in Section 2.3). Hence, the adversary creates a reverse lookup table to match up the *template hash* to the corresponding binary/software component. Due to the reverse lookup table, the adversary has the advantage of knowing if the attester runs versions of software with unpatched vulnerabilities. Consequently, the adversary can deploy tailored attacks against the attester's platform and exploiting these vulnerabilities.

Another scenario is if the vendor who runs software on a target platform does not want to reveal their software components to another verifier. Thus, the verifier in the case of the traditional RA would gain the knowledge of all running programs. Here, the verifier can be an adversary to take advantage of knowing all the software running on the target platform (attester).

Therefore, this thesis designs a concept for a privacy-preserving remote attestation protocol by applying the method of non-interactive zero-knowledge proof (Schnorr signature). This concept addresses the privacy deficiency of the traditional remote attestation scheme.

6.2 General Idea

This section presents the Privacy-Preserving Remote Attestation (PPRA) concept against the previous explained threat model. Here this work gives a general context about the PPRA approach. Figure 6.1 presents the fundamental overview of the PPRA concept. The PPRA approach docks after the trusted boot sequence (see Section 2.2). From here on the traditional Integrity Measurement Architecture (IMA) measurement process will be adapted (see Section 2.3). The task of the IMA is to measure each application/software component ($APP_0, \ldots, APP_N$, see Figure 6.1) which will be loaded from the operating system into memory. Therefore, the first mechanism is to compute measurements of each application (step 1 in Figure 6.1). As already mentioned in Chapter 2, the term *measurement* is referred to the computed hashes over software applications/binaries/software components. After computing the measurements, the measurements need to be blinded by applying a privacy-enhancing technology (in case of this work non-interactive zero-knowledge is used as privacy-enhancing technology) (step 2 in Figure 6.1).The component *blind measurement* in Figure 6.1 adapts the standardized Linux IMA (see Section 2.3). Hence, this component is necessary to guarantee privacy. Moreover, the output (blinded measurement) of the component does not reveal any information about the computed measurement (step 1). The computed blinded measurement will be saved into a log file (privacy-preserving-SML). The log does not only contain the blinded measurement as well as it stores the associated measurement (i. e. file hash) and extra data for the blinded measurement (step 3 in Figure 6.1). The

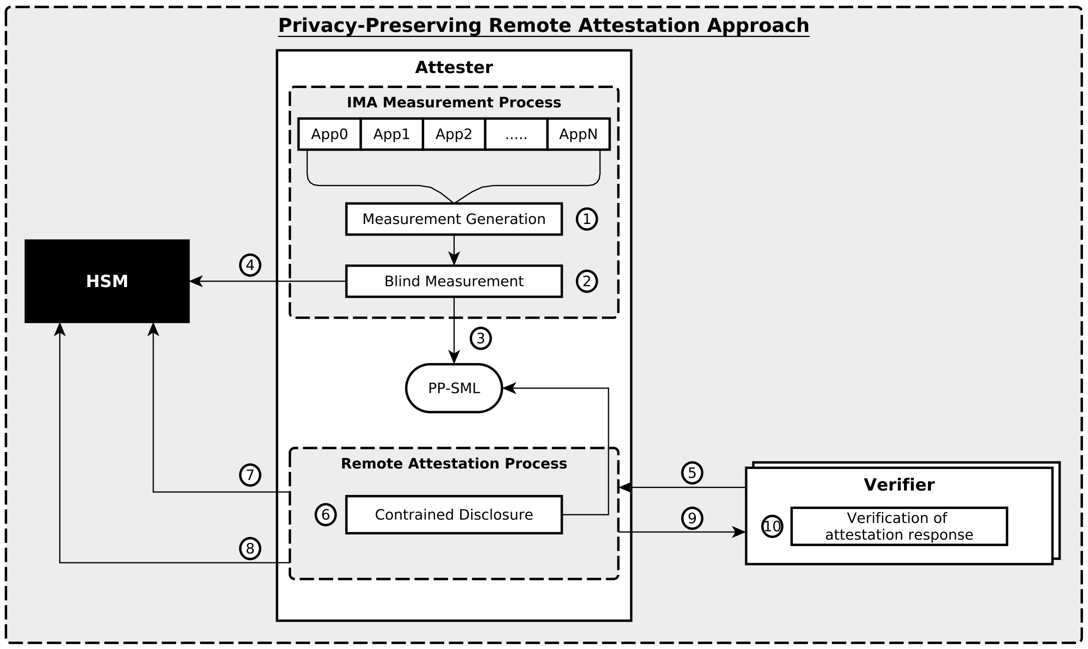

Figure 6.1: This graphic illustrates a general overview of the PPRA concept.

next process step is to anchor the blinded measurement into a trusted Hardware Security Module (HSM) to achieve the chain of trust and to preserve the integrity, and authenticity of the operational state of the attester's system (step 4 in Figure 6.1). Therefore, the HSM needs to provide operations to fulfill the previous mentioned requirements and an interface to trigger the operations. In Figure 6.1 the steps 1-4 demonstrated the *measurement process* of the PPRA approach.

After the *measurement process* is finished, the approach initiates the second process, the *remote attestation process*. This process is triggered if the verifier requests for the operational state of the target system (attester) (step 5 in Figure 6.1). Next, the attester checks the constraints against the verifier's request information and discloses only the associated subset of the entries in the PP-SML (step 6 Figure 6.1). Further, the attester requests from the trusted HSM to provide a quote, which represents integrity and authenticity of the operational state of its system (step 7 in Figure 6.1). The next process step is the HSM returns a quote of the systems operational state (step 8 in Figure 6.1). The attester sends the quote and the associated entries of PP-SML and all blinded measurements to verifier (step 9 in Figure 6.1). Finally, the verifier validates the received data. Whether the validation yields to a positive value without revealing all running software on the attester's system, then the attester's system is trustworthy while preserving privacy. During the verification the verifier applies operations of the privacy-enhancing technology to validate the associated blinded measurement, while validating the whole integrity of the operational state of the target system.

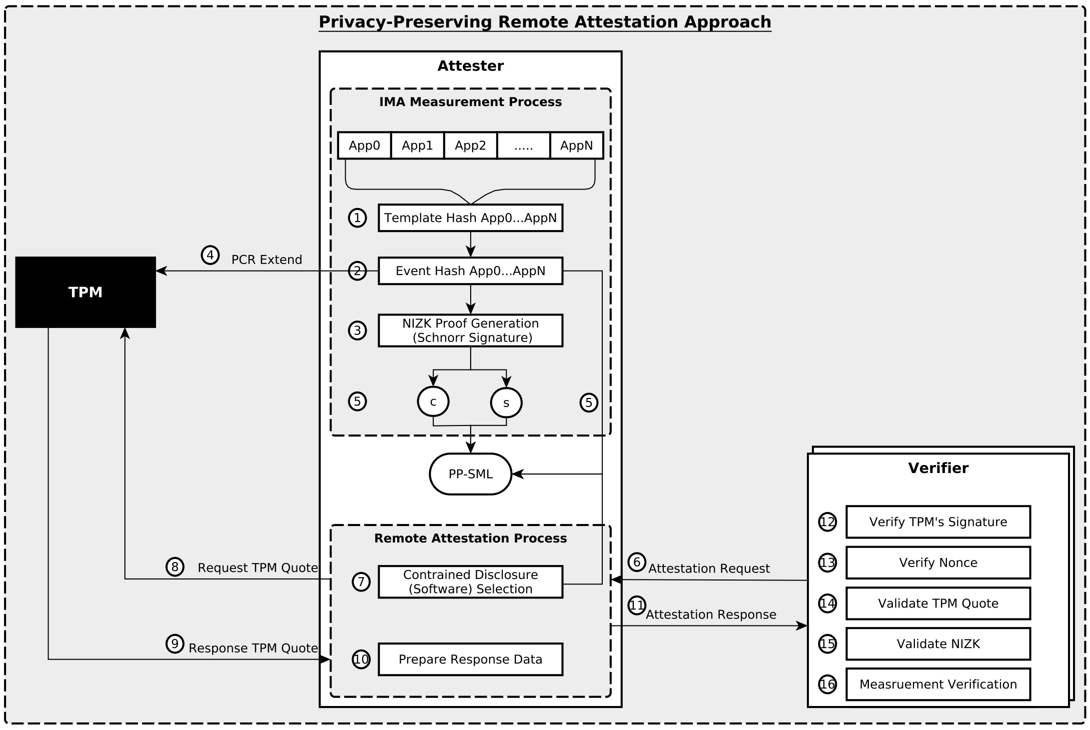

Figure 6.2: This graphic illustrates a detailed overview of the PPRA concept.

Later on in the upcoming sections, this work will explain in-depth each step and components of the PPRA approach (see Figure 6.2). In the upcoming sections the trusted HSM in Figure 6.1 will be replaced by a Trusted Platform Module (TPM), which provides the operations to realize the PPRA concept.

This chapter continues by describing the two adapted processes, *measurement process* and *remote attestation process* in detail. Further, it shows on how to maintain privacy, integrity and authenticity of the operational state of the attester's system.

6.3 Measurement Process

The PPRA approach initiates with the measurement process, where the chain of trust starts with an immutable Core Root of Trust for Measurement (CRTM). The CRTM measures itself and stores the measurement in the TPM by extending the Platform Configuration Register (PCR). The next steps are the same as described in Chapter 2, where it explains how trusted boot works. After the trusted boot procedure, the IMA measures every software loaded into memory of the Linux operating system and stores the measurement into a log file called SML. The privacy-preserving approach adapts the measurement process at the IMA measurement. The measurement step is the same as described above (each software will be measured and a *template hash* will be generated (step 1 in Figure 6.2)). However, instead of extending the PCR with the *template hash* (see Chapter 2), an *event hash* will be created (step 2 in Figure 6.2) and the PCR will be extended with the *event hash* (step 4 in Figure 6.2).

6.3.1 Event Hash

The *template hash* reveals more information about the SML than desired. Therefore, a non-deterministic hash is established called *event hash*. The *event hash* is the blinded *template hash* by applying the non-interactive zero-knowledge signing process, otherwise known as Schnorr signature. First, the *template hash* function is defined as $h_T : \mathcal{X} \rightarrow \mathcal{Y}$ [26]. The $\mathcal{X}$ defines the set of the *template hashes* and transforms them into the range $\mathcal{Y}$ with the function h_T. In order to generate the *event hash* a generator g of a (cyclic) group G needs to be selected (the generator g is in the case of this work the base point of an appropriate elliptic curve over a finite field). Second, the function for the *event hash* is defined as $h_{eventhash} : \mathbb{Z} \times \mathcal{X} \rightarrow G$. The *event hash* function defines that $\mathbb{Z}$ is denoted as the ring of integers, which is multiplied into the domain $\mathcal{X}$ and mapped to an element of the group G, which is computed as [26]:

$$h_{eventhash}(r, x) := g^{r \cdot \varphi(h_T(x))} \tag{6.1}$$

The first step of the computation is to blind the *template hash* by generating a random integer $r \in \mathbb{Z}$. Next, the x is an element of the *template hash* domain $x \in \mathcal{X}$. The injective function $\varphi : \mathcal{Y} \rightarrow \mathbb{Z}$ maps $\mathcal{Y}$ into the integer domain $\mathbb{Z}$. After establishing the *event hash* for each measured application/software component, the *event hash* will be extended into the TPM and saved into the SML instead of the *template hash* column (step 4 and 5 in Figure 6.2) (note: The scalar r is not saved in the SML).

6.3.2 Non-Interactive Zero-Knowledge Proof Generation

After generating the *event hash*, the attester (i. e. IMA) needs to generate a (non-interactive) “proof of knowledge” based on the well-known Schnorr signature and the Fiat-Shamir heuristic, which is then provided to the verifier during the remote attestation process (step 3 in Figure 6.2). The non-interactive zero-knowledge proof needs to be generated for each measured binary or software component in the SML. Using the notation from the previous section (Section 6.3.1), the attester (in IMA) computes for every measured entry (each measured binary or software component) i a generator $g_i := g^{\varphi(h_T(x_i))}$ [26]. Next, it chooses a random integer $v_i \in \mathbb{Z}$ and computes the value $t_i := g_i^{v_i}$. Afterwards, the challenge $c_i := H(g_i, t_i, h_{eventhash}(r_i, x_i))$ is generated, where H is defined as a cryptographic hash function [26]. Finally, the attester computes the scalar $s_i := v_i - c_i r_i \mod |L|$. The pair (c_i, s_i) is the “extra data” for each entry in the newly designed privacy-preserving SML. Hence, after computing the “extra data” it will be stored with the associated *event hash* (step 5 in Figure 6.2). The “extra data” is necessary for the verifier to verify the *event hash* since the verifier has zero knowledge of the key r_i. The computation steps are based on the standard Schnorr Non-Interactive Zero-Knowledge (NIZK) proof over an elliptic curve (see Section 2.9).

PCR Index	Event Hash	IMA Template	File Hash	File Path	c	s
10	c4456…331	ima-pp	65727…12b	boot_aggregate	b4612…1ac	3cbd2…a47
10	73c9b…eff	ima-pp	153f3…c95	/lib64/ld-linux-x86-64.so.2	9ce45…2bf	feb14…476
10	9b0ed…c09	ima-pp	4ebaf…c9a	/lib/x86_64-linux-gnu/libc.so.6	2bb78…100	2199e…fd5
10	fef56…71b	ima-pp	64743…f69	/bin/dash	5234d…edd	06069…e68
⋮	⋮	⋮	⋮	⋮	⋮	⋮
10	ee599…667	ima-pp	fc66b…cef	/bin/mkdir	de567…4bd	7fc90…a2b
10	089c4…4bf	ima-pp	2f43e…699	/bin/ln	7d400…031	6b1a7…e90
10	78bd2…14c	ima-pp	e46fd…b48	/bin/mount	067d…93d	d7fc6…70c
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.1: Privacy-Preserving SML (SML_{ima-pp}) [26].

6.3.3 Privacy-Preserving Stored Measurement Log

The newly designed SML for privacy introduces a new column *event hash*, replacing the *template hash* (see Table 6.1). Further, the privacy-preserving SML is presented in Figure 6.2 as *PP-SML*. The column *IMA template* presents the new *IMA template ima-pp*. As mentioned in the previous subsections, an element $r_i \in \mathbb{Z}$ is randomly chosen for each entry i in the SML, which will be used to compute the *event hash* value $h_{eventhash}(r_i, x_i)$. x_i is an element of the domain $\mathcal{X}$ of template hashes. The element x_i (*template hash*, see Section 2.4) is a concatenation of *File Hash* and *File Path* of each entry i in the SML (the integer i can be considered as the index of a log entry). Besides, the columns c and s are known as the extra data as shown in Table 6.1. The pair (c,s) is necessary for the verifier to verify the Schnorr signature or the "proof of knowledge".

6.4 Remote Attestation Process

The second process of the PPRA is the *remote attestation process*. It has the same challenge-response procedure as the traditional RA. Except, the verifier needs to send more information than the traditional binary attestation. In the case of the PPRA, the verifier triggers the RA process and sends in addition to a nonce and a PCR selection, a selection of software, introduced as *swSelection* (step 6 in Figure 6.2). The *swSelection* is a subset of entries of the SML and the subset is affiliated to the particular verifier. After receiving the attestation request from the verifier, the attester checks the *swSelection* with a policy, if the selected software is affiliated to the verifier (step 7 in Figure 6.2). This policy contains all affiliated software/binaries for each verifier. If the crosscheck between the policy and *swSelection* is valid, then the attester requests the TPM to generate a *TPM Quote* (i. e. signed PCRs from the TPM) (step 8 in Figure 6.2). Next, the TPM sends the generated *TPM Quote* (signed by the TPM) to the attester (step 9 in Figure 6.2). Subsequently, the *TPM Quote*, the values of the complete column *Event Hash* of the SML and the entries of the SML that correspond to the selected binaries ($S \subset SML_{ima-pp}$) are sent to the verifier by the attester (step 10 and 11 in Figure 6.2). The verifier receives the message from the attester. Further, the verifier does the same verification steps as described in Section 2.6. Therefore, this work recalls the steps again in the verification process. First, the verifier verifies the TPM signature by using the public key (AK_{pub}) of the Attestation Key (AK) (step 12 in Figure 6.2). Second, the verifier verifies, whether the external data (nonce) matches with the previously sent nonce to the attester, in order to trigger a remote attestation (step 13 in Figure 6.2). The major difference appears when the verifier reaches the step to verify the integrity of the SML.

The verification process of the SML is divided into:

- Verifying the whole SML by accumulating all entries of the *event hashes* and recomputing the PCR value. The verifier compares *accumulative hash* against the actual hash from the *TPM Quote* (step 14 in Figure 6.2).
- Verifying each entry in the SML is not possible anymore, since only the affiliated entries to the verifier will be validated by verifying the Schnorr signature. In order to verify, the verifier computes for each of the affiliated entries the generator g_i which is the same as in the *measurement process*. Further, the scalar $t_i' := g_i^{s_i} \cdot h_{eventhash}(r_i, x_i)^{c_i}$ is computed, which will be used in the cryptographic hash function H to compute $c_i' := H(g_i, t_i', h_{eventhash}(r_i, x_i))$ (note: the corresponding *event hash*, c_i, s_i can be directly read from the column *eventhash, c, s* of the privacy-preserving SML) [26]. Finally, the verifier accepts the *proof of knowledge*, if and only if $c_i == c_i'$ (step 15 in Figure 6.2). This verification process allows the attester to prove to the verifier, that the *event hash* is the *blinded template hash* without revealing the key. By applying this procedure the verifier has "*zero-knowledge*" over the integer r_i.

In the end, the measurement verification is a procedure which takes the received SML from the attester and checks against the Reference Integrity Measurements (RIMs). However, in case of this approach, the RIMs contains only the verifiers associated measurements (step 16 in Figure 6.2).

Figure 6.3 visualizes the remote attestation process as described above, separated from Figure 6.2. Figure 6.3 exists to give an understanding that *measurement process* and *remote attestation process* are separate procedures. However, the *remote attestation process* depends on the *measurement process*. The $E \subset extraData$ defines the subset of the corresponding pairs (c_i, s_i) of the verifier's software selection (see Figure 6.3). Here, this work illustrates in Figure 6.4 with mathematical notation the adapted version of the non-interactive zero-knowledge proof from Section 2.9. This embellishes the adaption to show that the non-interactive zero-knowledge proof is used as sub-protocol of the PPRA approach. Further, Figure 6.4 visualizes the mathematical steps on the attester side for the *measurement process* (to generate the Schnorr signature). Next, the figure presents on the verifier side the mathematical steps for the *remote attestation process* (to validate the Schnorr signature/"proof of knowledge"). The figure demonstrates the core of the PPRA protocol, since the concept relies on the mathematical operations to preserve privacy.

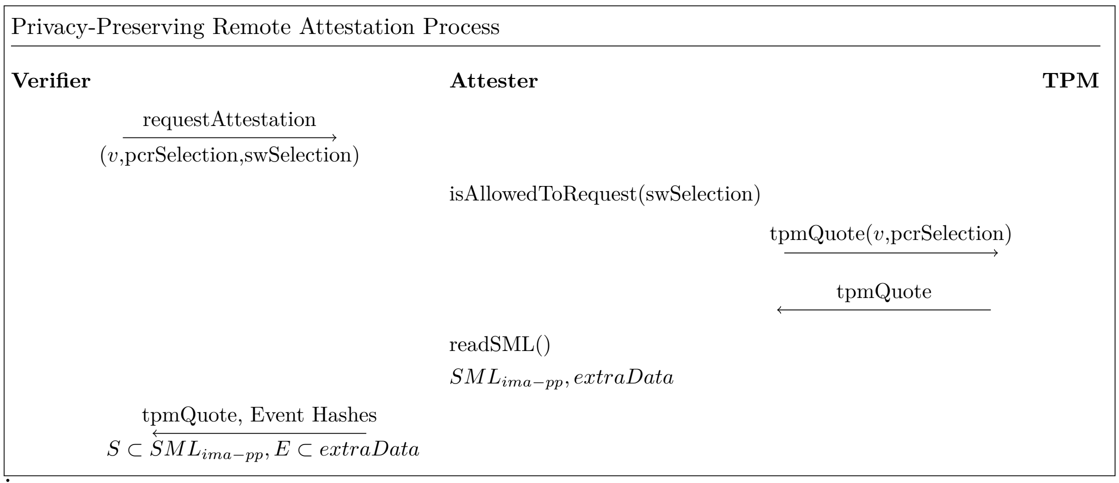
Privacy-Preserving Remote Attestation Process
Verifier
Attester
TPM
requestAttestation
(v,pcrSelection,swSelection)
isAllowedToRequest(swSelection)
tpmQuote(v,pcrSelection)
tpmQuote
readSML()
$SML_{ima-pp}, extraData$
tpmQuote, Event Hashes
$S \subset SML_{ima-pp}, E \subset extraData$

Figure 6.3: Privacy-Preserving Remote Attestation process between verifier and attester.

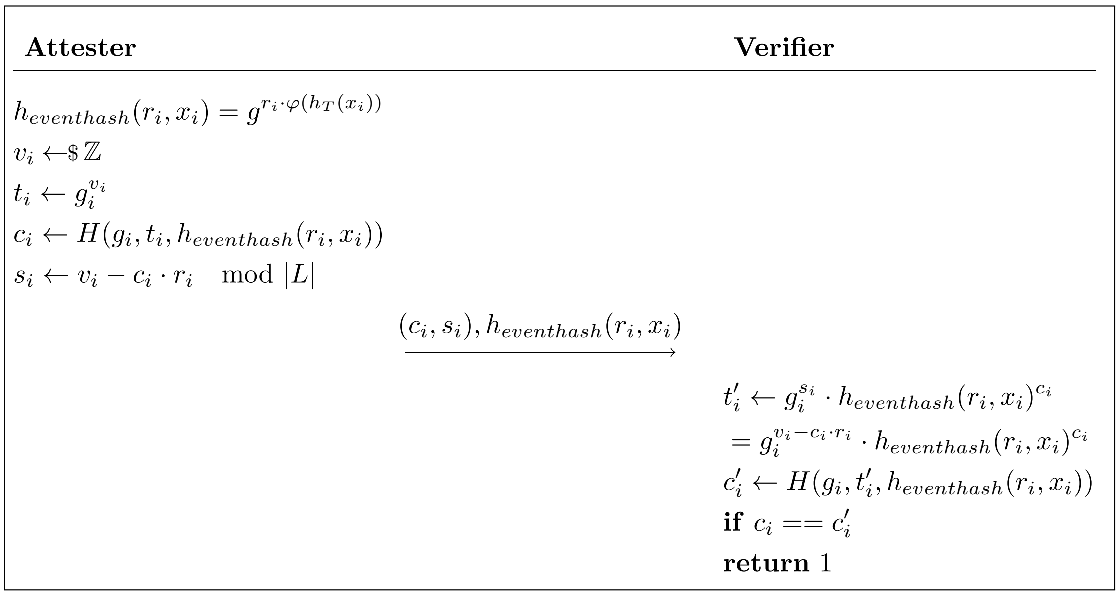
Attester
Verifier
$h_{eventhash}(r_i, x_i) = g^{r_i \cdot \varphi(h_T(x_i))}$
$v_i \leftarrow\$ \mathbb{Z}$
$t_i \leftarrow g_i^{v_i}$
$c_i \leftarrow H(g_i, t_i, h_{eventhash}(r_i, x_i))$
$s_i \leftarrow v_i - c_i \cdot r_i \mod |L|$
$(c_i, s_i), h_{eventhash}(r_i, x_i)$
$t'_i \leftarrow g_i^{s_i} \cdot h_{eventhash}(r_i, x_i)^{c_i}$
$= g_i^{v_i - c_i \cdot r_i} \cdot h_{eventhash}(r_i, x_i)^{c_i}$
$c'_i \leftarrow H(g_i, t'_i, h_{eventhash}(r_i, x_i))$
if $c_i == c'_i$
return 1

Figure 6.4: Adapted Non-Interactive Zero-Knowledge Proof.

CHAPTER 7

Proof of Concept

This chapter elaborates on how the theory of the previous Chapter 6 can be implemented as a Proof-of-Concept (PoC). Therefore, this work analyzes which elliptic curve is suitable for the implementation of the privacy-preserving remote attestation protocol in C. Followed, by presenting the libraries used for the implementation. Further, this chapter gives an in-depth explanation of the architectural implementation design. In the end it embellishes the major algorithms of this work.

7.1 Elliptic Curve Analysis

This work first analyzed, which elliptic curve is suitable for the PoC. The requirements are, it needs to be safe and the curve size is restricted to 32 bytes. The size is important because the hardware Trusted Platform Module (TPM) 2.0 on the target platform (attester) supports a data size of 32 bytes for the *PCR Extend* operation. Therefore, this thesis researched for curves which fulfill the requirements and these are :

- secp256k1 [50, 9]
- Curve25519 [50, 9]
- NIST P-256 [9]

Curve *NIST P-256, secp256k1* and *Curve25519* provide suitable size for the TPM 2.0. However, the curve size is not enough, the curve should be *safe* as well. Therefore, Tanja Lange and Daniel J. Bernstein (D.J.B.) established a webpage called "*SafeCurves: choosing safe curves for elliptic-curve cryptography*" [9], where they list known elliptic curves and arguments which one is safe to use, based on their security properties and in-depth analysis. In the case of this work, the *Curve25519* is safe to use, based on their analysis [9]. However, curve *secp256k1* and *NIST P-256* are not safe. One of the properties which they do not provide is indistinguishability from random strings. The elliptic curve points are normally distinguishable from uniform random strings [7]. The problem is that the Elliptic Curve Cryptrography (ECC) protocols send the points of the elliptic curve in clear text such as long-term public keys, challenges, and ciphertexts [7]. "*Hence, the points are distinguishable from uniform random strings for the attacker*" [5]. D.J.B. states that ECC-based protocols face the same issue [5]. Therefore, it is necessary to construct a mapping of the points, which is then indistinguishable from uniform random strings. Since this work develops a PoC protocol, it is necessary to know if the selected elliptic curve provides these properties. For further details, D.J.B. et al. [6] explain how to map the elliptic curve points to b-bit strings, so the adversary cannot distinguish between uniform random strings and the points. Next, the curve *secp256k1* and *NIST P-256* does not support the ladder method, which is an important computation method in ECC for single-scalar multiplication (see Section 2.8.2). The *Curve25519* supports the Montgomery ladder, where it needs only one input and it uses internally only two coordinates to represent x [8]. The ladder improves the efficiency of the singular-scalar multiplication [8]. For further details see [8]. Further, the Montgomery ladder method is resistant against side-channel attacks [54]. After analyzing the properties of the curves for the protocol, the elliptic curve which provides efficiency and security is *Curve25519*. In the end, this work selected the *Curve25519*, which will be elaborated on in the next subsection.

7.1.1 Curve25519

The *curve25519* has a Montgomery form which was defined by D.J.B. over a prime field [3]:

$$M_{A,B} : y^2 = x^3 + 48662x^2 + x \tag{7.1}$$

, where B= 1 and A = 486662 (see Chapter 2). The prime number $2^{255} - 19$ for the prime field is the finite field size. D.J.B. elaborates in his paper that the *Curve25519* is secure and highly efficient [3]. Therefore, computing the *Curve25519* in high speed is a great advantage in the embedded systems domain. The generated key size for the public and private key are 32 bytes.

In the case of this work, the curve cannot be directly used. Due to the design of the Privacy-Preserving Remote Attestation (PPRA), it uses the Schnorr signature scheme. Therefore, the curve *Ed25519* is necessary. *Curve25519* is recommended for the Diffie-Hellman key exchange method and for digital signature scheme twisted Edwards curve (*Ed25519*) [37, 3]. Bernstein et al. stated in [4] for signature scheme they chose Ed25519, since it provides fast scalar arithmetic and the equivalent security as the *Curve25519* [4]. The *Curve25519* can be used for digital signature operations, however, Ed25519 is more efficient [37, 4]. T.Lange and D.J.B. states in their paper [4] that *Curve25519* is birationally equivalent to the twisted Edwards curve (birationally equivalent means that a small set of points from the *Curve25519* cannot be mapped to the twisted Edwards curve) (*Ed25519*). For a detailed understanding of the twisted Edwards curve *Ed25519*, see following reference [4].

The curve *Ed25519* and *Curve25519* have the order $8L$. L is the large prime number: $2^{252} + 27742317777372353535851937790883648493$. However, the paper from Brendel et al. [14] states that the Schnorr signature scheme needs to be established over primer order groups. Hence, non-prime order groups make the process more complicated due to its complex group structure (such as Curve25519 *and* Ed25519). The cofactor 8 provides four sub-groups, which can lead, that the computed element (the point on the curve) does not lie in the prime order subgroup [14]. Therefore, there exists a technique called Ristretto, which resolves this complication.

7.1.2 Ristretto

The Ristretto technique was designed by Mike Hamburg, where he introduced a method for constructing prime order elliptic curve groups. Mike Hamburg stated in his work [32] a decaf design, where he introduced a technique to create a group of prime order L based on twisted Edwards curve with a cofactor 4. This technique is more efficient than checking the membership of the element if it is in the correct subgroup and eliminates the cofactor. This check is complex and inefficient. Ristretto is the extended technique for curves with the cofactor 8 such as *Curve25519*/*Ed25519*. Furthermore, existing systems with *Ed25519* signatures can be extended with complex zero-knowledge proof protocols by applying the Ristretto technique [46].

Moreover, the cofactor provides some pitfalls such as small-subgroup attack. The sub-group attack is if the attacker retrieves information about the scalar, which is arithmetically used for a private key [32]. For instance, the attacker sends or replaces the point P with T, where P is a point of order L and T is a point of order 8 or is a point in a small subgroup [32]. Following, the user generates a random scalar r, which is the private key, and multiplies it with T, and it results in rT. Hence, if the point rT is known to the adversary, s/he can retrieve the scalar r, since T is in a small group and not in a large prime order [32]. This attack can be prevented by using a prime order group with an appropriate size.

The Ristretto method extends the existing Edward curve implementation for complex protocols such as this work. Therefore, the existing Edwards curve needs functions, which map Edwards point to Ristretto point, and a validation function that checks if the canonical encoding of the Ristretto point is valid. In conclusion, the Ristretto technique guarantees no further cryptographic assumptions are necessary and is safely extendable for Ed25519 signature with zero-knowledge protocols [46].

7.2 Libraries

Here, the section explains which crypto library was chosen and other libraries to realize the PPRA protocol. The crypto library selection was based on whether the library supports the Curve25519 implementation. First, the Nettle library [49] with GMP library [29] was considered, but the issue was the GMP library is not in constant time. Therefore, this work researched further, to find a library that guarantees constant time and has features such as group operations and scalar arithmetic pre-implemented. In the end, the LibSodium library met all the conditions.

7.2.1 Libsodium

LibSodium [21]is a well known crypto library. Normally, Libsodium provides the implementation for encryption, decryption, signatures, and password hashing. Regarding this work, the API of LibSodium provides an advanced section [22], where implementors can develop cryptographic protocols based on Ed25519 or within this work, the Ristretto technique. Further, it provides group operations such as point addition and point multiplication based on the Ristretto technique. Next, it supports functions for scalar arithmetic. The Nettle library did not provide any further scalar arithmetic operations. Therefore, the GMP library is a necessity for scalar arithmetic operations. Here, the Libsodium library was chosen since it provided all the operations and no extra library is needed.

7.2.2 tpm2-tss

The *tpm2-tss* [28] is a library which provides the API for hardware TPM 2.0 as well as software TPM. The TSS stands for TPM 2.0 software stack. Further, the software stack is layered in the following structure [28]:

- Feature API (FAPI)
- Enhanced System API (ESAPI)
- System API (SAPI)
- Marshaling/Unmarshaling (MU)
- TPM Command Transmission Interface (TCTI)

This work will be working with Enhanced System API. The feature API makes it for developer easier work with TPM 2.0 since this API exposes high-level functions. The Enhanced System API maps one to one of the TPM 2.0 commands to functions. These functions are used in this work to interact with the hardware TPM 2.0.

7.2.3 CHARRA

CHARRA (CHAllenge-Response based Remote Attestation with TPM 2.0) [45] is a project, which was brought to life by Michael Eckel. The project is a proof-of-concept implementation for challenge-response remote attestation model. From this project, this work used few implementation components to realize the PPRA protocol. Further, the CHARRA project is used to compare the performance measurements between the functions of CHARRA and the PPRA protocol.

7.2.4 QCBOR

QCBOR [12] is the C implementation of the RFC 8949 [13] standard. CBOR stands for Concise Binary Object Representation, which uses the same concept of JSON. CBOR is a data format, which does not need a schema. Further, it is fast to decode and encode for defined CBOR formats. In this project, the library is used to map the C structs into binary with a given format as Concise Data Definition Language (CDDL). After encoding the C struct into a binary format it can be sent as bulk through the communication channel and afterwards it can be easily decoded if the CDDL is known. QCBOR can be seen as a library, which supports the developer in C to serialize and deserialize the C structs (such as DTOs).

7.2.5 mbedTLS

mbedTLS [60] is a C library customized for embedded systems. It implements crypto primitives. Further, it provides an API for SSL/TLS and DTLS protocol. Moreover, this library is used in this work for encrypted communication between attester and verifier. However, this is not the only feature this work uses, the library provides mutual authentication between attester and verifier with certificates. Explained later in the *implementation* section.

Figure 7.1 illustrates how the previously described libraries are used in the project. The *tpm2-tss* is used to communicate with hardware TPM 2.0. Next, the *QCBOR* library is used for encoding and decoding binaries into C structs. As already mentioned, *mbedTLS* is used for encrypted communication and mutual authentication.

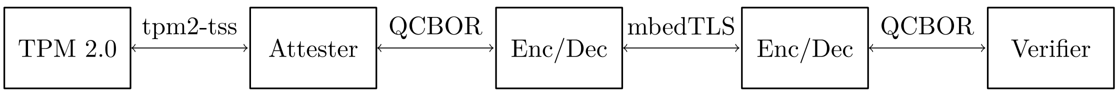

Figure 7.1: Library usage.

7.3 Implementation

This section elaborates how the theory of Chapter 6 will be realized as a proof-of-concept. As already mentioned the PoC will be implemented in C.

7.3.1 Overview

The PoC will be implemented in the user space. In the future, it should be mapped into the kernel space. Therefore, it is written in C, which could be "*easily mapped*" from user space into the kernel space. However, this is not only the reason for implementing the approach in C, the API for the TPM is provided in C as well.

The reason for the conversion from user space into kernel space is, because of the Integrity Measurement Architecture (IMA), which is a part of the Linux kernel. Even though the IMA is in kernel space, the PoC is in user space, where the IMA is simulated as a function (see subsections below). Nevertheless, this work focuses on how to implement the core concept of PPRA protocol, which proves the partial integrity of the (verifier's affiliated) Stored Measurement Log (SML) entries with non-interactive zero-knowledge proof, while maintaining the integrity of the entire system's operational state and authenticity. Therefore, the IMA process is simulated as a function and as future work the PPRA protocol should be integrated into the IMA. To understand the IMA process of the Linux kernel, Figure 7.2a visualizes how the measurement process is executed in IMA. First, the IMA generates from the file hash and file path a *template hash* with *SHA-1*. The original Linux IMA supports only *SHA-1* for generating the *template hash*. Second, the *template hash* will be extended and anchored into the TPM, and stored into the SML. However, the approach of this work upgrades the simulated IMA by supporting the hashing algorithm *SHA-256* (see Figure 7.2b) (note: this work implemented the approach based on the assumption of the "upgraded IMA" using a collision-resistant hashing algorithm). Next, Figure 7.2b visualizes after generating the *template hash* that Non-Interactive Zero-Knowledge (NIZK) proof is applied to compute the *event hash*. Afterwards, the scalars c, s are computed and stored with the *event hash* into the SML. Further, the *event hash* is extended into the TPM. Here, this work recaptured the IMA measurement process to illustrate the scope of this work and what will be implemented, and which process will be simulated. The detailed implementation of the simulated measurement process will be elaborated on in the upcoming subsection. Furthermore, the Remote Attestation (RA) process as described in Chapter 6, will not simulate any functions, since it is the major focus of this work, and it will be tested on real hardware.

7.3.2 Design

Here, this work demonstrates the implementation design to understand how the PPRA protocol was realized. The PoC implementation consists of eight C header files and for each header file an implementation. Figure 7.3 illustrates the dependency between the header files. Besides, it shows the implementation design of the PoC.

attestphase.h is the core of the implementation since it includes most of the surrounding header files. The *attestphase.h* consists the implementation for the *measurement process and remote attestation process*. Further, it implements encoder and decoder of the *QCBOR* library functions for the C structs, which is used to convert them into binary and save them into files (such as SML).

nizk.h implements the NIZK signing and verification functions, where the NIZK signing function is used in the *measurement process* and the NIZK verification function in the *remote attestation process*. These functions will be explained in the upcoming subsection, how they were designed.

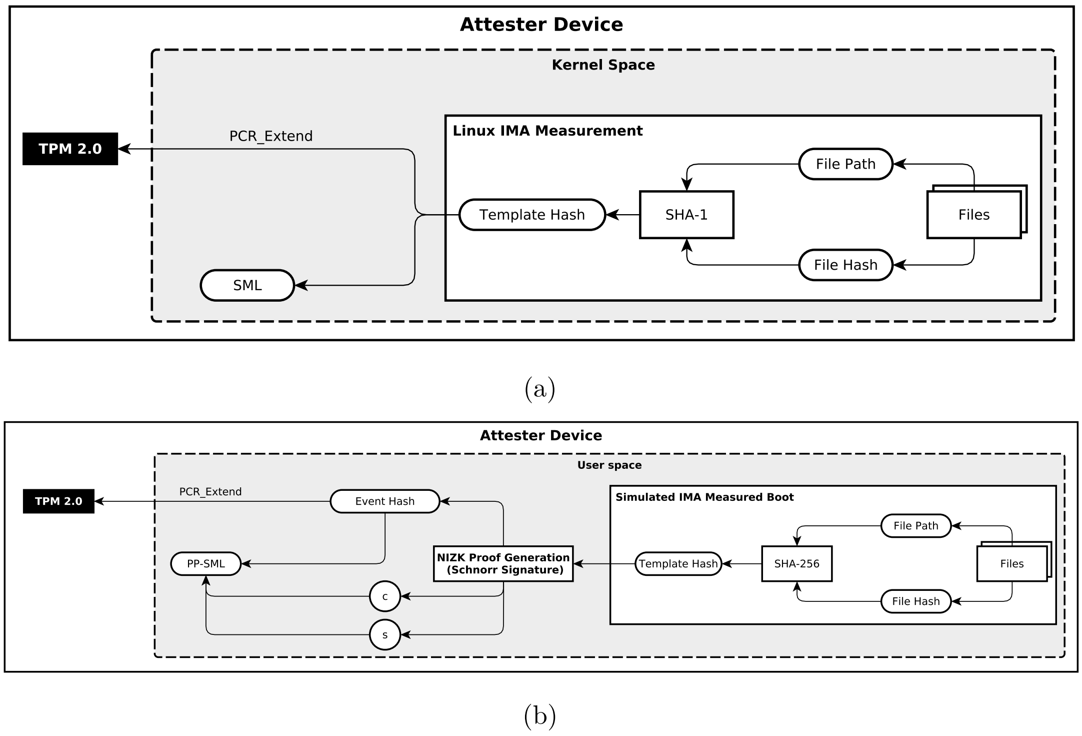

Figure 7.2: (a) illustrates the original Linux IMA measurement process. (b) illustrates the simulated IMA measurement process in the PoC (implementation).

ppra_dto_message_encdec.h implements the encoder and decoder with the *QCBOR* library for the data transfer objects (dto), which will be encoded into binary format and then decoded.

ppra_dto.h defines the C structs, what kind of information should be transferred between the functions as well as between attester and verifier.

tpm2_util.h contains the invocation of the tpm2 commands from the *tpm2-tss* library ESAPI. The header does not contain all the tpm2 commands, only the functions, which are necessary for realizing the PoC.

hash_sig_verify.h contains the implementation of functions for hashing and verifying the signature of the *TPM_QUOTE* with the *mbedTLS* library.

tpm2_charra is the package, which contains three header files, these provide functions from the *CHARRA* project, such as how to manage the attestation key from the TPM and how to convert the Platform Configuration Register (PCR) selection into an understandable format for the TPM.

attester.c invokes the functions from *attestphase.h* for generating all the *event hashes* for the measured binaries (which is a simulated data set). Next, *event hashes* will be extended into the TPM. The next step is to check if the attestation key exists in the NVRAM of the TPM. If not, a new key is generated and stored in the NVRAM. After all these steps, the attester publishes its socket information and waits for a connection. The *attester.c* implements the mbedTLS connection and certification verification.

verifier.c establishes a socket as well to connect to the attester, where they use the feature of mbedTLS to establish encrypted communication. Afterward, the verifier sends an attestation request and verifies the response from the attester, where it checks if the nonce, signature, *TPM QUOTE*, and NIZKs are valid.

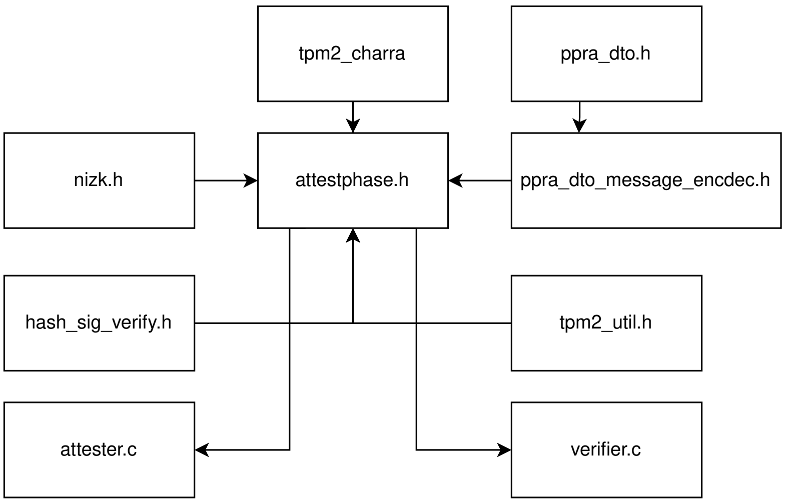

Figure 7.3: Implementation design.

7.3.3 Non-Interactive Zero-Knowledge Proof

Here, this work explains the core feature on which the whole concept relies on. This work provides two algorithms as pseudocode for the NIZK and provides the equivalent implementation of both algorithms in C in Chapter A (the implementation is in *nizk.h*). The Algorithm 7.1 illustrates the pseudocode how to implement the Schnorr signature (NIZK signing or NIZK proof generation). The Algorithm 7.1 takes as input an *event*, which contains a measured software/binary with the file path, file name, and the file hash. Next, on line 2 it generates the *template hash*. On line 4 it rehashes the *template hash*. The extension rehashes the *template hash* as SHA-512. The reason for the extension is that the implementors of Ristretto, mentioned in [66, 67] that the scalars needs to be in the range of 0 and L ($L = 2^{252} + 27742317777372353535851937790883648493$, the primer order group of curve25519). Nevertheless, Ristretto scalars are designed to be in the range of 0 and L for each arithmetic operation. Therefore, in this work, the length of the *template hash* is extended by rehashing or by concatenating the *template hash* twice and then reducing it with the modulo operation. Hence, the *template hash* is a SHA-256 hash, which has a size of 32 bytes, but the sampled bits are larger than L ($2^{256} > 2^{252} + 27742317777372353535851937790883648493$). Another option is to check if the value of the *template hash* is in the range, which could lead to extra resource costs. Further, the Libsodium implementation of the modulo operation only accepts 64 bytes as input and outputs a uniformly distributed value between 0 and L. Therefore, these extending approaches were used during the implementation to overcome the hurdle.

Next, the Algorithm 7.1 has validation checks, which checks after point multiplication (scalar multiplication) if the value is still a valid point on the elliptic curve. The calculation steps of the algorithm resemble the mathematical steps in Figure 6.4. Moreover, line 9 generates a new generator, if it is a valid point on the curve it will be accepted as a generator. Normally, the basepoint of the elliptic curve is known as the generator, but if the base is used in the scalar multiplication and the output is a valid point on the curve it can be used as a generator as well (see Section 8.1). Further, the lines 5, 6, 15, 16, 17 in the Algorithm 7.1 uses the modulo operation, so that the scalars in the arithmetic operations are in the range. At the end, the algorithm returns the *event hash* and the scalars c, s.

Algorithm 7.1: NIZK Signing

```
   Input: $event$
   Output: $event_{hash}, c, s$
 1 Function nizksign($event$):
 2     $templatehash \longleftarrow h_T(event)$;
 3     $r \longleftarrow random()$;
 4     $templatehash_{extended} \longleftarrow h_{512}(templatehash)$;
 5     $reduced_{digest} \longleftarrow templatehash_{extended} \mod L$;
 6     $r_h \longleftarrow r \cdot reduced_{digest} (\mod L)$;
 7     $event_{hash} \longleftarrow g^{r_h}$;
 8     if $event_{hash}$ is not a valid point then return error;
 9     $g_i \longleftarrow g^{reduced_{digest}}$;
10     if $g_i$ is not a valid point then return error;
11     $v \longleftarrow random()$;
12     $t_i \longleftarrow g_i^v$;
13     if $t_i$ is not a valid point then return error;
14     $c \longleftarrow H(g_i \| t_i \| event_{hash})$;
15     $reduced_c \longleftarrow c \mod L$;
16     $r_c \longleftarrow r \cdot reduced_c (\mod L)$;
17     $s \longleftarrow v - r_c (\mod L)$;
18     return $event_{hash}, c, s$;
19 End Function
```

The next Algorithm 7.2 is the NIZK verification, which has as input an *eventrecord*.

The input *eventrecord* contains the *event hash, c, s and event.* The algorithm returns true, if the validation was successful, otherwise false. The lines 3-7 are the same as in the previous Algorithm 7.1, where the *template hash* is generated and extended, and the same generator is generated if the *eventrecord* contains the same *event* used in the Algorithm 7.1. The interesting part is on line 13 in Algorithm 7.2, where the two valid points on the curve are added together which results in $t_i\prime$. If and only if the $c\prime$ is equivalent to *eventrecord.c* then the proof of knowledge is successful. In the verification algorithm, the modulo operation was used as well. Therefore, the next subsection analyzes the modulo operation with the method of the **statistical distance**.

Algorithm 7.2: NIZK Verification

```
   Input: $eventrecord$
   Output: $verify$
1  Function nizkverify($eventrecord$):
2      $verify \longleftarrow$ false;
3      $templatehash \longleftarrow h_T(eventrecord.event)$;
4      $templatehash_{extended} \longleftarrow h_{512}(templatehash)$;
5      $reduced_{digest} \longleftarrow templatehash_{extended} \mod L$;
6      $g_i \longleftarrow g^{reduced_{digest}}$;
7      if $g_i$ is not a valid point then return error;
8      $gi_s \longleftarrow g_i^{eventrecord.s}$;
9      if $gi_s$ is not a valid point then return error;
10     $reduced_c \longleftarrow eventrecord.c \mod L$;
11     $eventhash_c \longleftarrow eventrecord.event_{hash}^{reduced_c}$;
12     if $eventhash_c$ is not a valid point then return error;
13     $t_i\prime \longleftarrow gi_s \cdot eventhash_c$
14     $c\prime \longleftarrow H(g_i \| t_i\prime \| event_{hash})$;
15     if $c\prime == eventrecord.c$ then $verify \longleftarrow$ true;
16     return $verify$;
17 End Function
```

Modulo operation

The proof-of-concept uses the modulo operation in the mathematical computation steps for creating a signature and verifying it, as described in the previous section. The reason for applying the modulo operation is to create uniformly distributed values in the range of the group order of the Ed25519 (same as in Curve25519). Each mathematical operation of the NIZK signature generation, as well as the verification, needs to be uniform values. Since the template hash and challenge c are not uniform values. Thus, it is necessary to apply the modulo operation with the modulus of the prime group order of the Ed25519. Therefore, it is necessary to show, the statistical distance between the modulo operation and the input of a specific bit string. The statistical distance tells the distance between

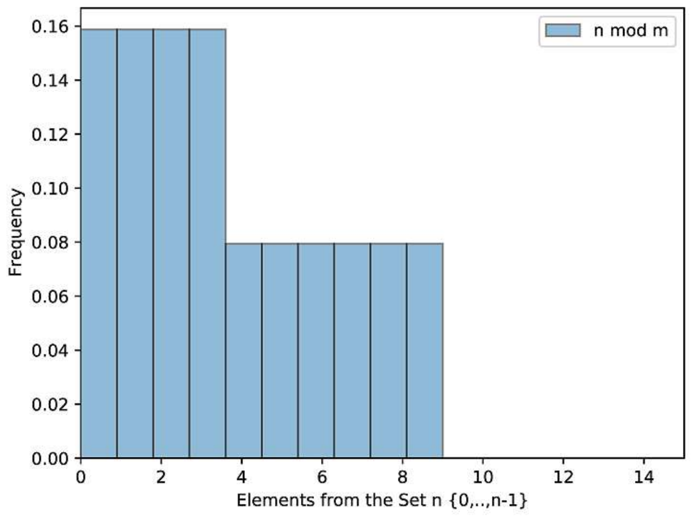

Figure 7.4: Example distribution of the modulo operation.

two probability distributions. In cryptography, the statistical distance is used to see whether the distance is negligible between two distributions. In other words, if the distance is negligible, a powerful and unbounded adversary cannot distinguish them [27]. Hence, the probability distribution over modulo operation will be here generalized. Let $\mathcal{X}$ be a uniform distribution over X, where set $X = \{0, ..., n-1\}$ and $\mathcal{Y}$ is an uniform distribution over Y, where set $Y = \{0, ..., m-1\}$. Let $\mathcal{W} : X \to Y$ be a transformation or function $\mathcal{W}(x) = x \mod m \in \{0, ..., m-1\}$, where $X > Y$. The question is, what is the statistical distance between $\mathcal{Y}$ and $\mathcal{W}(\mathcal{X})$?

The first step to answer the question is to find out the distribution of $\mathcal{W}(\mathcal{X})$. Therefore, it is necessary to analyze how the samples are distributed in the transformation $\mathcal{W}(\mathcal{X})$. For this purpose, this section provides an example where $n = 15$ and $m = 10$ and the Figure 7.4 illustrates the distribution of the modulo operation of n and m. Further, Figure 7.4 shows that the first four values appear twice than the rest. This indicates that the distribution of $\mathcal{W}(\mathcal{X})$ of a sample has two cases. The two cases are representing the probability of a sample of the transformation $Pr[\mathcal{W}(\mathcal{X}) = x]$, stated in Equation (7.2). If $x \in \{0, .., n \mod m\}$ then the probability for the sample x is $Pr[\mathcal{W}(\mathcal{X}) = x] = \frac{\lfloor \frac{n}{m} \rfloor + 1}{n}$. However, if $x \in \{n \mod m, .., m-1\}$ then the probability is $Pr[\mathcal{W}(\mathcal{X}) = x] = \frac{\lfloor \frac{n}{m} \rfloor}{n}$.

$$Pr[\mathcal{W}(\mathcal{X}) = x] = \begin{cases} \frac{\lfloor \frac{n}{m} \rfloor + 1}{n} & 0 \leq x < (n \mod m) \\ \frac{\lfloor \frac{n}{m} \rfloor}{n} & m > x \geq (n \mod m) \end{cases} \tag{7.2}$$

After defining the cases for $Pr[\mathcal{W}(\mathcal{X}) = x]$, the next step is to answer the question. Before answering the question it is necessary to understand what the statistical distance states.

The statistical distance in cryptography defines the distance between two distributions. The outcome of the statistical distance tells if it is negligible. This means if the adversary is capable of creating an algorithm in Probabilistic Polynomial Time (PPT) to distinguish between two distributions. Further, the statistical distance is an upper bound over the probability that the computationally bounded adversaries cannot distinguish between two distributions [27]. Therefore, the Equation (7.3) shows how to calculate the statistical distance between the distributions $Pr[\mathcal{Y} = x]$ and $Pr[\mathcal{W}(\mathcal{X}) = x]$. Further, with the statement of the statistical distance, it is possible to determine how negligible the outcome is.

$$\begin{aligned}
A &= \{0, ..., n \mod m\} \\
B &= \{n \mod m, ..., m-1\} \\
SD(\mathcal{Y}, \mathcal{W}(\mathcal{X})) &= \frac{1}{2} \sum_{x \in [0,m)} |Pr[\mathcal{Y} = x] - Pr[\mathcal{W}(\mathcal{X}) = x]| \qquad (7.3) \\
&= \frac{1}{2} \left(\sum_{x \in A} \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor + 1}{n} \right| + \sum_{x \in B} \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor}{n} \right| \right)
\end{aligned}$$

Since, the cases of $Pr[\mathcal{W}(\mathcal{X}) = x]$ and the $SD(\mathcal{Y}, \mathcal{W}(\mathcal{X}))$ are generalized, it is possible to fill n and m with accurate values. In the case of this work, it is necessary to know the statistical distance between the prime group order and 2^{256}, which illustrates 256 bits. The Equation (7.3) was used in SageMath to calculate the SD between the distribution over 2^{256} bits, which represents SHA-256 and over the prime group order $2^{252} + 27742317777372353535851937790883648493$ of the Curve25519. The equation 7.3 was applied in SageMath to calculate the concrete values. The Listing 7.1 represents the calculation process. Further, the result shows, that it is near to negligible.

```
sage: m = 2^252+27742317777372353535851937790883648493
sage: n = 2^256
sage: add0 =((abs((1/m)-((floor(n/m)+1)/n)))*(n%m)).n()
sage: add0
3.83339731895009e-39
sage: add1 = ((abs((1/m)-((floor(n/m))/n)))*(m-(n%m))).n()
sage: add1
3.83339731895009e-39
sage: SD = ((add0+add1)/2).n()
sage: SD
3.83339731895009e-39
```

Listing 7.1: Calculation of SD for 256 Bits.

Listing 7.2 represents the result for 317 Bits. The statistical distance of prime group order and 317 bits was calculated because Libsodium [22] stated that if the input of their modulo function is at least 317 bits, then the output will be at least uniform. Therefore, the calculation was done to see if there are some remarkable differences between the SD of 256 bits and 317 bits.

```
sage: m = 2^252+27742317777372353535851937790883648493
sage: n = 2^317

sage: SD
3.83339731895009e-39
```

Listing 7.2: Calculation of SD for 317 Bits.

As the outcome of Listing 7.2 presents, the result is equal to the result of 256 bits. This means there does not exist much difference and is still negligible. If this work uses a 512-bit string in the modulo operation, it is more negligible than the previous calculation. Negligible means if it is appearing close to zero. In other words, the result defines how near it is to be uniform.

```
sage: m = 2^252+27742317777372353535851937790883648493
sage: n = 2^512
sage: SD
9.40956857410997e-80
```

Listing 7.3: Calculation of SD for 512 Bits.

After applying the statistical distance of different lengths of bit strings, the statement tells it is better to use 512 bits since it is more negligible/near to uniform than the other two bit strings. To conclude a powerful and unbounded adversary is not capable of distinguishing the two distributions and therefore, the extension methods in the Algorithm 7.1 and Algorithm 7.2 are applied.

7.3.4 Measurement Process

Here, this work continues with the implementation of the *measurement process*. As already mentioned in this work, the measurement process of the IMA will be simulated by the function called *simulated_measured_boot()*. The function will be illustrated in this work as pseudocode which resembles the C implementation of the work.

Algorithm 7.3: Simulation of the IMA process.

```
   Input: $filepath$
1  Function simulated_measured_boot($filepath$):
2      $buffer, buffer_{size} \longleftarrow loadDataSet(filepath)$;
3      $event_{list} \longleftarrow events\_decode(buffer, buffer_{size})$;
4      $eventrecords \longleftarrow [event_{list}.size]$;
5      $pcr \longleftarrow 10$;
6      for $i \leftarrow 0;\ i < event_{list}.size;\ i \leftarrow i + 1$ do
7          $evenrecords[i].event \longleftarrow event_{list}[i].event$;
8          $event_{hash}, c, s \longleftarrow nizksign(event_{list}[i].event)$;
9          $eventrecords[i].event_{hash} \longleftarrow event_{hash}$;
10         $eventrecords[i].c \longleftarrow c$;
11         $eventrecords[i].s \longleftarrow s$;
12         $tpm2\_pcr\_extend(pcr, event_{hash})$;
13     end
14     $eventrecords_{encoded} \longleftarrow eventrecords\_encode(eventrecords)$
15     $saveSML(eventrecords_{encoded})$;
16 End Function
```

The Algorithm 7.3 represents the simulation of the IMA process. As input, it takes a *file path*, which is the path of the data set used to simulate the process. The data set contains 250 entries of software binaries of the system *bin* folder of a Linux OS with their file path, file name, and file hash. The data set will be loaded as a byte string and afterward decoded to a C struct filled with the data and saved into $event_{list}$ (Algorithm 7.3 line 2-4). Next, each *event* will be taken from the list and the *event hash* and the scalars c, s are generated, which will be saved into *eventrecords*. To generate these values, the Algorithm 7.1 is invoked on line 8. After saving the values into *eventrecords*, the *tpm2_pcr_extend* [65] function will be invoked. This function/tpm2 command extends the PCR bank of the given PCR number as a parameter. With this approach, the *event hash* will be extended into the Core Root of Trust for Measurement (CRTM) and expands the chain of trust (the Chapter 2 describes the theory behind the PCR extend operation). At the end of the Algorithm 7.3, the *eventrecords* will be encoded as a byte string and saved as the privacy-preserving SML.

After the attester extended the generated *event hashes* into the TPM, it checks if the TPM stored the primary key (Attestation Key (AK)) in the non-volatile RAM (NVRAM)[64].

If not, a new key will be generated and stored in the NVRAM. The reason, this check exists is to reduce the resource overhead on the TPM. Otherwise, for every remote attestation, it generates a new private and public key, and each generation needs around one minute on the hardware TPM. Therefore, this work implemented the function *ppra_create_store_tpm2_key()*, which does the check. Otherwise, if the function does not exist, it would falsify the measurements in the evaluation Chapter 8.

Despite all the description of the *measurement process*, this work illustrates a sequence diagram Figure 7.5 on how each of the algorithms presented above will be invoked from the attester. After invoking all these functions, it establishes a socket and waits until a verifier connects for remote attestation. The *remote attestation process* will be explained in the next subsection.

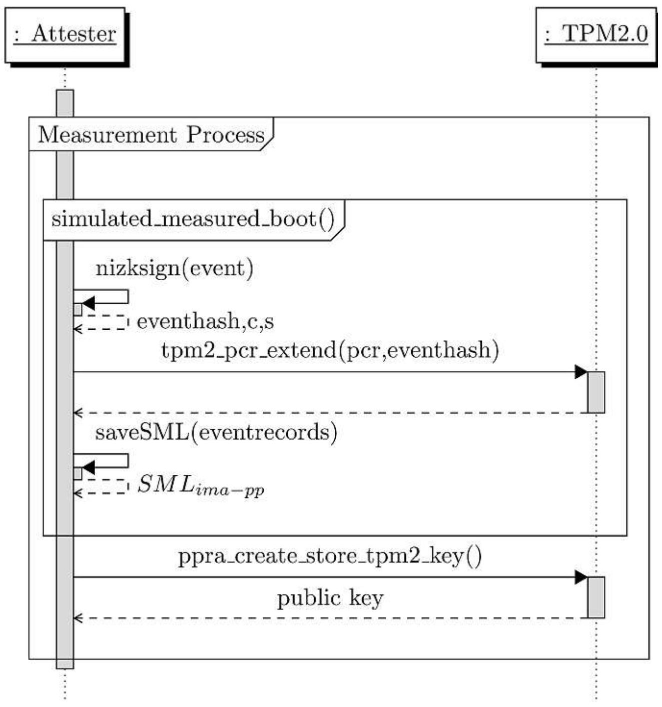

Figure 7.5: Illustration of the function invocations for the measurement process.

7.3.5 Remote Attestation Process

This work continues with the *remote attestation process* and introduces the implementation approach.

Communication

The communication between attester and verifier needs to be secured. Therefore, the *mbedTLS* library is used. Before the remote attestation request is sent to the attester, the verifier establishes a handshake. During the handshake, they do a mutual authentication, where both have a certificate issued by the Certification Authority (CA). Consequently, the project uses a self-signed CA certificate and certificates for the attester and the verifier are derived from the self-signed CA certificate. Hence, this work did not set up a public key infrastructure since it was not the scope of the work. Furthermore, *mbedTLS* provides functions to load the certificates into the attester and verifier. Moreover, each of them loads the self-signed CA certificate and each of theirs. Next, Figure 7.6 illustrates that during the handshake both of them sends each other's certificates and validates the certificate against the CA certificate, which verifies if the attester's and verifier's certificate are valid. In the matter of the project, the certificate authority is internal. In the practical use case, the certificate authority should be an external trusted party in the public key infrastructure. After authenticating each other, they establish a secured end-to-end communication.

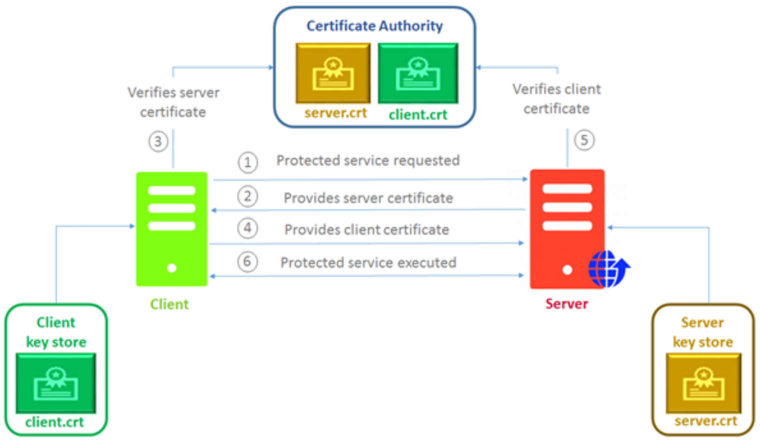
Certificate Authority
server.crt
client.crt
Verifies server certificate
Verifies client certificate
3
5
1 Protected service requested
2 Provides server certificate
4 Provides client certificate
6 Protected service executed
Client
Server
Client key store
client.crt
Server key store
server.crt

Figure 7.6: Mutual Authentication [58].

Attestation Request

Shortly after establishing a secured connection the verifier creates an attest request, which contains a nonce, PCR selection, and the software selection. The created request will be encoded to a byte string based on a Concise Data Definition Language (CDDL) with the *QCBOR* library. Next, the verifier sends the encoded byte string over the secured line to the attester, where it decodes with the known CDDL, the byte string. CDDL is a representation form for Concise Binary Object Representation (CBOR) [10]. As an exemplary structure, Listing 7.4 shows how a CDDL for this project was defined.

```
eventrecord = [
pcr: (0..23) .default 10 ,
event_hash: bstr,
c: bstr,
s: bstr,
events: [+ event]
]


event = [
file_hash: bstr,
file_name: string,
file_path: string,
]
```

Listing 7.4: CDDL example of eventrecords.

However, Listing 7.4 does not only show the structure, but it also defines the data type of each element. Thereby, the data types define how to implement the decoder and encoder by using the *QCBOR* library. Nonetheless, the listing symbolizes similarities to JSON structure.

After the detour in CDDL, this subsection continues with the next steps at the attester side after receiving the attestation request. First, the attester checks the received software selection against a policy. In the case of the work, the policy is represented for simplicity as a file. The name of the file is the extracted certificate ID from the verifier's certificate and loads the file and checks the software selection against the verifier's policy (see Figure 7.7). As the Figure 7.7 illustrates the sequence of the remote attestation process, the function *is_authenticated* does the check. Then the attester reads from the privacy-preserving SML only the entries of the software selection and the whole *event hash* column and stores them into *eventrecords*. Thereafter, the attester loads from the NVRAM the public key of the TPM. Next, the attester requests the *TPM QUOTE* from the TPM, with the nonce, PCR selection and the public key. The TPM selects the PCR bank of the received PCR selection, then rehashes the PCR bank and signs it with the attestation key and returns the *tpmQuote* (see Figure 7.7). In the end, the attester encodes all the information as a byte string sends it over the secured channel to the verifier. The verifier initiates the verification procedure after receiving the response from the attester.

Verification

The verification process has the same procedure as described in Section 2.6. Except, there exists an extra step in the verification, which is to validate the proof of knowledge for each associated entry. Therefore, the Algorithm 7.4 represents the core part of the verification procedure how the verifier verifies each entry. The first step is to decode the response and then to load the Reference Integrity Measurement (RIM). Afterwards, each entry's proof of knowledge will be validated, if one entry is not valid then the whole attestation fails. After each successfully validated proof, the entry will be cross-checked with the entries in the RIM. The measurement verification is partial since the affiliated software components of the verifier are compared against the RIMs (The RIMs only contain the list of the verifier's responsible software components.).

Then the same procedure of the *PCR_extend* function is implemented on the verifier side, where it extends each *event hash* of the received column of the SML from the attester. After generating the folding hash from all the *event hashes* the verifier checks if all the proofs are valid and if all the cross-checks with RIM was valid as well. Next, the verifier validates the signature from the TPM. Subsequently, the verifier checks the nonce if the received one is still the same as the one which was sent to the attester. Finally, the verifier rehashes the *folding hash* and compares it with the actual value of the received *TPM Quote*. If and only if all the validation succeeds, then the algorithm Algorithm 7.4 returns *success*. Otherwise, the remote attestation failed and the integrity and authenticity of the operational state of the attester side are jeopardized.

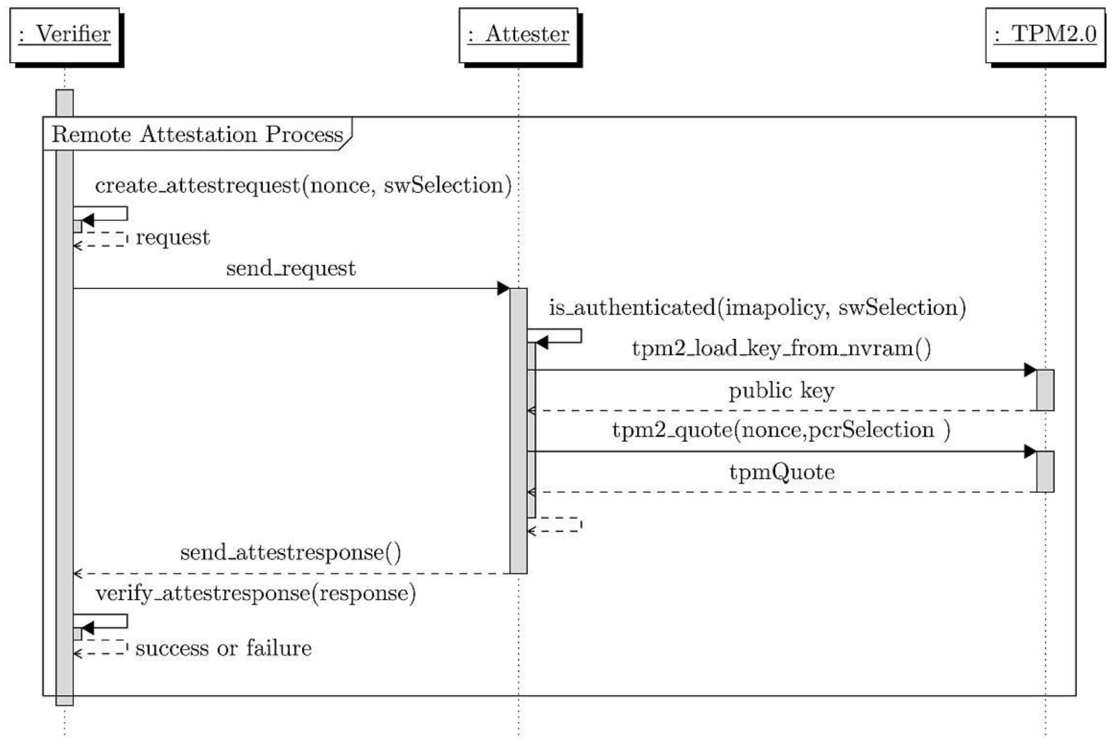
: Verifier
: Attester
: TPM2.0
Remote Attestation Process
create_attestrequest(nonce, swSelection)
request
send_request
is_authenticated(imapolicy, swSelection)
tpm2_load_key_from_nvram()
public key
tpm2_quote(nonce,pcrSelection)
tpmQuote
send_attestresponse()
verify_attestresponse(response)
success or failure

Figure 7.7: Illustration of the function invocations for the remote attestation process.

Algorithm 7.4: Verification of the attestation response.

```
Input: $nonce, rim_{path}, response$
Output: $success \lor failed$
Function verify_attestresponse($nonce, rim_{path}, response$):
    $partial_{integrity} \leftarrow$false
    $rim_{match} \leftarrow$false
    $counter \leftarrow 0$
    $rim_{matchcounter} \leftarrow 0$
    $eventrecords \longleftarrow events\_decode(response.eventrecords)$
    $rim \leftarrow loadRIM(rim_{path})$
    $pcr \leftarrow 10$
    for $i \leftarrow 0;\ i < eventrecords.size;\ i \leftarrow i + 1$ do
        if $event_{records}[i].cis\ not\ empty$ then
            if $nizkverify(eventrecords[i])$ then
                $break$
            end
            for $j \leftarrow 0;\ j < eventrecords.size;\ j \leftarrow j + 1$ do
                if $rim[j].fname == evenrecords[i].event.fname$ then
                    $rim_{matchcounter} \leftarrow rim_{matchcounter} + 1$
                    $break$
                end
            end
            $counter \leftarrow counter + 1;$
        end
        $pcr\_folded\_digest \leftarrow$
         $extend\_folding\_digest(pcr, eventrecords[i].eventhash);$
    end
    if $(counter == rim.size) \land (rim_{matchcounter} == rim.size)$ then
     $partial_{integrity} \leftarrow$true
    $rim_{match} \leftarrow$true

    ⋮
    /* The checks for the tpmquote, the signature and the
       nonce are not stated, since they are the same checks
       as in the original remote attestation verification
       procedure.                                           */
    ⋮
    if $all\ checks\ are\ valid$ then
        return $success;$
    else
        return $failed;$
    end
End Function
```

7.3.6 Implementation Versions

This work revealed how to realize a proof-of-concept from the elaborated theory in Chapter 6. Henceforth, the implementation is not enough, the interesting part is to evaluate the implemented concept as well. Therefore, this work implemented four versions to compare the measured performance results. Here, this thesis will introduce the four versions and their implementation differences.

SHA-256 Version The *SHA-256 version* is the version which this thesis introduced in the previous subsection, where it rehashes the *template hash* with the hash algorithm SHA-512. Afterwards, the rehashed hash will be reduced by the modulo operation (see Algorithms 7.1 and 7.2).

SHA-256 memcpy Version The *SHA-256 memcpy version* is nearly the same implementation as *SHA-256 Version*. However, the difference lies on line 4 in both Algorithms 7.1 and 7.2. Instead of rehashing the *template hash* with SHA-512, the *template hash* will be concatenated with itself. The line 4 will be replaced by:
$templatehash_{concatenated} \leftarrow (templatehash \| templatehash)$. The concatenation produces a 512-bit string which will be reduced by the modulo operation. This version exists to analyze the performance, whether the C implementation with the concatenation (in C the concatenation is represented as memcpy operations) or the *SHA-512* function is resource-efficient.

SHA-256_SHA-512 Version Next, the *SHA-256_SHA-512 version* mixes both SHA-256 and SHA-512 algorithms in the protocol. SHA-256 is used in the *PCR_extend* operation and the SHA-512 is used for the file hash and *template hash*. Figure 7.2b illustrated the simulated IMA measurement process, where it uses SHA-256 for the *template hash* instead in this version SHA-512 is applied. Therefore, the work generated the version to see whether it is resource-efficient without rehashing or concatenating the *template hash*, if the *template hash* is sampled as SHA-512. However, the hardware TPM supports the hashing algorithms from SHA-1 to SHA-256. Thus, the *template hash* was reduced in Algorithm 7.1 to 32 bytes and the outcome was extended into the *TPM*.

SHA-512 Version Last, the *SHA-512 version* was implemented for the Docker setup environment (see Chapter 8), since the hardware TPM 2.0 does not support SHA-512 algorithm. In the whole project, where a hash algorithm is applied, it was replaced by SHA-512 algorithm. Furthermore, this version does not need any extra instructions for rehashing/concatenating the *template hash*, since it is computed in SHA-512. The only hurdle which has to overcome was that the *PCR_extend* accepts only data with the size of 64 bytes. So, the *event hash* has the size of 32 bytes, and it was necessary to concatenate the *event hash* for the *PCR_extend* operation ($eventhash \| eventhash$).

In the end, this section provided, how the theory was successfully mapped into a practical proof-of-concept. Still, this work had to overcome some hurdles. Moreover, it provided different variants of the protocol implementation to compare them each other in Chapter 8.

CHAPTER 8

Evaluation

This chapter presents the evaluation of the *Privacy-Preserving Remote Attestation (PPRA) protocol* and its Proof-of-Concept (PoC). Moreover, it analyzes based on the defined requirements in Chapter 5 whether the PPRA approach fulfills the security, privacy and functional requirements. Therefore, the chapter is separated into two sections. First, it evaluates and discusses the security and privacy requirements. Following, the functional requirements will be analyzed based on the approach. Afterwards, the performance results of the implemented PoC will be discussed in correlation with the functional requirements. Further, the results will be analyzed to find out if the PoC provides some limitations and benefits.

8.1 Security and Privacy Requirements

In Chapter 5 this work defined security and privacy requirements to establish the PPRA approach in Chapter 6. Further these requirements were derived from the four use cases in Chapter 4 and the threat model in Section 6.1. This section explains whether the requirements from Tables 5.2 and 5.3 were met. Besides, it analyzes the achieved security and privacy requirements in the PPRA approach.

Chapter 3 elaborated the related concepts on how to preserve privacy in remote attestation. Luo et al. [43] realized the privacy property while relaying on the *XOR* operation. However, the *PCR.secret* needs to be sent over the channel, where the verifier can recompute the hash with the secret (see Chapter 3). Further, Jie et al. [36] has a similar approach: instead of *XOR'ing* they are concatenating the *template hash* with a random factor otherwise known as *shielded factor*. The *shielded factor* must be sent through the channel to the verifier. In both methods, the verifier needs to know the *secret/random value/salt/shielded factor* for the affiliated binaries of the verifier. However, these values are not unique anymore if one binary or software component is needed for the integrity check by multiple verifiers. Therefore, all the verifiers know the secret value of one particular software component, which is used by multiple verifiers. These approaches can be improved by adding extra communication overhead by using the Diffie-Hellman key exchange protocol between attester and verifier. Though the Diffie-Hellman key exchange protocol has too much communication overhead and the entry of one binary exists multiple times for each verifier with different shared keys. By applying the Diffie-Hellman key exchange protocol, it breaks the original design of the remote attestation since it is necessary to have multiple request-response communication instead of one. In addition, all the shared secret keys have to be saved in the embedded system device. This consumes too much space and is resource-inefficient. The symmetric primitives show limitations in remote attestation. Therefore, this work introduced a new variant of the privacy-preserving concept (see Chapter 6). This new variant applied asymmetric primitives of the signature scheme, which was introduced in this work as the PPRA protocol. Hence, this thesis took advantage of the privacy-enhancing technology, the zero-knowledge proofs (P_1). To preserve privacy the Schnorr Non-Interactive Zero-Knowledge (NIZK) proof over an elliptic curve was used in PPRA protocol. However, the NIZK cannot be directly applied in the PPRA approach. The reason is that this work made adaptions on the NIZK proof. The NIZK proof over an elliptic curve in [33] uses the basepoint of an elliptic curve as a generator g (see Section 2.7). The PPRA approach defines a “new generator” g_i by computing: $g_i := g^{\varphi(h_T(x_i))}$ (the scalars and function in the computation are explained in detail in Section 6.3). As described in Section 2.7 a cyclic group has a generator, which generates all the elements in the group. Therefore, the generator g_i is an element generated by g. Thus, g_i is a generator as well due to the definition of a cyclic group, because g_i^y (where y is an exemplary value) dissolved is:

$$g^{\varphi(h_T(x_i))^y} = g^{(\varphi(h_T(x_i)))\cdot y} \tag{8.1}$$

The resolution in Equation (8.1) shows that using g_i produces outcomes which are still elements indirectly computed by g (where g is the basepoint in an elliptic curve). The reason for applying the function $h_T(x_i)$ (which represents the *template hash*) for g_i as the “new generator” is that the “proof of knowledge” can be validated over the affiliated *template hash* (see Figure 6.4) (P_2). Next, the *event hash* is the blinded *template hash*. The computation of *event hash* shows that the *template hash* is multiplied by a randomly computed scalar r, where r is the private key. Afterwards, the group operation (scalar multiplication) is applied by using the generator (for detailed explanation see Section 6.3). Then the *event hash* is used as the public key, because the affiliated *event hashes* will be sent to the verifier to validate the “proof of knowledge” (detailed explanation of the validation in Section 6.3). Based on the *event hash*, privacy is assured, because from the *event hash* it is not possible to infer the *template hash*. The reason is, the representation of the generator g^x is the point multiplication (scalar multiplication), where the point is the basepoint of the elliptic curve and x is a scalar. Therefore, the computation of the *event hash* shows that r cannot be retrieved, because the inverse function of the point multiplication cannot be applied. Hence, the point multiplication provides the security based on Elliptic Curve Discrete Logarithm Problem (ECDLP) (see Section 2.8). Further, all the adaptions of the NIZK proof are illustrated step by step in Figure 6.4.

Hence, to get a clear understanding of the changes: compare Figures 2.3 and 6.4, to see the adaptions based on the properties of the NIZK. This work shows the correctness of the NIZK proof is still valid while applying the changes (P_1):

$$\begin{aligned}
t'_i &= g_i^{s_i} \cdot h_{eventhash}(r_i, x_i)^{c_i} \\
&= g^{\varphi(h_T(x_i))\cdot(v_i - c_i \cdot r_i)} \cdot g^{r_i \cdot \varphi(h_T(x_i))\cdot c_i} \\
&= g^{\varphi(h_T(x_i))\cdot v_i - \varphi(h_T(x_i))\cdot c_i \cdot r_i} \cdot g^{r_i \cdot \varphi(h_T(x_i))\cdot c_i} \\
&= g^{\varphi(h_T(x_i))\cdot v_i - \varphi(h_T(x_i))\cdot c_i \cdot r_i + r_i \cdot \varphi(h_T(x_i))\cdot c_i} \\
&= g^{\varphi(h_T(x_i))\cdot v_i} \\
&= g_i^{v_i} \\
&= t_i
\end{aligned}$$

Besides, the NIZK proof is generated over an elliptic curve. Therefore, this thesis did an elliptic curve analysis to find a suitable curve for the PPRA approach (see Section 7.1). Based on the elliptic curve analysis in Section 7.1, this work uses the *Ed25519* which is birationally equivalent to *Curve25519* [4]. However, the curve *Ed25519* is not directly used, due to the complication of the cofactor. Hence, this work applies the Ristretto technique based on the *Ed25519* (Ristretto eliminates the cofactor without sacrificing security and performance, see Section 7.1) (S_6). While applying the Ristretto technique during the implementation of Algorithms 7.1 and 7.2, the *modulo operation* was used to compute uniform values in the range of the prime order group. Therefore, this work showed with statistical distance in Section 7.3.3, if an unbounded and powerful adversary can distinguish between the input and the computed output of the *modulo operation* (see Section 7.3.3). This outcome was achieved if a 512-bit string was used as an input, then the statistical distance is negligible (close to zero). This was done to provide an argument for using the *modulo operation* in NIZK implementation and showed that an adversary is not capable of distinguishing between the input and the outcome of the modulo operation.

After the detour in elliptic curve analysis, this section continues to explain how the NIZK is used to preserve privacy by enforcing the verifier constraints. By using the constraints, the attester only reveals the associated entries of the Stored Measurement Log (SML) to the verifier (P_2). From there on the verifier receives *all event hashes* and the associated subset of the scalars c, s (extra data) of the NIZK proof generation (see Section 6.3). Next, the verifier applies the mathematical operation to validate the *event hash* with "proof of knowledge", if the *event hash* is the corresponding *template hash* (note: The validation only works with scalars c and s to assure the zero-knowledge property. Further, the verifier receives to the associated software components c and s.). Hence, the verifier cannot use the scalars c and s for non affiliated *event hashes*. Due to the NIZK property of *soundness*, which defines that the attester is able to convince the verifier that the statement ("proof of knowledge") is incorrect. Hence, if the statement is incorrect for the associated information of the verifier, then the software component on attester's system is corrupted or jeopardized. The *correctness* of NIZK is given, if the attester is able to convince the verifier that the statement is correct without revealing r (private key). Hence, the properties of NIZK embellishes, why this work used NIZK for the PPRA approach. Further, if multiple verifiers validate the same software component, neither of them knows the private key. However, if validation is successful, then they are convinced that the *template hash* is the correct *event hash (blinded template hash)* and this implies that the software component's integrity is intact.

As mentioned above, between attester and verifier a communication channel is established, to transfer the attestation information (i. e. *event hashes*, c, s, TPM Quote). Therefore, the communication channel needs to be encrypted (S_3). This is realized by using TLS and the PoC utilizes the library *mbedTLS* for it (see Chapter 7). The TLS connection is used to prevent *passive man-in-the-middle* attack (see Section 6.1). Further, both parties need to authenticate each other. This was realized by using functions of the *mbedTLS*, which authenticates both parties certificates. This was already stated in Section 7.3.5 how the mutual authentication was done in PoC (S_4). In the PoC the certificates were self-signed and self-issued. In the case, if the PPRA approach is integrated into a system, a Public Key Infrastructure (PKI) is necessary to issue and revoke certificates. The PKI is not a mandatory component in the PPRA approach, because the concept focuses on preserving the privacy of the operational state of the attester's system.

Besides, the Schnorr NIZK proof assures privacy and the successful validation of "proof of knowledge" implicates the entry integrity in the SML. However, it does not guarantee the entire integrity of the operational state of the attester's system. Therefore, the PPRA concept utilizes the Trusted Platform Module (TPM) to use operations which maintain the chain of trust and preserves the integrity of the entire operational state of the attester's system. Each computed *event hash* of the corresponding software component will be anchored in the secure volatile storage of TPM by using the *Platform Configuration Register (PCR) Extend* operation (see Sections 6.3 and 7.3.4). Furthermore, the extend operation extends the chain of trust by accumulating each *event hash* and each *event hash* is logged in the SML (S_1). The SML represents the operational state of the attester's system. Hence, the integrity is verified over the SML by matching the actual value of the *TPM Quote*. Furthermore, the adversary cannot manipulate the measurements after the measured boot sequence, because the TPM is hardened against physical attacks. Therefore, the adversary cannot retrieve any secretes, nor manipulate *event hashes*. Moreover, to provide the authenticity of the operational state, the operation *TPM Quote* is used, where the TPM signs with the private part of the Attestation Key (AK) the operational state of the attester's system. Nonetheless, the verifier has the public part of AK to verify the authenticity/signature of the signed operational state (S_2). To provide integrity and authenticity of the operational state of the attester's system, the PPRA approach is dependent on the security properties of the TPM.

Overall, in the paragraphs above, this work discussed and argued how security and privacy are achieved. However, the trustworthiness of the attester's system is assured if all the steps in the verification process are successful (see Figure 6.2). If one of the steps fails, then the attester's system is jeopardized and not trustworthy. The verification of the trustworthiness of the attestation information is shown in the Chapter 6, where the PPRA approach is introduced (S_5).

In addition, it is assumed in a scenario where multiple verifiers want to verify the operational state of the attester's system that the verifiers do not communicate with each other, nor they know each other. The PPRA approach does not provide a possibility for the verifier to communicate with other verifiers because the verifier can only establish a connection to the attester (P_3).

8.2 Functional Requirements

This section explains, based on the theoretical approach (PPRA approach) and the implemented PoC, how all the functional requirements are met. Therefore, the explanation is separated into four parts: First, the achievements of the functional requirements are elaborated based on the PPRA approach. The second part embellishes the achieved functional requirements based on the performance evaluation. Third, the communication costs are evaluated. Last, a brief description is given of how the PPRA approach can be integrated into the use cases from Chapter 4.

8.2.1 PPRA Approach

The PPRA approach uses the Linux Integrity Measurement Architecture (IMA), which is a concept of the Trusted Computing technology. In the case of this work, the tasks of the IMA are simulated since it was not the scope of this work to mitigate with the actual Linux IMA (see Section 7.3.4). Hence, the PoC implemented the tasks of the Linux IMA (see Algorithm 7.3). The IMA is a trusted service, which measures the software components which are loaded into the memory of the attester's system. The PPRA concept relies during the measurement process on the IMA to compute the *template hashes* (F_1) (see Figure 6.1 (step 1)). Next, the core idea of the PPRA approach is to preserve privacy. Therefore, the computed measurements (*template hash*) of the IMA needs to be blinded. To blind the measurements, the PPRA approach takes advantage of the properties of the NIZK proof. Nonetheless, the PPRA approach integrates the method of the NIZK to fulfill the functional requirement F_2 (detail explanation of the blinding operation in Section 6.3). Moreover, the blinded measurements (*event hash*) need to be logged to keep the trace of all loaded software components. Hence, the blinded measurements are persisted in a log file (i. e. SML) (F_3). Further, the PPRA approach in Chapter 6 described that the computation outputs of the NIZK proof and the *event hash* are stored in the SML (F_3) (see Figure 6.2 step 3, step 5). Further, to preserve the integrity of the operational state of the attester's system, the TPM is utilized. The *event hash* is anchored in the TPM by triggering the TPM operation *PCR Extend* (F_4). The *event hash* needs to be anchored to extend the chain of trust in the attester's system and to keep the operational state "up to date". In the PoC the *tpm2-tss* library is used to access the functionalities of the TPM (F_4). The next functional requirement F_5 was fulfilled, by designing the

PPRA approach, while considering the necessary functions for remote attestation (see Section 6.4). Further, the PoC implemented functions for triggering a remote attestation, preparing the attestation information (on attester side) and verifying the attestation information (on verifier side) (see Section 7.3.5,Algorithm 7.4). Nonetheless, the verifier and the attester establishes communication during the remote attestation. Therefore, a communication structure needs to be defined so both parties can understand the transferred data during the remote attestation. Hence, this work utilizes *QCBOR* library to define a Concise Data Definition Language (CDDL), which both parties use to decode or encode the information (see Section 7.3.5) (F_6). Figure 6.2 illustrates that the PPRA approach uses only one round of the request-response scheme (F_7). Further, to preserve privacy, the attester limits the exposure by sending the associated subset of measurements to the verifier (F_8). This was realized by the PoC by a policy check. The policy in the PoC is a file which contains the associated software components of the verifier. This functional requirement F_8 is fulfilled at the time of the *remote attestation process* (Section 6.4). During the remote attestation, the attester sends an attestation response. The response has to be verified. While verifying the attestation response, each step of the verification process has to be successful (see Figure 6.2 (step 12-16)). Otherwise, if one of the steps fails, then the trustworthiness is not given, and the attester's system is either corrupted or jeopardized (see Section 6.4 and Algorithm 7.4) (F_9). Algorithm 7.4 in the PoC illustrates the implementation of all verification steps. The stated description above for each functional requirement shows that each of them was fulfilled while designing the PPRA concept. Moreover, the PoC is the implementation of the PPRA approach while fulfilling all the functional requirements.

8.2.2 Performance Evaluation

This subsection completes a performance analysis based on the functional requirements. The performance analysis illustrates the resource utilization of the PoC based on the functional requirements. The performance evaluation is separated into multiple parts. First, it describes the experiment setup, where the PoC will be tested and evaluated. Next, the implementation for the performance evaluation will be elaborated. The explanation of the performance evaluation implementation exists to reproduce the performance measurements. The following part evaluates the NIZK (Schnorr Signature) implementation. The implementation provides two functions, where the first function computes the signature and the second function verifies the computed signature (see Section 7.3.3). The CPU cycles and execution time of both functions will be measured. However, the measurements of NIZK will be compared against the different versions of the PoC implementation. The differences between the versions were elaborated in detail in Chapter 7. Based on the NIZK implementation the functional requirements F_{10} and $F_{10.1}$ are evaluated. The last part of the performance evaluation focuses on the performance measurement of the *privacy-preserving remote attestation protocol*. The performance is measured in ms (execution time) (F_{10}, $F_{10.2}$). The core concept of this work is to combine the traditional binary attestation with the privacy-enhancing technology (non-interactive zero-knowledge proofs) while preserving privacy and gain advantages. Therefore, the processing time/execution time of a whole attestation process will be measured. Next, the processing time of the attester and the verifier will be individually assessed. Afterwards, the individually measured values will be added together and then the result will be reduced from the execution time of the whole attestation process to calculate the communication time. These measurements represent the resource consumption of the PoC on restricted devices (F_{10}). Therefore, this work uses two hardware devices (presented in the upcoming subsections). Besides, each part of the evaluation process will be executed on different test setups and different devices.

Experiment Setup

The experiment setup is known as the test environment for the PPRA protocol and the NIZK functions. The experiment setup is used for measuring the CPU cycles and the execution time. Subsequently, the results of the evaluation of the test environment are recorded and compared with each other.

This work provides four versions of the PPRA protocol. The four versions were introduced in Chapter 7. The reason for the existence of four versions is, because of the limitation of the hardware TPM. The Infineon Optiga™ SLB 9670 TPM 2.0 which is attached on the Raspberry Pi 3 and does only support the hashing algorithm from SHA-1 to SHA-256 [59]. In Chapter 7 this work mentioned that due to the limitation of the hardware the *template hash* needs to be rehashed with SHA-512 or concatenated to generate a uniform value in the range of the prime group order of the Ed25519 (prime group oder is the same as in Curve25519). This version is called SHA-256 and where the concatenation approach is used, is known as SHA-256 memcpy see Table 8.1.
Next, the version SHA-256_SHA-512 uses the TPM 2.0 hardware attached on the Raspberry Pi 3. In this version the *PCR_extend* function of the TPM uses SHA-256 algorithm and the rest of the hashes such as the file hash and *template hash* use SHA-512. The last version is SHA-512, which uses SHA-512 algorithm everywhere, but it has a limitation in *PCR_Extend* since the event hash does only provide the size of 32 bytes due to the curve size of Ed25519. Hence, the *PCR_Extend* in the SHA-512 version needs data with the size of 64 bytes. The option is to either concatenate the event hash twice to fill the 512 bits or fill the last half of the bits with zeros. Otherwise, the PPRA protocol will fail in the verification process, since if the rest of the bits are not sampled it will be randomly filled.

Besides, the four implementation versions exist to see how the resource consumption affects different systems/architectures (F_{10}). These four versions are listed in Table 8.1. Moreover, Table 8.1 illustrates in each row the experiment setup. The column *Connection* represents if the connection is established locally or remotely between attester and verifier. Local connection means the device has both roles of the attester and the verifier. Remote means the role of attester and verifier are on different devices (see Table 8.1). Furthermore, the *TPM* columns illustrate if the test environment uses a hardware TPM or a software TPM. *swTPM2.0* is a software TPM, which emulates all the functions of a real TPM and even provides more hashing algorithms such as SHA-512. The software

Attester	Verifier	TPM	Connection	Version
Docker	Docker	swTPM2.0	Remote	SHA-256
Docker	Docker	swTPM2.0	Remote	SHA-256 memcpy
Docker	Docker	swTPM2.0	Remote	SHA-512
Docker	Docker	swTPM2.0	Remote	SHA-256_SHA-512
RPI3	RPI3	Infineon SLB9670 TPM2.0	Local	SHA-256
RPI3	Lenovo W540	Infineon SLB9670 TPM2.0	Remote	SHA-256
RPI3	RPI3	Infineon SLB9670 TPM2.0	Local	SHA-256 memcpy
RPI3	Lenovo W540	Infineon SLB9670 TPM2.0	Remote	SHA-256 memcpy
RPI3	RPI3	Infineon SLB9670 TPM2.0	Local	SHA-256_SHA-512
RPI3	Lenovo W540	Infineon SLB9670 TPM2.0	Remote	SHA-256_SHA-512

Table 8.1: Evaluation environments.

Device	Arch	CPU	RAM
Raspberry Pi 3 Model B V1.2	arm32	4 x ARM Cortex-A53 @1.2 GHz	1 GB
Lenovo W540	x86	4 x Intel i7-4700MQ CPU @ 2.40GHz	16 GB

Table 8.2: Devices used for the evaluation.

TPM provides the capability of integrating it into a virtualized environment such as Docker [2]. *Infineon Optiga™ SLB 9670 TPM 2.0* is a hardware TPM, which supports the hashing algorithms SHA-1 to SHA-256 [59]. Next, the *attester and verifier* columns define which device takes the role of the attester and verifier. The Docker container will be two different instances, and the connection between them is established through a Docker instanced network.

Devices The previous subsection explained the experiment setup and listed devices that are used for having the role of the attester or verifier. This subsection lists all the devices used in this work to evaluate each part of the performance evaluation (see Section 8.2.2). For this work, a Raspberry Pi 3 Model B V1.2 (RPI3) is used to emulate an embedded system. The major role of the RPI3 is to be the attester. The RPI3 needs a hardware TPM, therefore this work uses the Infineon Optiga™ SLB 9670 TPM 2.0 which is attached on the GPIO ports of the RPI3. Further, the RPI3 will be also playing the verifier for the local test environment. Moreover, the Lenovo ThinkPad W540 has the role of the verifier, for remote-based testing. The RPI3 is running a Raspberry Pi OS Lite as operating system and the Lenovo ThinkPad W540 is running Arch Linux. Further, on the Lenovo ThinkPad W540 the Docker environment is configured for the Docker-based test environment. Table 8.2 illustrates the detailed information about these devices to retrace the experiments and their results.

CPU Cycle & Execution Time Measurement Implementation

This subsection continues with the explicit implementation for measuring the CPU cycles and execution time. Further, it will explain the code snippets which were used to retrieve the measurements. The reason for the detailed explanation of the implementation is to provide reproducibility of the performance measurement results.

CPU Cycle For the Intel processor, the rdtsc and rtscp instructions were used to get the CPU cycles of a function. The rdtsc and rtcsp stand for read time stamp counter and processor [51]. The Listing 8.1 illustrates a few assembler instructions in the C function for measuring the CPU cycle. The *rdtsc* instruction loads the high order bits in to *edx* and the low order into *eax* [51]. The assembler instructions will be executed on a 64-bit Intel architecture. Therefore, the *rdtsc* will load the high order bits into *rdx* and the low order bits into *rax*. The instruction is used to read the time stamp from the register. Next, Listing 8.1 on line 4 shows an invocation of the CPUID instruction, which is necessary as a protection against the out-of-order execution [51]. Therefore, the CPUID instruction will be invoked once before the *rdtsc* instruction (Listing 8.1) and once after the *rdtscp* instruction (Listing 8.2).

```
1   static inline uint64_t cpu_clock_ticks_start(){
2     uint32_t lo, hi;
3     __asm__ __volatile__("CPUID\n\t"
4     "RDTSC\n\t"
5     "mov %%edx, %0\n\t"
6     "mov %%eax, %1\n\t"
7     : "=r"(hi), "=r"(lo)::"%rax", "%rbx", "%rcx", "%rdx");
8     return (uint64_t)hi << 32 | lo;
9   }
```

Listing 8.1: CPU Cycle count start for Intel Arch [15, 51].

Listing 8.2 is nearly the same as Listing 8.1. The difference lies in the different instruction invocation, *rdtscp*. The *rdtscp* reads the timestamp from the register for the second time. Further, the *rdtsc* instruction guarantees that all the C code (instructions) which need to be measured will be called, before the instruction itself will be executed [51].

```
1   static inline uint64_t cpu_clock_ticks_end(){
2     uint32_t lo, hi;
3     __asm__ __volatile__("RDTSCP\n\t"
4     "mov %%edx, %0\n\t"
5     "mov %%eax, %1\n\t"
6     "CPUID\n\t"
7     : "=r"(hi), "=r"(lo)::"%rax", "%rbx", "%rcx", "%rdx");
8     return (uint64_t)hi << 32 | lo;
9   }
```

Listing 8.2: CPU Cycle count end for Intel Arch [15, 51].

Listing 8.1 and Listing 8.2 bit shifts and bitwise or the high order value, and then it will be stored into a local variable. Listing 8.3 illustrates a *define* operation where the differences between the values from Listing 8.1 and Listing 8.2 are calculated. Further, the *block input* is the C code block, which needs to be measured.

```c
#define CYCLES_DURING(result, block)
do {
  uint64_t __begin = cpu_clock_ticks_start();
  do block while(0);
  *result = cpu_clock_ticks_end() - __begin;
} while(0);
```

Listing 8.3: Calculating the CPU Cycle of a function/block [15].

On the one hand, the Intel processor needs the CPUID in the assembler code, due to the reason of the out-of-order corruption. On the other hand, the ARM processor does not need it, but it needs an extra kernel module to enable the CPU counter for the user space. By default, the user space C code does not have access to the CPU Counter. Therefore, a kernel module was written and loaded into the system to measure the CPU cycle of the function on the RPI3. Listing 8.4 is an example code for enabling the cycle counter of the ARM architecture on a RPI3 [24, 44]. The function *enable_ccr* enables the cycle counter. Next, on each CPU, the function cycle counter register will be enabled. Then the module will be compiled as a kernel module and loaded into the system.

```c
#include <linux/module.h>
#include <linux/kernel.h>

void enable_ccr(void *info) {
  // Set the User Enable register, bit 0
  asm volatile ("mcr p15, 0, %0, c9, c14, 0" :: "r" (1));
  // Enable all counter
  asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r"(0x80000000)
      );
}

int init_module(void) {
  // Each cpu has its own set of registers
  on_each_cpu(enable_ccr,NULL,0);
  printk (KERN_INFO "Userspace access to CCR enabled\n");
  return 0;
}
```

Listing 8.4: ARM Kernel Module for CPU Counter enable [24, 44].

After loading the module into the system, the assembler code needs to be adapted since it is a different CPU architecture than the Intel processor. The instruction *mrc* is used to get the cycle counter. Next, in Listing 8.3 the methods were swapped out with the ARM function (see Listing 8.5).

```
static inline unsigned long cpu_clock_ticks_rasp(){
  uint32_t val;
  __asm__ __volatile__("mrc   p15, 0, %0, c15, c13, 0" : "=r"(val))
     ;
  return val;
}
```

Listing 8.5: ARM CPU Cycle Measurement [24, 44].

This approach was used to get the CPU cycle measurement results of both CPU architectures for the signing and verification process.

Execution Time The execution time was measured with *time.h* library, which is a standard library in C. The measurement results of the execution time returns the time of the wall clock. Since this work wants to know how long it takes for the whole attestation process. This block of code can be used everywhere in the project from measuring specific functions up to measuring the whole attestation process. Listing 8.6 visualizes that a start and end timer needs to be defined. After executing the code the difference between the end and start timer is calculated to get elapsed time in milliseconds.

```
// Start measuring time
struct timespec begin, end;
clock_gettime(CLOCK_REALTIME, &begin);

/*Function or Code Block to be measured*/

// Stop measuring time and calculate the elapsed time
clock_gettime(CLOCK_REALTIME, &end);
long seconds = end.tv_sec - begin.tv_sec;
long nanoseconds = end.tv_nsec -begin.tv_nsec;
//returns time in ms
double elapsed = seconds*1000+nanoseconds/1e6;
```

Listing 8.6: Measuring Execution time [15].

Performance Measurements of NIZK Implementation

The following section illustrates the measurements for the signing and verification process of the non-interactive zero-knowledge proof. Both processes were measured on two different CPU architectures as described in Section 8.2.2. Next, each architecture measurement will be described individually and then compared with each other. One measurement unit is the CPU cycles and the other one is the execution time in ns or ms. Here, the functional requirements F_3 is measured how long it takes to blind the measurement (NIZK signing operation) and how many CPU cycles it needs on two different devices. Further, the NIZK signing and verification are considered as cryptographic operation. Therefore, this subsection executes the operation on multiple

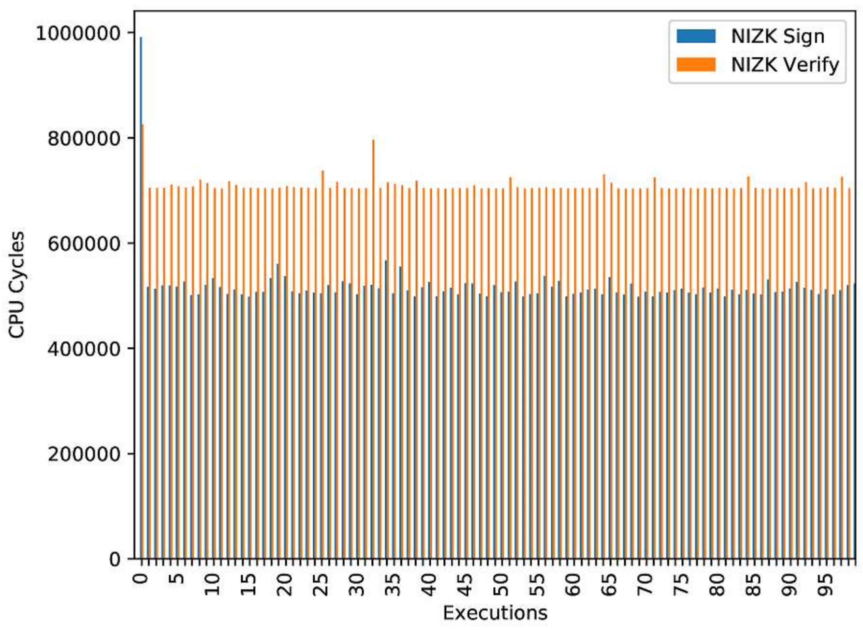

Figure 8.1: CPU cycle measurement of the NIZK signing and verifying process (Intel).

measurements to see, whether resource consumption is nearly the same. Moreover, the measurement is done on different CPU architectures, where the ARM architecture represents resource constraint devices (F_{10}). In the end of this subsection the outcome of the results are discussed.

Measurements on Intel CPU First, the measurements were taken from the CPU cycle on the Intel processor to see the impact of the NIZK signing and verification process. Figure 8.1 visualizes the measurement of the CPU cycles, which defines how many instructions are necessary for the signing and verification process. Next, it is visible for the first binary it needs more instruction, due to the reason of the initialization of the Libsodium library. The initialization process increases the CPU cycles. As the developer of the Libsodium library mentioned the init process of the library can lead to staging on the Linux systems [23]. Further, it is visible that the verifying process needs more instructions than the signing process. This induces that the verification of a signature needs more group operations on the elliptic curve than the signing process, despite the signing operation needs more scalar arithmetic. For scalar arithmetic, there does not exist extra cost for CPU cycles as illustrated in Figure 8.1. The bar chart tries to level off, but for some binaries, it stands out, however not extremely as in the beginning. Following, results of the execution time measurements are presented. Figure 8.2 represents for 100 different binaries how long each of them took for the signing and verification the process of the NIZK. Moreover, Figure 8.2 shows, that only the signing process needs more

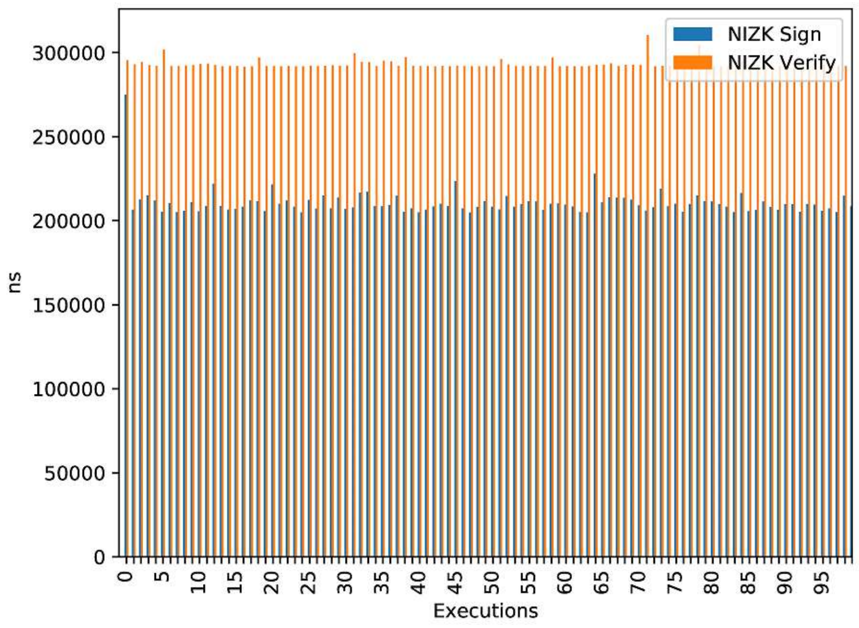

Figure 8.2: Execution time measurement of the NIZK signing and verifying process (Intel).

execution time at the beginning. Nonetheless, the verification process does have a peak at the beginning. The peak at the beginning indicates the initialization process of the Libsodium library. Afterward, the measurement results of the different binary executions are nearly the same.

Measurements on ARM Next, this paragraph analyzes how long the NIZK takes for the signing and verification on the ARM architecture. Further, measurements were taken from the CPU cycle on the ARM processor to see the impact of the NIZK signing and verification. Figure 8.3 illustrates the behavior of the NIZK implementation on the ARM architecture. The ARM processor needs more instructions than the Intel processor and has the same peak at the beginning, where it states to initialize the Libsodium library. Next, the execution time of the NIZK signing and verification process are measured in nanoseconds. It is clearly visible in Figure 8.4 it needs for the first seven binaries more execution time, and then it evens out.

Comparisons of Architectures and Versions This paragraph continues with the comparison between the architectures and versions of the NIZK implementation. Figure 8.5 plots a bar chart which illustrates the CPU cycle average measurement of 250 executions of each version and architecture. This chart should give a clear understanding, how much difference exists between the different versions. The previous charts illustrated

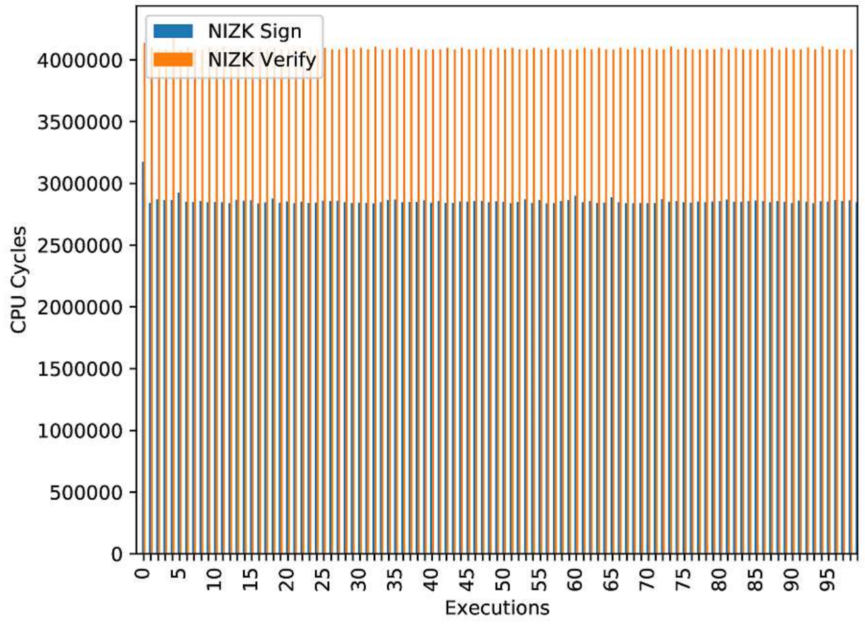

Figure 8.3: CPU cycle measurement of the NIZK signing and verifying process (ARM).

the differences based on CPU cycle measurements. Furthermore, the following Figure 8.6 visualizes the differences based on the execution time. Figure 8.6 visualizes the average execution time of the different versions and architecture. The y-axis illustrates the time of how long it took for each version. The x-axis represents the different versions of the NIZK implementation.

In the previous paragraphs the performance of the NIZK signing and verification were measured. The performance measurements illustrate the CPU cycles and execution time for a specific number of executions with different measurements (binaries) ($F_{10.1}$). While measuring the performance on the Intel architecture, the *Enhanced Intel SpeedStep® Technology and Hyper-Threading Technology* were disabled to give the ARM architecture a fair competition. By disabling these features of the Intel processor, it guarantees it will not influence the measurements. Figures A.1, A.5 and 8.1 represent the CPU cycle measurement and each of them show in the first NIZK signing execution a peak on the Intel processor. Figures A.3, A.7 and 8.3 of the ARM architecture mimic the same peak from the Intel measurements for the first execution in the signing process. However, on the Intel architecture it is clearly visible the resource consumption on various measurement is different in the signing process. The variation between each execution is not enormous, but on Intel architecture it is not constant. The ARM architecture showed promising results on each implementation version with different binaries (*template hash*). Each of the operation on the *template hash* presents each execution utilizes the same amount of the CPU cycle ($F_{10.1}$). The first peak indicates that the Libsodium library is

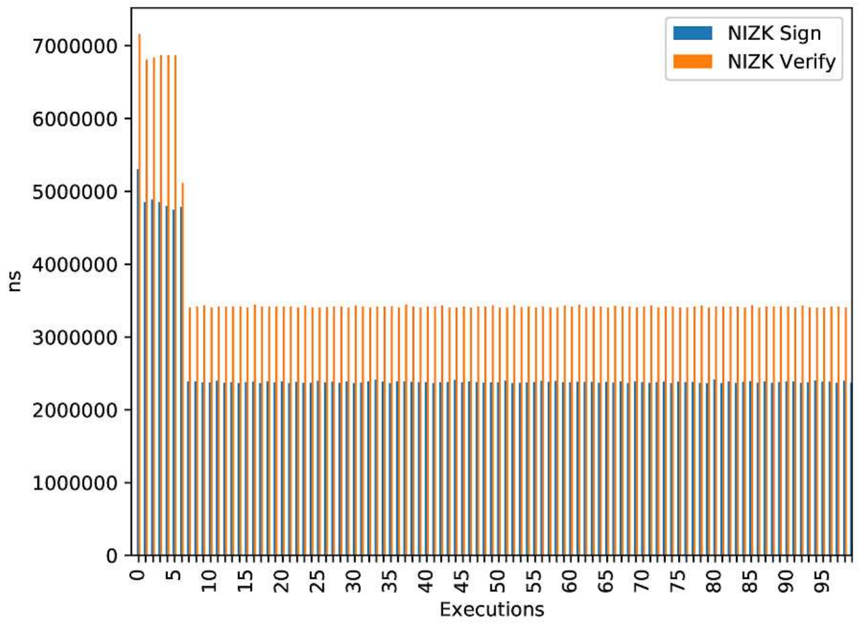

Figure 8.4: Execution time measurement of the NIZK signing and verifying process (ARM).

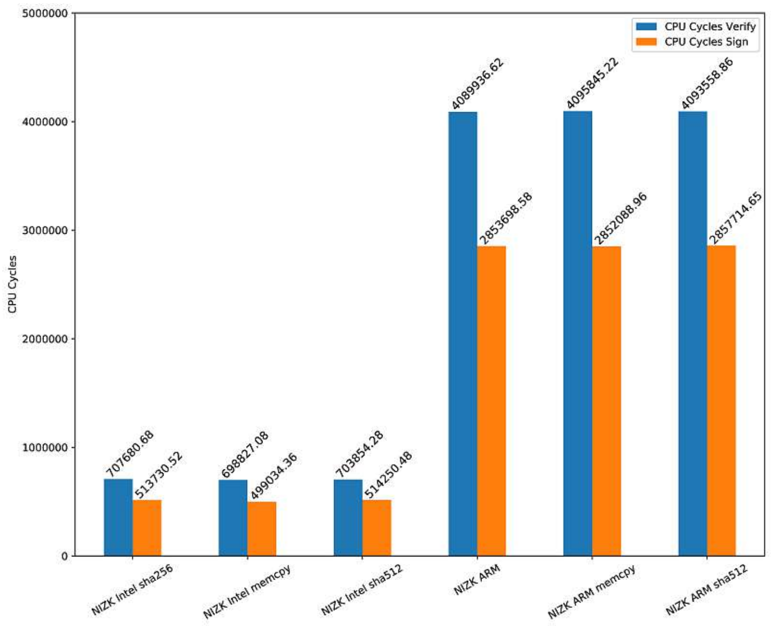

Figure 8.5: CPU cycle average measurement of each architecture and version.

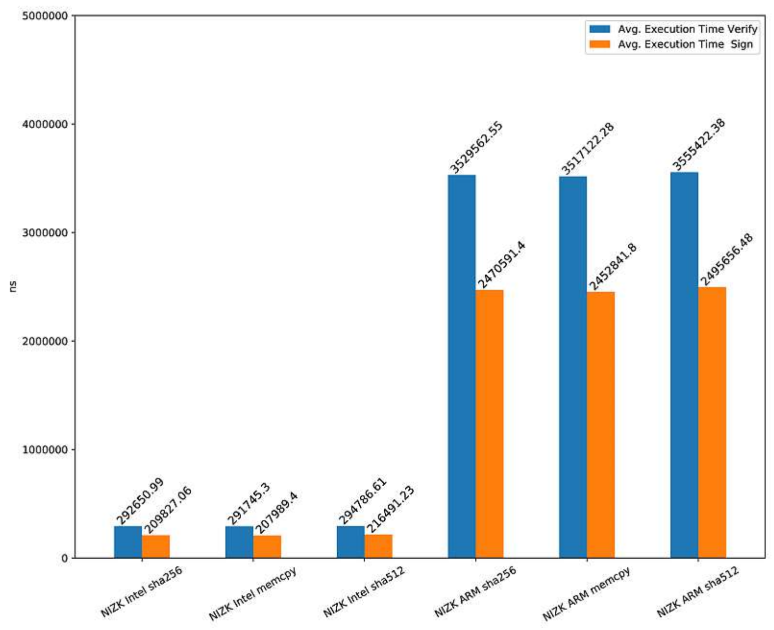

Figure 8.6: Average execution time of each architecture and version.

initialized. Therefore, more instructions are necessary for the first execution. Although, Figures A.1, A.3, A.5, A.7, 8.1 and 8.3 represent three different implementations of the NIZK on Intel and on ARM. These representations were made to see if one of the implementation versions could be used in a restricted device to reduce the resource consumption. However, on the ARM architecture there does not exist much difference, illustrated in Figures A.9 and 8.5. Based on these figures, the implementation differences does not affect the CPU Cycle on both architectures (F_{10}, $F_{10.1}$). Although, the version *SHA-256* and *SHA-512* need more instructions than the version *SHA-256 memcpy* on Intel (Figure 8.5). Due to the concatenation the version *SHA-256 memcpy* needs less instructions. In C it is represented as two memcpy instructions. On the other hand, the rehashing process in version *SHA-256* needs more instructions. However, these changes did not affect any of the measurements, because the results show no enormous differences. Besides, all the figures stated above visualize that the verification process needs more CPU cycles than the signing process. The reason is that the verification process needs more group operations than signing and this indicates more C instructions. Overall, the ARM architecture utilizes more CPU cycles than the Intel architecture. The reason is the performance differences of the used hardware (see Table 8.2).

In the following, this work discusses the measurement results of the execution time of the NIZK implementation. Figures A.2, A.6 and 8.2 represent similar outcome of the execution time. On the Intel processor, every figure shows at the first execution a higher peak than the following executions. This can be an indication of the Libsodium library initialization. However, the results of the ARM processor (Figures A.4, A.8 and 8.4) show a different behavior. The first five executions take more time. The reason is that the staging of the Libsodium library took longer than expected. In the end, the comparison between the architectures and versions of the execution time in Figures A.10 and 8.6 results as expected. The ARM architecture consumes more execution time due to hardware performance difference. The comparison between the versions of both architectures illustrate there does not exist an enormous difference. To conclude, the execution time and CPU cycles give an overview of the NIZK implementation and how it behaves on different architectures. Furthermore, the ARM architecture should represent the embedded system domain to visualize how much time and resources the current NIZK implementation consumes (F_{10}).

Performance Measurements of PPRA in a Hardware Environment

This section represents the measurement results of the PPRA protocol executed on the hardware-based setup environment (see Section 8.2.2). Further, measurements were recorded from the CHARRA project. A few modifications were made in the CHARRA project for the performance measurements. For instance, the key loading process was adapted as the same as in PPRA protocol. This subsection breaks down the measurement of the PPRA protocol in various paragraphs. The first paragraph presents the comparison between CHARRA and PPRA ($F_{10.2}$). The second paragraph presents the measurement results of the whole attestation procedure (F_5). The third paragraph will be providing the measurement results of the attester side (F_4, F_8) and the verification process on the verifier side (F_9). The last paragraph will present the measurement results of the communication process (F_6, F_7). All results are compared together to see the differences and limitations which will be analyzed and discussed at the end of this subsection .

Attestation Process The following subsection expresses the execution time of the request-response communication of the attestation process (F_5,F_7). The functional requirement F_5 is given, since the PoC implemented the functions for triggering the remote attestation and verifying the attestation response (see Section 7.3.5). The implementation of the PoC establishes only one round of communication. The both parties establishes a socket and the verifier sends the request and after receiving the response it closes the socket (F_7). Figure 8.7 compares the attestation process between CHARRA and PPRA. The difference between the two projects is clearly visible, because to achieve privacy it consumes more execution time ($F_{10.2}$). Figure 8.8 visualizes on the x-axis all different versions of the PPRA protocol implementation. Moreover, the x-axis defines four categories of the executions: 50, 100, Mixed, 200. The numbers represent the amount of the software selection for the attestation process and the *mixed* category defines that the software selection is not in order, which should simulate more execution time on the attester and the verifier side. The y-axis expresses the average execution time. Further, the legend in Figure 8.8 is separated in two categories first *local* and second *remote*. *Local* means the attester and verifier were on the same hardware device.

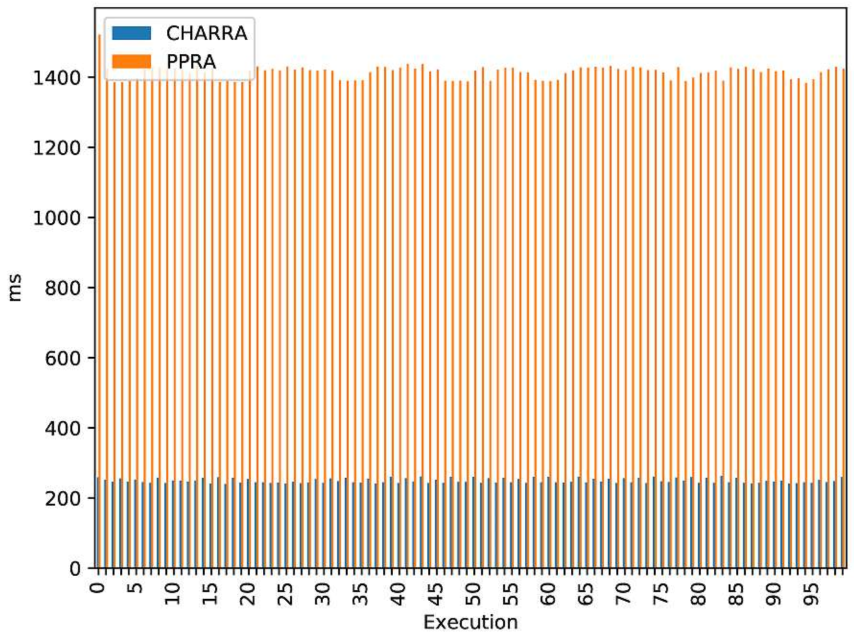

Figure 8.7: Comparison between CHARRA and PPRA on the attestation process.

In this case, it was a RPI3. *Remote* means the attester and verifier were on two different devices (see Table 8.1).

Attester This subsection speaks for the measurements of the attester side. Figure 8.9 shows the average execution time from the different implementation versions of the PPRA protocol. The legend, the x-axis, and the y-axis have the same definition as in Figure 8.8. As expected, the attester side has the same measurement results for the *local* and *remote* evaluation environment, because both uses the same attester device. Moreover, the measurement result includes the policy check, retrieving the *TPM QUOTE* and preparing the response data.

Verifier On the verifier side, the measurements were taken for generating the attestation request and for verifying the attestation response, such as the signature, TPM Quote, nonce, and the NIZK proof from the attester (F_9). The functional F_9 is fulfilled, since the PoC implements on the verifier side each verification step. Therefore, the execution time measurements were taken. The legend represents two categories. The first category *local* speaks for the local test environment and the second category *remote* for the remote connection between two different devices (see Table 8.2). The x-axis of Figure 8.10 lists all different versions with different amounts of software selections to see how it affects the hardware setup. Further, the average time is calculated over 100 iterations of each version. Figure 8.11 illustrates the average time for generating the attestation request.

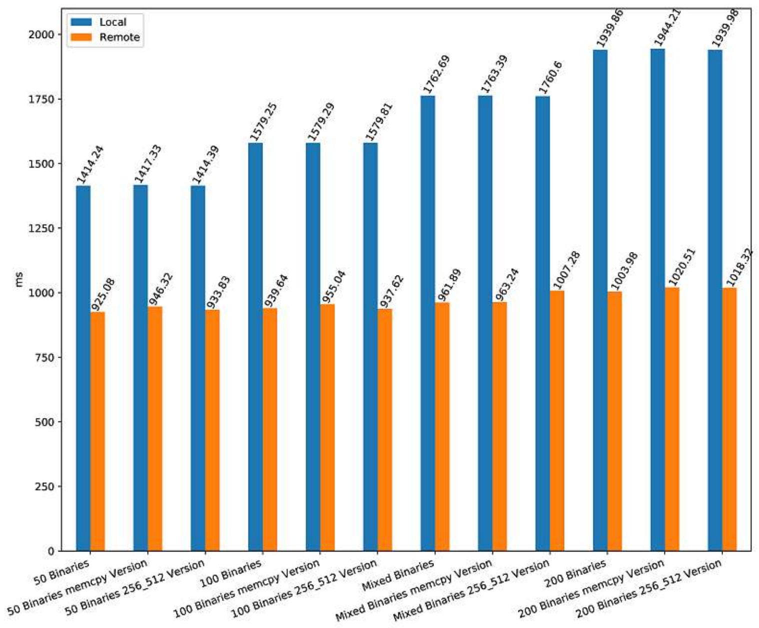

Figure 8.8: Comparison between the PPRA versions of the attestation process.

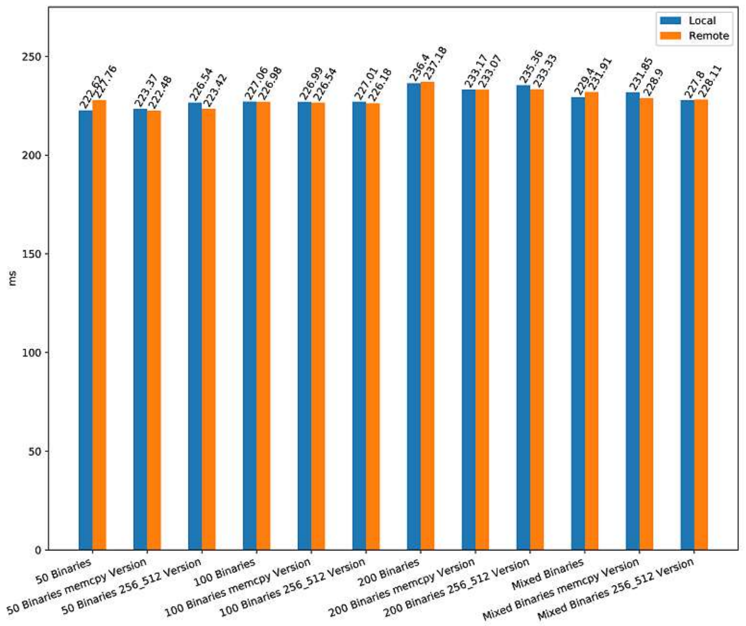

Figure 8.9: Comparing average execution time of different versions on attester side.

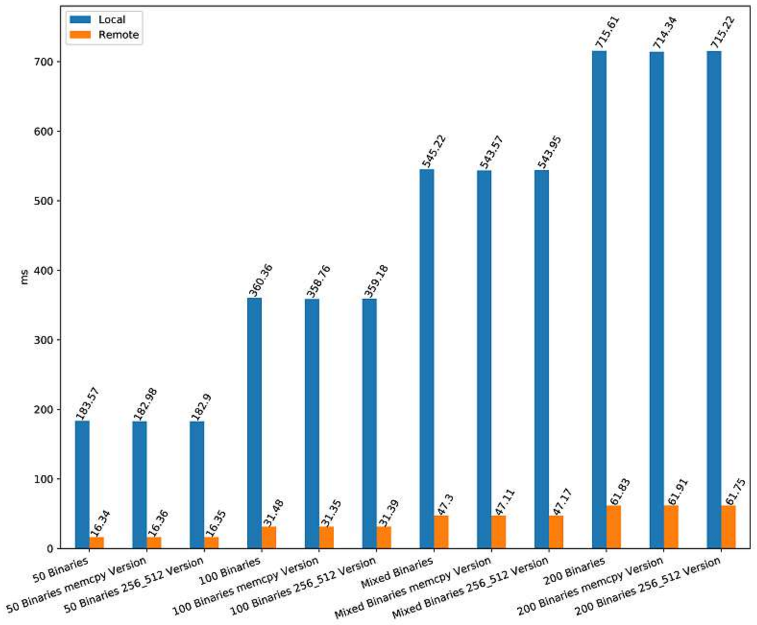

Figure 8.10: Comparing average execution time of different versions on the verification process.

The attestation request measurements show how long it takes to fill the C struct with data and convert them into a byte sequence with CBOR (F_6).

Communication The following paragraph presents the result for establishing the connection between attester and verifier and for transferring the data between them (F_6, F_7). The communication measurements were calculated with the following equation :

$$\text{communication} = \text{attestation process} - (\text{attestor side} + \underbrace{\text{attestrequest} + \text{verification process}}_{\text{verifier side}}) \tag{8.2}$$

In the following each parameter used in the stated equation above will be elaborated. The *attestation process* is the measurement of a whole attestation process between attester and verifier, where the verifier triggers the attestation request and the attester sends in return the attestation response which will be verified by the verifier. Next, the *attester side* is the measurement of loading the key from the NVRAM, creating the TPM Quote, preselecting the affiliated entries of the verifier from the SML, and establishing the response data for the verifier (F_4, F_8). The *attestrequest and verification process* are operations on the verifier side. These are the measurements for generating the attestation request and verifying the response from the attester. The equation above shows how

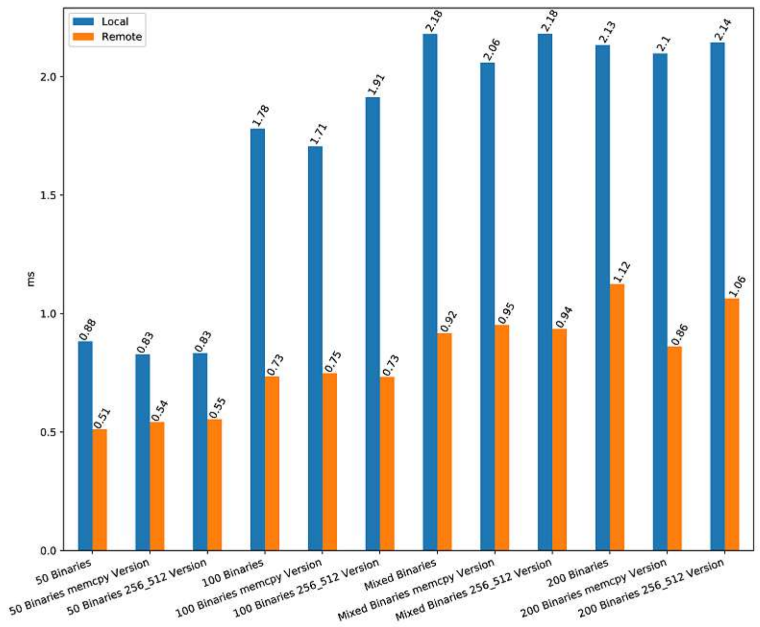

Figure 8.11: Comparing average execution time of different versions on attestation creation.

this work calculated the communication execution time based on the previous presented measurements.

Figure 8.12 is a bar plot which illustrates the different versions with the amount of software selection on the x-axis. The y-axis represents the average execution time for the communication of the PPRA protocol. The interesting part in Figure 8.12 is the local communication establishment takes longer than the remote communication. The communication measurement contains the socket establishment, TLS handshake (S_3), mutual authentication (S_4), and data transfer from both parties (F_6).

Figure 8.13 visualizes a stacked bar plot. The legend describes the average execution time from the attester side (blue), the verifier side (attestation request and verification) (green and orange), and the communication between both parties (red). The stacked bar chart expresses which part of the PPRA protocol consumes the most execution time during an attestation process. The stripes represent the remote communication between two different devices (see Table 8.1).

Performance Measurements of PPRA in Docker The current subsection describes the measured results in the Docker test environment. The Docker setup provides four different versions of the PPRA protocol, because the software TPM is capable of emulating the hash algorithm SHA-512. Here, the whole remote attestation process in

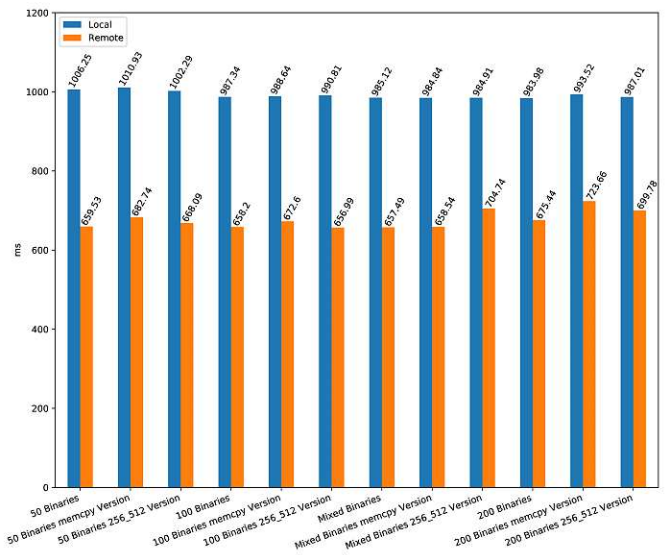

Figure 8.12: Comparing average execution time of the different implementation versions on the communication duration.

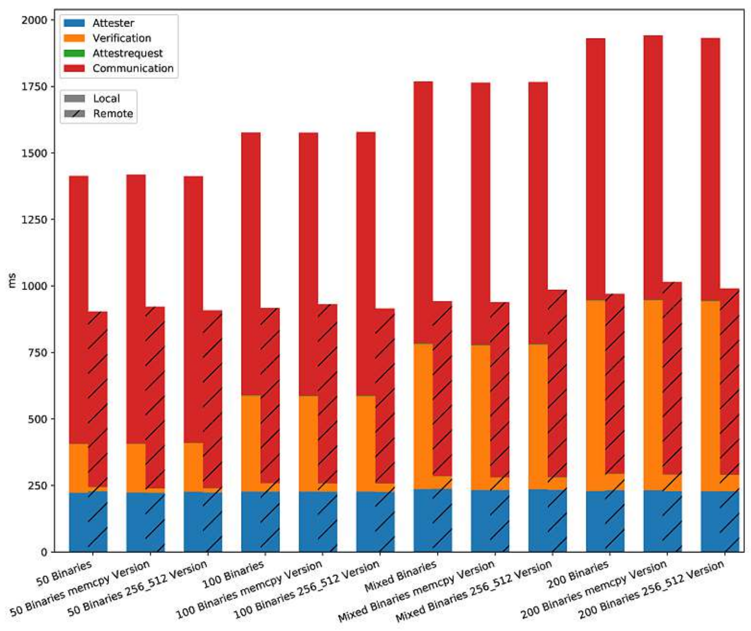

Figure 8.13: Comparison between local and remote execution of the PPRA protocol.

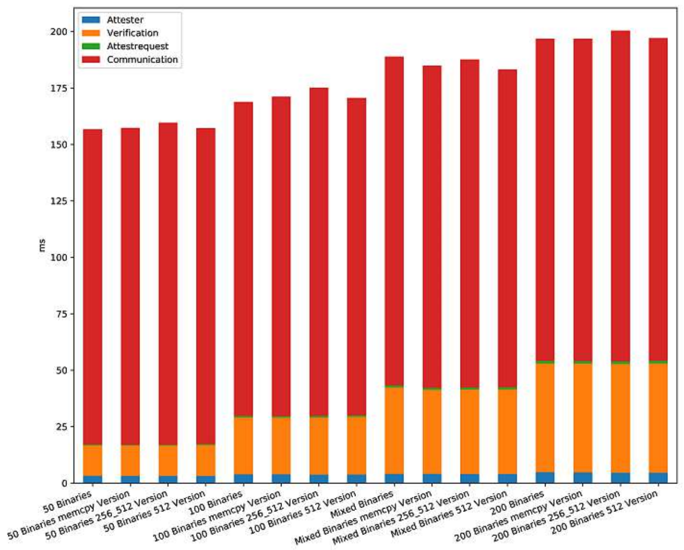

Figure 8.14: Comparing average execution time of attestation process with separation on each operation.

Docker is presented by one stacked bar chart. Moreover, the bar plot does compare between the versions. However, there does not exist the category remote or local. The measurements were taken between two Docker instances, where each simulates a party. Figure 8.14 visualizes a stacked bar plot. The legend describes the average execution time from the attester side, verifier side (attestation request and verification), and communication between both parties. The stacked bar chart expresses which part of the PPRA protocol consumes most of the execution time during an attestation process. The bar highlights that the communication process devours most of the time during an attestation process.

This work not only analyzed the NIZK implementation but also investigated the measurement results of the remote attestation process. The comparison between the adapted CHARRA project and PPRA protocol show the execution time for the *remote attestation process*. Figure 8.7 presents the difference between both projects. The figure illustrates how much execution time must be considered while also providing encrypted communication, mutual authentication, and preserving privacy (S_3, S_4, $F_{10.2}$). The PPRA protocol needs nearly 1200 ms longer than CHARRA, to meet all the assumptions of the security and privacy requirements. Next, the results of the execution time of the whole remote attestation process shows the expected results that the local environment needs more time than the remote, because the verifier device in the remote test environment provides better hardware performance than the local verifier (Figure 8.8). Further, the software selection influences the attestation response. It affects the result as expected by increasing the execution time. The more software selection is made, the more time is required for the attestation process (F_8). This differing implementation version does not show extensive differences (F_{10}). The same result pattern goes for the Docker test environment in Figure A.19. The expected measurement for the average execution time on the attester's side should not differ in the local and remote environment. Figure 8.9 met the expectation. Because Figure 8.9 visualizes none extensive differences between the different implementations. This implicates the implementation differences do not have an enormous impact on the execution time ($F_{10.1}$). Further, the measurement results of the verification process and attestation request generation on the verifier side are stated in Figures A.25, A.26 and 8.10 (F_9). The results visualize an increase in time if the amount of the software selection increases. Otherwise, all the different implementations do not show massive differences in the average execution time. This again leads to the conclusion that the different implementation versions have no impact on the execution time. The communication time was not measured, it was calculated from the measured results above. Moreover, the description on how the execution time was calculated for the communication is stated in Equation (8.2) . The interesting part of the execution time of the communication is, that the amount of the software selection does not affect the results (Figures A.27 and 8.12). The software selection does not increase the average execution time of the communication. Besides, the local communication consumes more time than the remote. The reason is that the local communication is slower than remote due to the communication establishment, handshake, and mutual authentication are conducted on the RPI3. Due to the reason, the RPI3 takes both roles for the local test environment which utilizes more resources.

In the end, all the separate measurements were illustrated on stacked bar charts to see which area in the remote attestation process consumes most of the execution time (F_{10}). If the area is located, it can be addressed in the future for implementation optimization. In both Figures 8.13 and 8.14 state that the communication devours most of the execution time in all test environments and different implementation versions. Therefore, in the future of this work, the communication needs to be optimized.

8.2.3 Communication Cost

This section explains how the communication costs altered when the classic remote attestation was transformed to a privacy-preserving remote attestation (F_7, F_{10}, $F_{10.2}$). The traditional binary attestation sends the entire SML file over the communication channel. However, in this work, the whole SML file will not be sent, due to the reason to preserve the privacy property. Moreover, this subsection proves that the functional requirement F_7 is fulfilled. During the remote attestation one communication round (request-response) is established between verifier and attester (see Section 6.4).
Within this subsection equations are defined, which calculates the communication costs in each direction. These equations illustrate the trade-off between privacy and resource consumption ($F10.2$).

Attestation Request

In the traditional remote attestation, the verifier sends only the *nonce* and the *PCR selection* to the attester. The nonce is used to prevent reply attacks. The size of the nonce is 24 bytes and the size of the PCR selection depends on how many will be selected. In the privacy-preserving remote attestation, the verifier sends the nonce, the PCR selection, and in addition the software selection. The software selection contains the filename and file path. The communication cost increases with each entry in the software selection for the attestation request. The costs of the attestation request from the verifier to attester can be defined by the following equation:

$$\text{attestationrequest} = \text{nonce} + \text{pcrSelection} + n * (\text{filepath} + \text{filename}) \tag{8.3}$$

The n in Equation (8.3) defines the number of entries in the software selection. The file path and filename are the lengths in bytes and 24 bytes nonce. In the stated PoC in Chapter 7, the communication increased in the direction from the verifier to the attester.

Attestation Response

The traditional remote attestation sends as response from the attester to the verifier, the whole SML, which reveals the entire entries to the verifier. Therefore, the PPRA protocol reveals only the affiliated entries to the verifier. However, it increases the communication overhead, it sends extra information to the verifier. This extra information is necessary for the verifier to validate the proof of knowledge. This subsection explains how the communication costs can be calculated. Further, it defines a condition, if this condition is met, then the communication costs should be less than the original communication costs of the traditional remote attestation. Before the equations are explained, Table 8.3 lists the sizes of the parameters in the used equations.

Equation (8.4) shows how the communication costs of the traditional remote attestation can be calculated. The n in the equation stands for the number of entries in the SML.

$$\text{comcosts} = \sum_{i=1}^{n} (\text{filehash}_i + \text{filepath}_i + \text{pcr} + h_T(x_i)) \tag{8.4}$$

Equation (8.5) is a part of Equation (8.6). Equation (8.5) calculates the communication costs of the software selection, which is associated to the verifier. The n_{subset} defines the amount of entries the verifier selected.

$$\text{subset} = \sum_{i=1}^{n_{subset}} (\text{pcr} + \text{eventhash}_i + \text{filehash}_i + \text{filepath}_i + \text{filename}_i + c_i + s_i) \tag{8.5}$$

Attestation Information	**Size [Bytes]**
File path	depends on each entry in SML
Filename	depends on each entry in privacy-preserving SML
File hash (SHA-256/SHA-512)	32/64 Bytes
PCR	1 Byte
$h_T(x)$ (SHA-256/SHA-512)	32/64 Bytes
eventhash	32 Bytes
c	64 Bytes
s	32 Bytes

Table 8.3: Parameter sizes in bytes used in the equations.

Equation (8.6) represents the calculation of the communication costs of the privacy-preserving solution.

$$\text{comcosts\_pp} = (8.5) + \sum_{i=1}^{n-n_{subset}} (\text{pcr} + \text{eventhash}_i + 2) \tag{8.6}$$

Next, the conditions in Equation (8.7) and Equation (8.8), defines how large the communication costs should be for the PPRA protocol to be less than the communication costs of the traditional remote attestation.

$$\begin{aligned}&\sum_{i=1}^{n}(\text{filehash}_i + \text{filepath}_i) > \\ &\sum_{i=1}^{n_{subset}} (\text{filehash}_i + \text{filepath}_i + \text{filename}_i + c_i + s_i) + \sum_{i=1}^{n-n_{subset}} 2\end{aligned} \tag{8.7}$$

$$(8.4) > (8.6)|(8.7) \tag{8.8}$$

These equations stated above generalize the communication costs between verifier and attester. The Equation (8.3) shows, due to ensuring the privacy property, the verifier has to send a software selection, which increases the communication costs in one direction. However, this work states a condition in Equation (8.7), which guarantees to reduce the communication costs from attester to verifier. If the condition holds, the costs will be less than the traditional RA in one direction. To meet the condition the verifier needs to select a minimal set of software, which will not be the case in the real-world scenario. The condition will hold in the theory. The reason is that Equations (8.4) and (8.6) show the difference that PPRA protocol includes more parameters to ensure privacy. The extra two parameters are c and s . The pair (c, s) is necessary for the proof of knowledge. In conclusion, the communication cost will be higher in the newly designed PPRA than the traditional RA. These equations were defined to illustrate that in the security domain to achieve optimal resource utilization, privacy is demanding. Hence, if privacy needs to be ensured then the resource utilization suffers, in case of the work the communication costs.

8.2.4 Use Cases

This work presented a fully functional proof-of-concept of the PPRA. Further, Chapter 4 listed four real-world use cases. To integrate the PPRA protocol in the private user domain, the IMA and the TPM need to be integrated into the mobile device. Further, the Trusted Computing Group (TCG) introduced a mobile architecture, to integrate a TPM for security mechanisms such as Remote Attestation (RA) [63]. Next, the applications of the mobile devices receive updates or patches. Therefore, the IMA needs to provide a service, which updates the measurements or the verifier has Reference Integrity Measurements (RIMs) with the updated measurements and RA will fail. If all the components are given, the PPRA protocol can be integrated and should run in an environment with restricted access. The use cases health care and cyber-physical systems need to integrate as stated, a TPM and the IMA service. However, the environment of cyber-physical systems as well as health care systems, are heterogeneous. Therefore, each facility needs to integrate a TPM and the Linux IMA and to attest each of the facilities. Another option would be to extend the PPRA approach to a collective attestation. In the automotive context, the vehicle should integrate at least five TPMs [41]. Each TPM is assigned in order to preserve the integrity and authenticity of each specific task. For instance, one TPM is responsible for the autonomous driving task while another is in charge of the onboard computer and entertainment system. Hence, for each task, a measurement service needs to be integrated since each task could contain multiple software components. In the end, if the PPRA protocol is integrated, the vendor for the entertainment system attests the entire operational state of the vehicle without gaining the information about the other running software on the vehicle. Further, if the firmware of multiple electronic control units needs to be attested, then a collective approach is necessary. This collective approach aggregates all of electronic control unit measurements and anchors them into the TPM. Thereby, the PPRA protocol can be integrated into the vehicular environment. In conclusion, each of the use cases needs a TPM integrated into their individual devices or infrastructure. Next, a trusted boot process and the IMA needs to be ensured. This work showed that the privacy-preserving remote attestation protocol is a generic approach that can be applied in varying sectors if all the requirements in Chapter 5 are met.

CHAPTER 9

Conclusion and Future Work

This thesis addressed the privacy deficiency of binary attestation by designing a Privacy-Preserving Remote Attestation (PPRA) protocol. In addition, this work realized proof-of-concept based on the design and documented the development of thePPRA protocol. Further, this work conducted a novel analysis based on the requirements analysis and discussed security, privacy, and the performance results of PPRA protocol.

In Chapter 1 it was found that in binary attestation exists an open challenge of preserving privacy. Hence, this work designed an approach, which resolves the open challenge. Chapter 1 elaborated the motivation behind this thesis and the problem statement, why this research is conducted. Moreover, this work aimed to contribute research results for the first two research questions.

Chapter 2 defined the fundamentals, to understand the design approach of the PPRA. The chapter explained the concept of the trusted boot and remote attestation process, where these two components are an integral part of the remote attestation procedure. Further, the basics of abstract algebra were elaborated to follow the explanation of the elliptic curve cryptography and the Schnorr signature scheme.

Chapter 3 described the related work in terms of privacy-preserving remote attestation. Further, it stated the difference between property-based attestation and binary attestation. Additionally, the chapter elaborated the drawbacks of the property-based remote attestation and why binary attestation provides a better advantage in terms of remote attestation.

However, the open challenge in binary attestation is privacy, which was resolved with this thesis. To have a broad understanding on how and where the remote attestation procedure could be applied in real-world, Chapter 4 introduced four use cases. The chapter was introduced to give an overview while applying the traditional remote attestation, where privacy is not given. Therefore, a design has to be conceptualized.

To establish the PPRA approach, a requirements analysis was done. Hence, Chapter 5 listed and described the functional, security and privacy requirements to design the PPRA protocol. These requirements were derived from the use cases in Chapter 4. Chapter 6 presented based on the requirments in Chapter 5 the PPRA approach. First, Chapter 6 illustrated the general idea of the PPRA protocol. Afterwards it elaborated in-depth the adaptions of the traditional remote attestation to ensure privacy. Additionally, the chapter explained how the Schnorr signature was adapted to be advantageous to the design of the PPRA protocol.

This work provided a proof-of-concept of the design in Chapter 7. During the research on how to realize the theory to an actual prototype, this work was confronted by problems. These were explained and the solution was given as well. For instance, the elliptic curve analysis found out that the *Curve25519* cannot be directly applied for the non-interactive zero-knowledge proof. Therefore, this work used the Ristretto technique, which resolved the issue. Furthermore, the architecture design was introduced for the implementation of the PPRA protocol. The algorithms of the Non-Interactive Zero-Knowledge (NIZK) implementation were explained and statistical distance analysis were done in terms of the modulo operation. In addition, the work listed different versions of the implementation to analyze the performance differences.

After, the implementation of the Proof-of-Concept (PoC), Chapter 8 evaluated the PPRA approach based on the functional, security and privacy requirements. Further, this chapter evaluated if the requirements were fulfilled and how. Following, the performance of the PPRA approach was measured, based on the functional requirement. The performance measurements were done to illustrate the resource consumption of the approach on restricted devices (i. e. Raspberry Pi 3 Model B V1.2 (RPI3)). During the evaluation of the privacy and security requirements, this work found out that the design of a privacy-preserving remote attestation with symmetric cryptographic primitive would lead to an over-engineered protocol. Therefore, this work introduced the design with asymmetric cryptographic primitives by applying a non-interactive zero-knowledge proof known as Schnorr signature. Besides, the outcome of the evaluation was that all the requirements were fulfilled based on the design of the PPRA protocol and the PoC.

Finally, Chapter 8 discussed the achieved privacy and security. The achievements were argued, based on - the elliptic curve operations, the properties of NIZK and the Trusted Platform Module (TPM). Moreover, the performance evaluation point out the most resource consumption in the PoC is located during the communication between verifier and attester. In the end, the chapter described how feasible the designed PPRA protocol is to be integrated into the use cases from Chapter 4.

Overall, this thesis provided a solution that addresses one of the open challenges in binary attestation, while merging the traditional Remote Attestation (RA) with a privacy-enhancing technology. Further, this contribution should be a motivation to the research society in the remote attestation.

Future Work

This thesis presented how to resolve the privacy problem in binary attestation. Further, this work should be considered as a foundation for future works and follow-up projects.

Implementation Optimization As stated in Chapter 8 the communication between attester and verifier consumes most of the execution time. Therefore, a future optimization task is to reduce the execution time. The suggestion for improving the optimization is by switching out the *mbedTLS* library with *noise-c* library. The *noise-c* library is an implementation of the Noise protocol in C [53]. The Noise protocol is a framework, where the developers can implement cryptographic protocols based on Diffie-Hellman key agreement. Further, it would be interesting to see if the *noise-c* can be used in the embedded systems domain. The authors of the Noise Protocol Framework compared their concept with TLS 1.2. The findings are [52]:

- Noise protocol does not reveal the identities during the handshake since the handshake is encrypted except for the ephemeral public key.
- Noise protocol does only need one round trip before the client sends data. However, the TLS 1.2 defines 2 rounds.
- If the ephemeral ECDH is used in TLS 1.2, signatures are required. Nevertheless, the Noise protocol does not require signatures, since it relies only on ECDH.

All these findings illustrate the benefit of the Noise protocol. Further, it reduces the bandwidth, and it is stated to be a robust protocol. Therefore, it is a follow-up project to see the evaluation, based on the Noise protocol, if it will reduce the execution time.

Besides, the implementation optimization, the communication costs can be reduced as well by removing the filename from the Equations (8.3), (8.5) and (8.7). By removing the filename, the policy check needs to be changed. The change is to compare the file path instead of the filenames since the file path contains the basename. In this work, the filename was considered for simplification. Considering the scope of this work was if it was possible to implement a non-interactive zero-knowledge proof while guaranteeing the integrity, authenticity, and privacy of a systems operational state.

Another suggestion is to change the policy check, in Figure 7.7. It can be changed by integrating the policy into the verifier's certificate. For instance, if the certificate is issued for the verifier it integrates the affiliated binaries/software components. In the current solution the *mbedTLS* does not provide such an option, but in OpenSSL. OpenSSL allows to abuse the OID section in the certificate to put customized information such as the checklist for the affiliated binaries [16, 35]. Therefore, the attester does not need to have local policy files to check, instead it extracts the information from the certificate and cross-checks with the received software selection [35].

IMA An important future work is to integrate this work from the user space into the kernel space . Moreover, if the project is integrated into the kernel space it needs to be unified with Integrity Measurement Architecture (IMA) as a new operation for maintaining privacy. However, the IMA needs to be upgraded in the future as well from SHA-1 to SHA-256/SHA-512 (collision resistant hash algorithms). Thus, the PPRA protocol needs to be less adapted. The interesting parts will be the performance evaluation in the kernel space as well as the upcoming hurdles of integrating the project into the kernel space and merging it with IMA.

Collective Attestation The collective attestation is an attestation method, in a network, where it collects from every device (in the network) the attestation and sends the collection to the verifier. The verifier verifies the received attestation from the whole network. The traditional attestation scheme is one attester and one or multiple verifiers. The collective attestation is if the verifier needs to attest multiple devices in one network. Therefore, in the traditional scheme, the verifier needs to do a request-response with each device in the network. However, the collective attestation scheme collects all the attestations and one device sends the aggregated attestation response to the verifier. Here, it would be interesting as future work, if the current design of this work is capable of being integrated into an existing work of the collective attestation or if it can be extended to a collective approach [40]. The future work should not only be preserving the existing privacy property, it needs to provide confidentiality as well. If confidentiality is given, then the verifier will be unable to see the blueprint of the network or the IT infrastructure. The survey about remote attestation states [57], the open challenges in remote attestation are privacy and confidentiality.

Property-based Attestation The property-based attestation was explained in Chapter 3. If a concrete implementation of property-based attestation exists which implements the proof of membership, it would be interesting to compare the performance differences with this work.

Here, this work briefly answers the third research question in Chapter 1. The major two research questions in Chapter 1 were answered by Chapters 6 and 7. The third research question is future work because with the current PPRA approach the question cannot be addressed. However, in the following paragraph, this work briefly argued the reason and provided solutions on how the current approach can be extended.

RQ3: How could such an approach be used in a scenario with multiple verifiers in order to prove to an interested third party that all entries of the Stored Measurement Log (SML) are verified?

Answer The answer is, with the current design and implementation, not all verifiers may prove to the interested third party that all entries of the SML are valid. Due to this reason, that each verifier needs to send the non-interactive zero-knowledge proof with the template hash (see Chapter 6). Therefore, the interested third party will know all the running software on the attester's system. Moreover, by knowing the template hashes the interested third party has the power to execute the attack vector described in Section 6.1 and the privacy property is not preserved anymore.

Another option to convince the interested third party is that each verifier sends only the TPM signature, the column *event hash* of the SML and the *TPM Quote*. The interested third party verifies all the received data from each verifier and only if all of them are valid then the whole SML is valid. Here, the benefit is that the interested third party does not know what software is running on the attester side, but it is convinced that the integrity and authenticity are intact. The drawback is that the interested third party needs to verify every received data from multiple verifiers, which could take some time. This option forces the interested third party to trust the verifiers, whether they are sending accurate information.

Another approach is to extend the current work on the verifier side, where all verifier initiates an interactive zero-knowledge proof of their verification result. The idea was taken from Chatzigiannakis et al. [17], where they introduced an interactive zero-knowledge proof for a single bit. The concept is to encode the verification result of the verifier as a bit, zero for failed attestation and one for successful attestation. The benefit is the third party knows if all the interactive zero-knowledge proofs are valid then the attester is trustworthy. However, the drawbacks are the verifier side has extra communication overhead and there is no guarantee if all the subsets of all verifiers cover all entries of the SML.

The last idea is to use a Public Key Infrastructure (PKI). The PKI which issues the certificate to the verifier and the attester should receive the validation result from all verifier. The interested third-party requests from the PKI if all SML entries are valid. The PKI will cross-check the results of all verifiers and if and only if all the results are true then the PKI gives the green light to the interested third party that the entries of the SML are trustworthy. Here, the drawback is a interested third party (PKI) is introduced and the concept is dependable on the PKI.

In conclusion, the author of this work thought about useful ideas and suggested them for the research question. Moreover, for each idea, their drawbacks and benefits were mentioned. Therefore, this question is considered to be a future work to extend the current work with ideas and implementation to formulate a concrete answer.

APPENDIX A

Appendix

The appendix provides further plots from the evaluation chapter and C implementation listings. These plots are for interested readers to see the difference between remote and local execution of the Privacy-Preserving Remote Attestation (PPRA) protocol. Some of them are not listed in Chapter 8, although some of them were used to discuss the evaluation outcome in Chapter 8.

A.1 C Implementation

The listings below demonstrate the accurate implementation with the *Libsodium* library. The Listing A.1 illustrates in C the same computation steps of the Algorithm 7.1.

```
int nizksign_eventrecord(eventrecord *rec)
{
  if (sodium_init() == -1)
  {
    printf("init is minus one");
    return 1;
  }

  uint8_t digest[TPM2_SHA256_DIGEST_SIZE];
  uint8_t *random = calloc(crypto_core_ristretto255_SCALARBYTES,
      sizeof(uint8_t));
  uint8_t *reduced_digest = calloc(
      crypto_core_ristretto255_SCALARBYTES, sizeof(uint8_t));
  uint8_t *r_h = calloc(crypto_core_ristretto255_SCALARBYTES,
      sizeof(uint8_t));
  uint8_t *event_hash = calloc(crypto_core_ristretto255_BYTES,
      sizeof(uint8_t));
  uint8_t *v = calloc(crypto_core_ristretto255_SCALARBYTES, sizeof(
      uint8_t));
    uint8_t *g_i = calloc(crypto_core_ristretto255_BYTES, sizeof(
        uint8_t));
    uint8_t *t_i = calloc(crypto_core_ristretto255_BYTES, sizeof(
        uint8_t));
    uint8_t *reduced_c = calloc(crypto_core_ristretto255_SCALARBYTES,
         sizeof(uint8_t));
    uint8_t *r_c = calloc(crypto_core_ristretto255_SCALARBYTES,
        sizeof(uint8_t));
    uint8_t *s = calloc(crypto_core_ristretto255_SCALARBYTES, sizeof(
        uint8_t));

    /*Templatehash*/
    //templatehash(rec->event.e[0].file_path, digest);
    templatehashevent(&rec->event.e[0], digest);
    /*random scalar r generating*/
    crypto_core_ristretto255_scalar_random(random);


    unsigned char c1[crypto_generichash_BYTES_MAX];
    //memcpy(c1,digest,TPM2_SHA256_DIGEST_SIZE);
    //memcpy(&c1[TPM2_SHA256_DIGEST_SIZE],digest,
        TPM2_SHA256_DIGEST_SIZE);


    crypto_hash_sha512_state sha512state;

    crypto_hash_sha512_init(&sha512state);
    crypto_hash_sha512_update(&sha512state, digest,
        TPM2_SHA256_DIGEST_SIZE);
    crypto_hash_sha512_final(&sha512state, c1);

    crypto_core_ristretto255_scalar_reduce(reduced_digest, c1);


    /*r * h_t(x)*/
    crypto_core_ristretto255_scalar_mul(r_h, random, reduced_digest);

    if (crypto_scalarmult_ristretto255_base(event_hash, r_h) != 0)
    {
      printf("base mult for nd not okay");
    }

    if (crypto_core_ristretto255_is_valid_point(event_hash) != 1)
    {
      printf("ND is not valid");
    }


    rec->event_hash = calloc(crypto_core_ristretto255_BYTES, sizeof(
        uint8_t));
    memcpy(rec->event_hash, event_hash,
        crypto_core_ristretto255_BYTES);


    /*END OF ND-HASH*/

    /*Generator */
    crypto_scalarmult_ristretto255_base(g_i, reduced_digest);
    if (crypto_core_ristretto255_is_valid_point(g_i) != 1)
    {
      printf("gi is not valid");
    }


    /*t_i*/
    //randombytes_buf(v, sizeof v);
    crypto_core_ristretto255_scalar_random(v);
    if(crypto_scalarmult_ristretto255(t_i, v, g_i) == -1) printf("not
         valid t_i");
    if (crypto_core_ristretto255_is_valid_point(t_i) != 1)
    {
      printf("t_i is  not valid");
    }


    /*c_i*/
    uint8_t c[crypto_generichash_BYTES_MAX];
    crypto_generichash_state state;

    crypto_generichash_init(&state, NULL, 0, sizeof c);

    crypto_generichash_update(&state, g_i,
        crypto_core_ristretto255_BYTES);
    crypto_generichash_update(&state, t_i,
        crypto_core_ristretto255_BYTES);
    crypto_generichash_update(&state, event_hash,
        crypto_core_ristretto255_BYTES);
    //crypto_generichash_update(&state, nonce, crypto_box_NONCEBYTES)
        ;

    crypto_generichash_final(&state, c, sizeof c);
    rec->c = calloc(crypto_generichash_BYTES_MAX, sizeof(uint8_t));
    memcpy(rec->c, c, crypto_generichash_BYTES_MAX);

    /*s = v -r*c */
    crypto_core_ristretto255_scalar_reduce(reduced_c, c);
    crypto_core_ristretto255_scalar_mul(r_c, random, reduced_c); // (
        a*c)mod group order
```

```

  crypto_core_ristretto255_scalar_sub(s, v, r_c); // (v-z)mod group
      order


  rec->s = calloc(crypto_core_ristretto255_SCALARBYTES, sizeof(
     uint8_t));
  memcpy(rec->s, s, crypto_core_ristretto255_SCALARBYTES);

  free(random);
  free(reduced_digest);
  free(r_h);
  free(event_hash);
  free(v);
  free(g_i);
  free(t_i);
  free(reduced_c);
  free(r_c);
  free(s);

  return 0;
}
```

Listing A.1: NIZK signing.

The Listing A.2 illustrates in C the same computation steps of the Algorithm 7.2.

```
bool nizkverify_eventrecord(eventrecord *rec)
{


  if (sodium_init() == -1)
  {
    printf("init is minus one");
    return 1;
  }
  bool verify = false;

  uint8_t digest[TPM2_SHA256_DIGEST_SIZE];
  uint8_t *reduced_digest = calloc(
     crypto_core_ristretto255_SCALARBYTES, sizeof(uint8_t));
  uint8_t *g_i = calloc(crypto_core_ristretto255_BYTES, sizeof(
     uint8_t));
  uint8_t *g_i_s = calloc(crypto_core_ristretto255_BYTES, sizeof(
     uint8_t));
  uint8_t *event_hash_c = calloc(crypto_core_ristretto255_BYTES,
     sizeof(uint8_t));
  uint8_t *t_i_prime = calloc(crypto_core_ristretto255_BYTES, sizeof(
     uint8_t));
```

```
uint8_t *reduced_c = calloc(crypto_core_ristretto255_SCALARBYTES,
    sizeof(uint8_t));

/*Templatehash*/ /*Templatehash*/

templatehashevent(&rec->event.e[0], digest);



unsigned char c1[crypto_generichash_BYTES_MAX];
//memcpy(c1,digest,TPM2_SHA256_DIGEST_SIZE);
//memcpy(&c1[TPM2_SHA256_DIGEST_SIZE],digest,
    TPM2_SHA256_DIGEST_SIZE);

crypto_hash_sha512_state sha512state;
crypto_hash_sha512_init(&sha512state);
crypto_hash_sha512_update(&sha512state, digest,
    TPM2_SHA256_DIGEST_SIZE);
crypto_hash_sha512_final(&sha512state, c1);

crypto_core_ristretto255_scalar_reduce(reduced_digest, c1);


/*Generator */
crypto_scalarmult_ristretto255_base(g_i, reduced_digest);
if (crypto_core_ristretto255_is_valid_point(g_i) != 1)
{
  printf("gi is not valid");
}

if(crypto_scalarmult_ristretto255(g_i_s, rec->s, g_i) == -1) printf
    ("not valid g_i_s!");
if (crypto_core_ristretto255_is_valid_point(g_i_s) != 1)
{
  printf("gis is not valid");
}


crypto_core_ristretto255_scalar_reduce(reduced_c, rec->c);
if(crypto_scalarmult_ristretto255(event_hash_c, reduced_c, rec->
    event_hash) == -1) printf("not valid event_hash_c");
if (crypto_core_ristretto255_is_valid_point(event_hash_c) != 1)
{
  printf("ndc is not valid");
}

if (crypto_core_ristretto255_add(t_i_prime, g_i_s, event_hash_c) !=
    0)
{
    printf("Point addition was not an success!!\n");
  }

  unsigned char c_prime[crypto_generichash_BYTES_MAX];
  crypto_generichash_state state;

  crypto_generichash_init(&state, NULL, 0, sizeof c_prime);
  crypto_generichash_update(&state, g_i,
     crypto_core_ristretto255_BYTES);
  crypto_generichash_update(&state, t_i_prime,
     crypto_core_ristretto255_BYTES);
  crypto_generichash_update(&state, rec->event_hash,
     crypto_core_ristretto255_BYTES);
  crypto_generichash_final(&state, c_prime, sizeof c_prime);

  if (memcmp(rec->c, c_prime, crypto_generichash_BYTES_MAX) == 0)
  {
    verify = true;

  }

  free(reduced_digest);
  free(event_hash_c);
  free(g_i_s);
  free(g_i);
  free(t_i_prime);
  free(reduced_c);

  return verify;
}
```

Listing A.2: NIZK verification.

A.2 Measurements of NIZK SHA-256 memcpy Version

The measurements of the NIZK SHA-256 memcpy version are not in Chapter 8, since it showed the same behavior as the other versions.

A.2.1 Measurement on Intel

The measurements of the CPU cycles and execution time are measured on the Intel architecture.

CPU Cycles

In Chapter 7 this work explained precisely why three versions of the non-interactive zero-knowledge process exist. Therefore, the Figure A.1 illustrates how the second

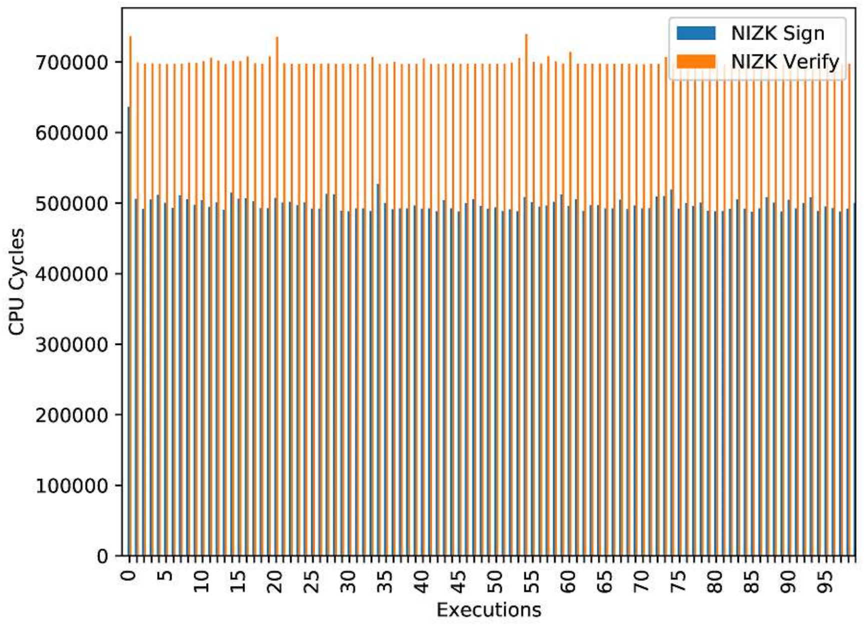

Figure A.1: CPU cycle measurement of the NIZK signing and verifying process SHA-256 memcpy version (Intel).

version of the Non-Interactive Zero-Knowledge (NIZK) performs on the Intel CPU. In the beginning, it illustrates the same behavior as in Figure 8.1, which illustrates the necessity for more instructions due to the library initialization. Further, the bar chart tries to level off for the different binaries, however some standout but not extremely. Next, the NIZK SHA-256 memcpy version does need fewer instructions than the other version. The comparison between the version will be analyzed at the end of the section.

A.2.2 Execution Time

Figure A.2 shows the measurement of the SHA-256 memcpy version of the NIZK. The comparison between Figure 8.2 and Figure A.2 visualizes that there does not exist much difference, since the range of the execution time resembles the same range.

A.2.3 Measurement on ARM

This section continues with the NIZK SHA-256 memcpy version of the ARM architecture.

CPU Cycles

The next Figure A.3, shows how many instructions the SHA-256 memcpy version implementation need. Here it is visible that the measured values are in the same range as in the previous Figure 8.3. Moreover, it indicates the same behavior at the beginning with

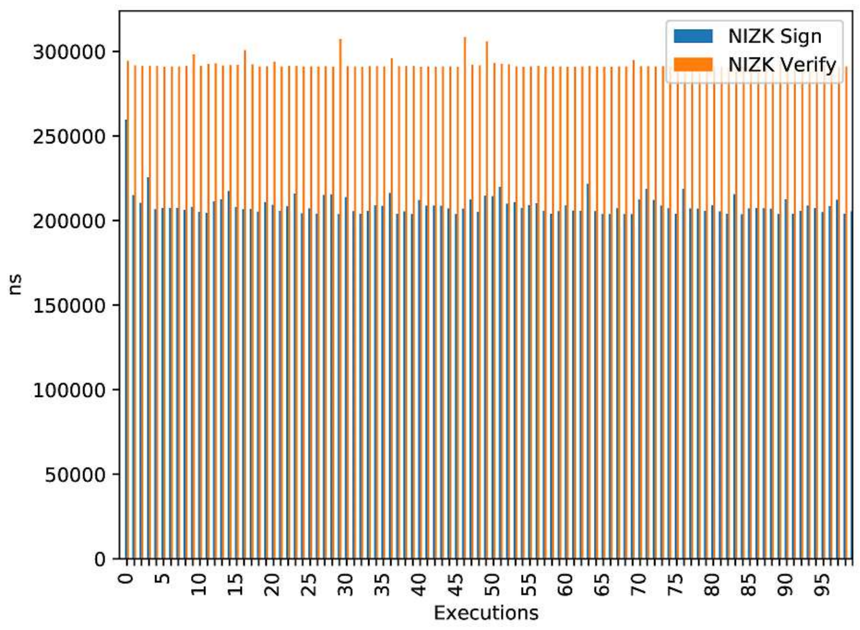

Figure A.2: Execution time measurement of the NIZK signing and verifying process sha256 memcpy version (Intel).

the Libsodium library initialization. After the peak, the signing and verification process even out on the upcoming executions.

The range of the two versions (Figure 8.3,Figure A.3) on the ARM architecture are the same, however a minimal difference is visible.

Execution Time

The next Figure A.4 illustrates the execution time of the SHA-256 memcpy version. The same behavior pattern appears in Figure A.4 as well as in Figure 8.4. This pattern illustrates the ARM architecture needs for the first seven binaries more time and after, it level-offs for the rest of the binaries.

A.3 Measurements of NIZK SHA-512 Version

The measurements of the NIZK SHA-512 version are not in Chapter 8, since it showed the same behavior as the other versions.

A.3.1 Measurement on Intel

The measurements of the CPU cycles and execution time are measured on the Intel architecture.

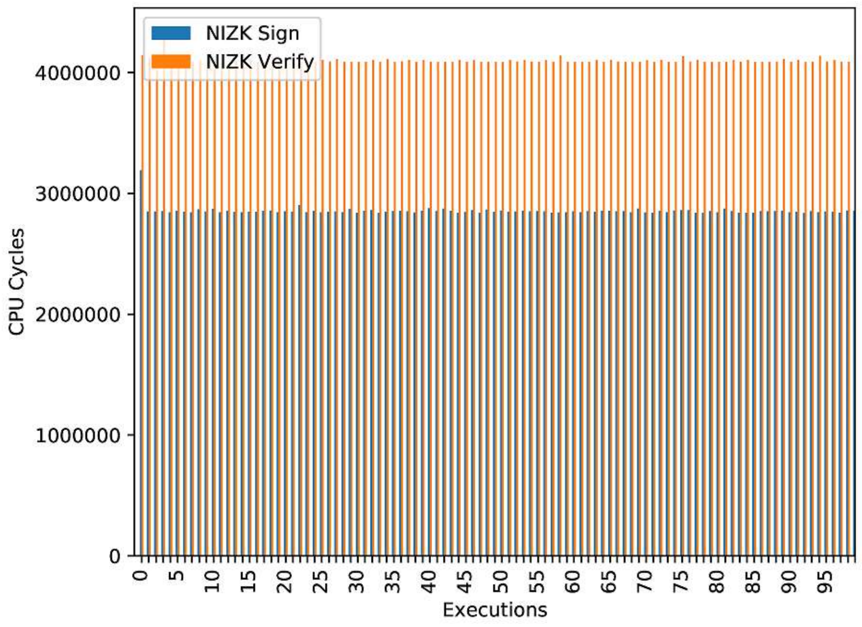

Figure A.3: CPU cycle measurement of the NIZK signing and verifying process sha256 memcpy version (ARM).

CPU Cycles

Figure A.5 shows at the beginning a peak, which indicates the Libsodium library initialization. The x-axis represents the different execution on different binaries. The y-axis provides the information on how many instructions are necessary to do the NIZK signing and verification process.

Execution Time

Figure A.6 shows the execution time for each iteration with different measurements(binary measurement) for the NIZK SHA-512 version. The y-axis provides the measurement in nanoseconds.

A.3.2 Measurement on ARM

This section continues with the NIZK SHA-512 version of the ARM architecture.

CPU Cycles

Figure A.7 shows at the beginning a peek, which indicates the Libsodium library initialization. The x-axis represents the different execution on different binaries. The y-axis

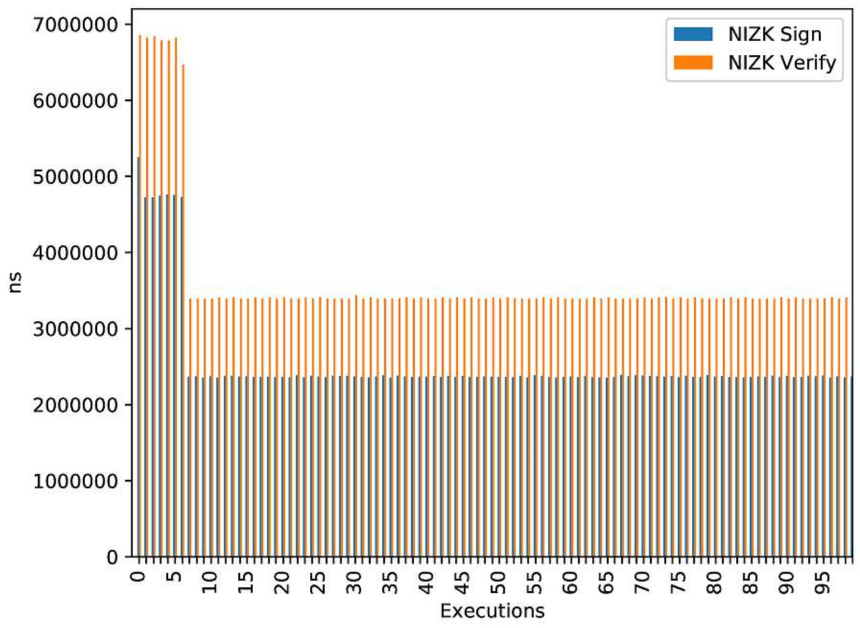

Figure A.4: Execution time measurement of the NIZK sign and verification process memcpy version (ARM).

provides the information on how many instructions are necessary to do the NIZK signing and verification process.

Execution Time

Figure A.8 illustrates the execution time for 100 iterations for the signing and verification process. It behaves the same as the other versions, the ARM architecture even off the execution time after 15 iterations.

Comparisons of Architectures and Versions

Figure A.9 compares the three versions of the NIZK implementation. At the beginning of the signing process, a peak is clearly visible in all three versions which are indicating the Libsodium library initialization. The legend in Figure A.9 is based on the order of the appearance of the line charts. Further, all three versions of the NIZK verification process on the ARM architecture have some outliners, and after, it evens out.

Figure A.10 represents the behavior over 100 executions and each execution is done with a different binary based on the execution time.

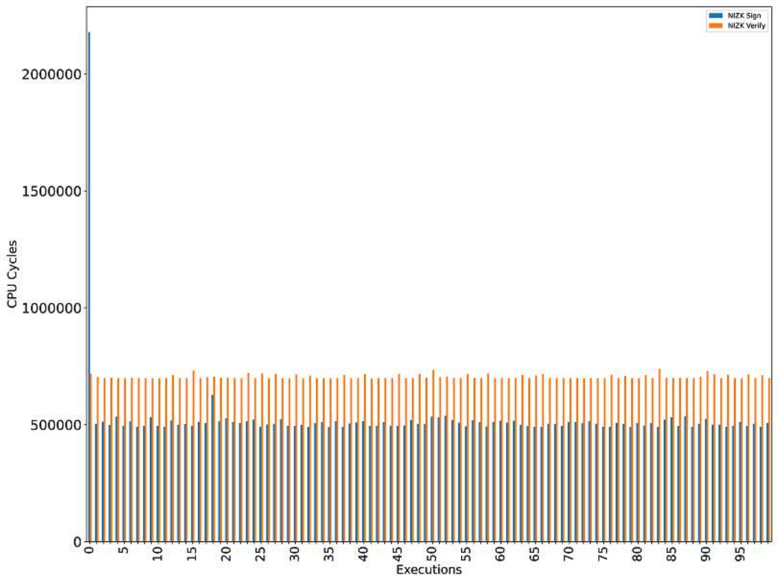

Figure A.5: CPU cycle measurement of the NIZK signing and verifying process(Intel).

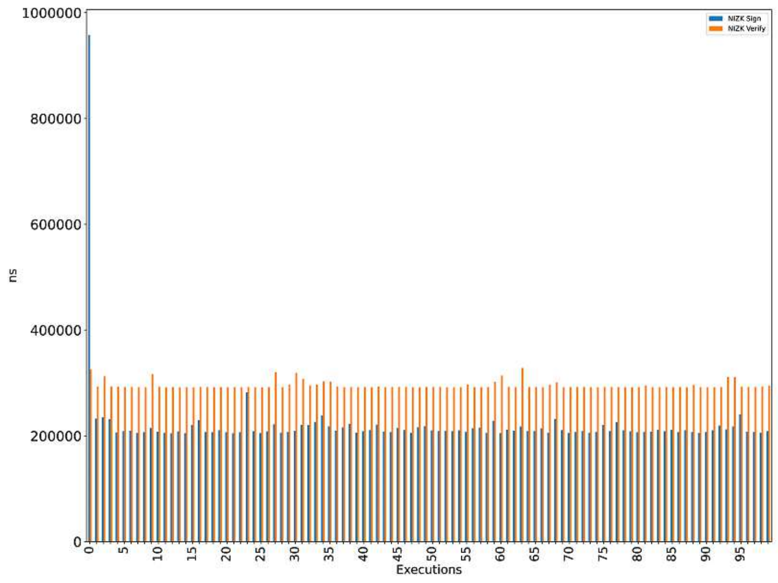

Figure A.6: Execution time measurement of the NIZK signing and verifying process (Intel).

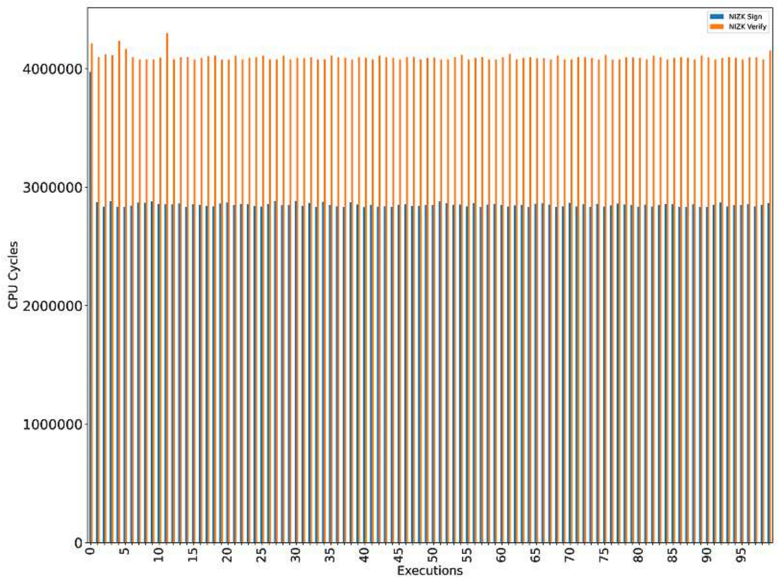

Figure A.7: CPU cycle measurement of the NIZK signing and verifying process(ARM).

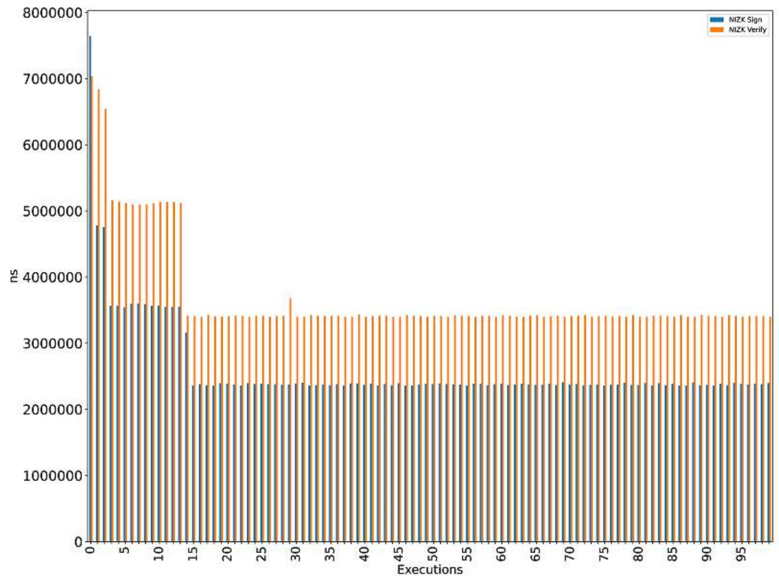

Figure A.8: Execution time measurement of the NIZK signing and verifying process (ARM).

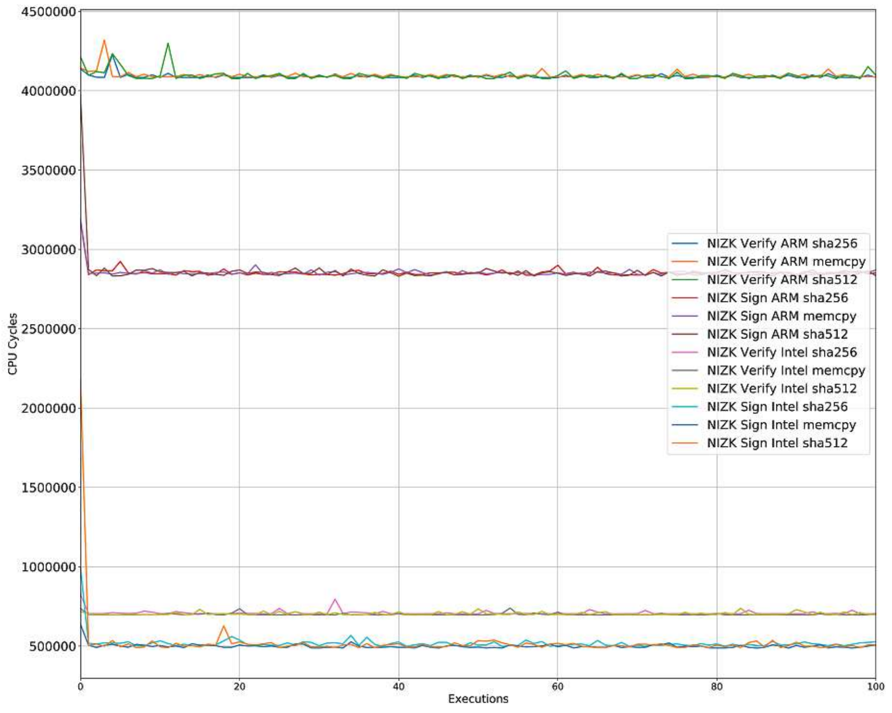

Figure A.9: Comparison between architecture and versions (CPU cycles).

A.4 Performance Measurements of PPRA in a Hardware Environment

This section points out the comparison between the CHARRA and PPRA. Further, it will illustrate the outcomes of the measurement on the attester and verifier sides.

A.4.1 Comparison with CHARRA

Figure A.11 is a line chart which expresses the behavior of CHARRA and PPRA on 50 iterations. It concludes that the hardware Trusted Platform Module (TPM) is inconsistent, while creating a quote it fluctuates often. The line charts in figure A.12 show that CHARRA and PPRA have a difference of around 1 ms nearly in every iteration.

Figure A.13 illustrates how long both projects need for the operation *TPM_Quote*. The x-axis represents different executions of CHARRA and PPRA. The y-axis shows the execution time in ms for each execution in PPRA protocol and CHARRA. Both projects are in the range between 208 ms and 220 ms for establishing the TPM Quote, but the measurement gap is not extensive.

The bar plot (Figure A.14) visualizes the comparison between CHARRA and PPRA for the function loading the key from the NVRAM of the TPM. The x-axis represents different executions of CHARRA and PPRA. The y-axis shows the execution time in ms

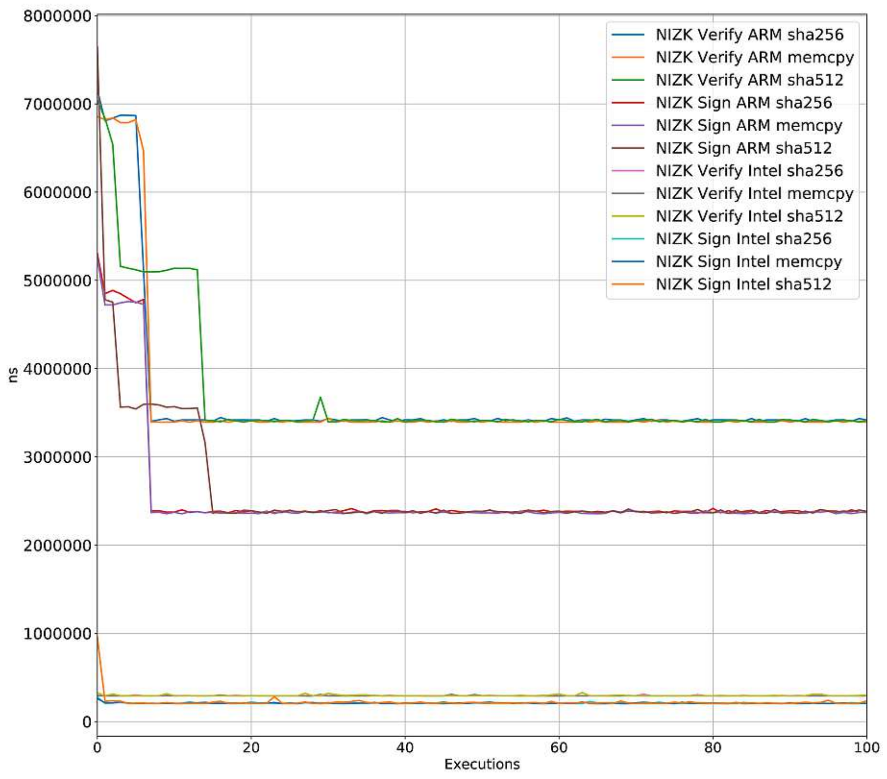

Figure A.10: Comparison between architecture and version (Execution time).

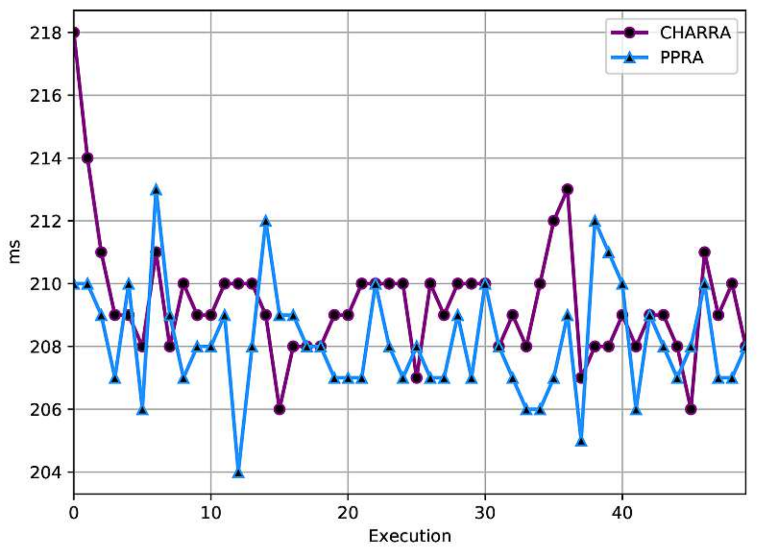

Figure A.11: Comparsion between CHARRA and PPRA for the TPM Quote operation.

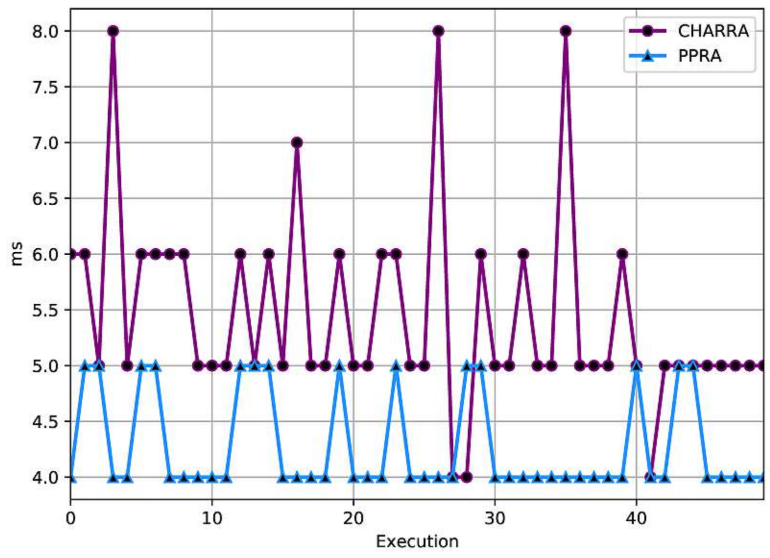

Figure A.12: Comparsion between CHARRA and PPRA for loading the key from the TPM.

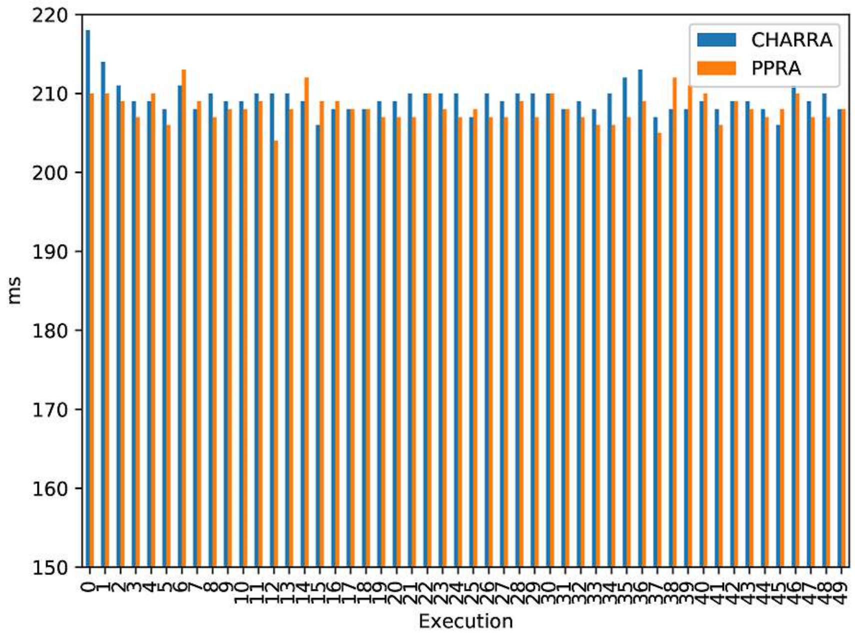

Figure A.13: Comparison between CHARRA and PPRA on TPM Quote.

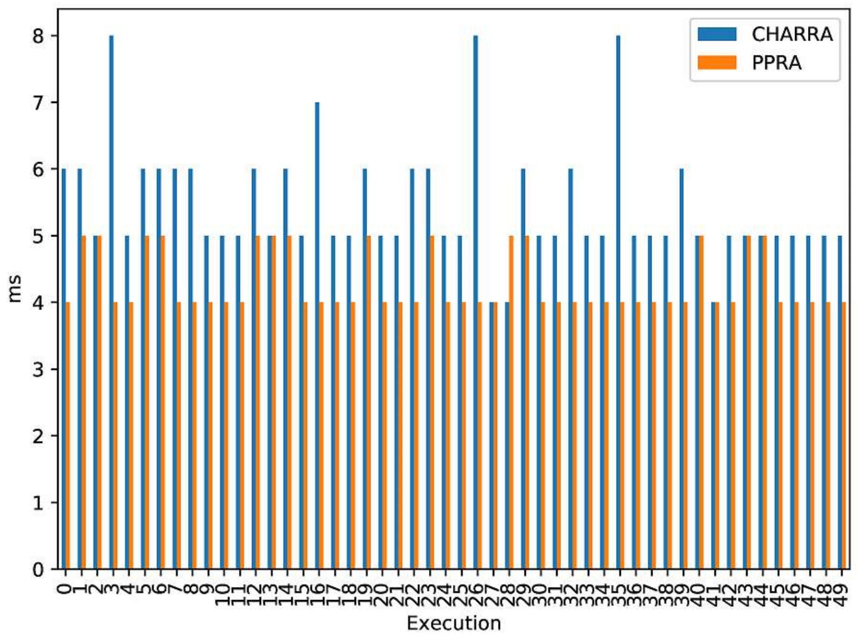

Figure A.14: Comparison between CHARRA and PPRA on loading the key from the TPM.

for each in PPRA protocol. The measurement shows that PPRA takes less than one ms than CHARRA for loading the key from the NVRAM.

A.4.2 Attestation Process

Figure A.15 compares all remote versions of the PPRA protocol implementation. This chart illustrates for the reader how it behaves during 50 iterations. It is visible there are some outliers. Further, all versions are nearly in the same range of 800 to 1200 ms after the first three iterations. Figure A.16 compares the execution time of the SHA-256 memcpy version and SHA-256 version of the PPRA protocol implementation over 50 iterations. Further, it shows the results of the PPRA protocol on a local and remote connection. The legend shows not only the version it is also indicating the amount of software selection, which needed to be attested. Figure A.16 compares the execution time of the mixed version with SHA-256 and SHA-512 of the PPRA protocol implementation over 50 iterations. Further, it shows the results of the PPRA protocol a local and remote connection. The legend shows not only the version it is also indicating the amount of software selection, which needed to be attested. Moreover, the Figure A.16 expresses the local execution on ARM Architecture needs more execution time than remote connection. Next, the number of software selections indicates the higher the selection the execution time will increase. Figure A.18 compares all various versions from PPRA protocol implementation, which is executed on the Raspberry Pi 3 Model B V1.2 (RPI3), where

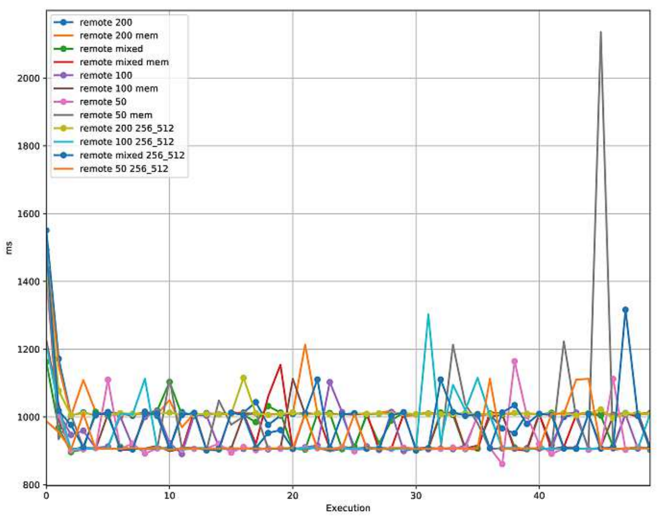

Figure A.15: Comparing all remote versions.

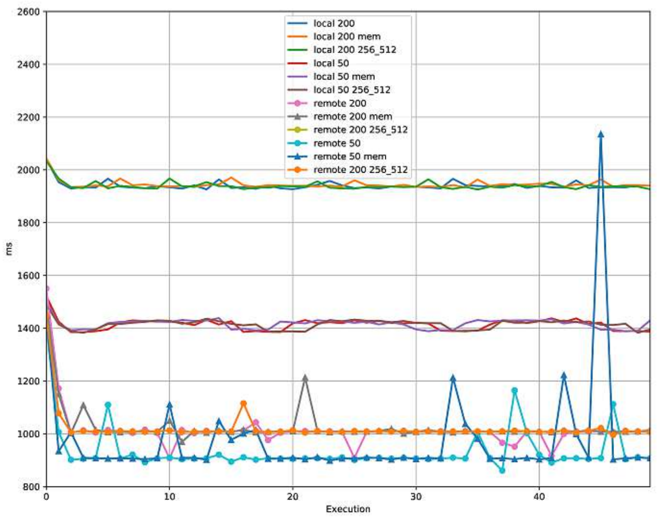

Figure A.16: Comparing execution time between different versions with different software selections.

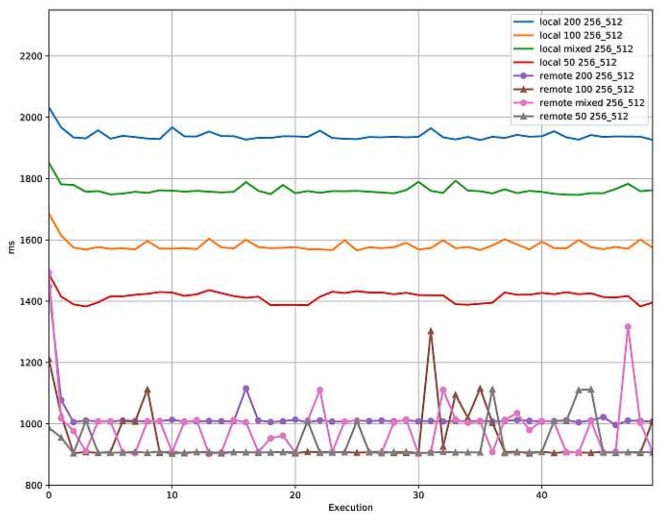

Figure A.17: Comparing all version of the mix version.

the RPI3 simulates both roles of the verifier and attester. The line chart visualizes the linear increase in the execution time in all versions due to the increase in the software selection. This should be the expected behavior.

A.5 Performance Measurements of PPRA in Docker

This section provides the reader with further evaluation results from the simulation perspective. These charts were plotted from the measurement results of the docker test environment.

A.5.1 Attestation Process

Figure A.19 visualizes on the x-axis all different versions of the PPRA protocol implementation same as in Figure 8.8 except the Docker setup provided the opportunity to implement one more version. The y-axis illustrates the average execution time for the whole attestation process between verifier and attester. Figure A.20 illustrates on the x-axis each iteration of an execution. Each iteration represents each version with a different amount of software selection. In total each execution presents 16 bars. Since each execution shows 16 bars only the first ten iterations are shown. Figure A.20 shows how each of the versions behaves on each execution.

Figure A.21 visualizes the 50 iterations of each execution from the different versions of the PPRA protocol with a dissimilar amount of software selections. The idea of Figure A.21

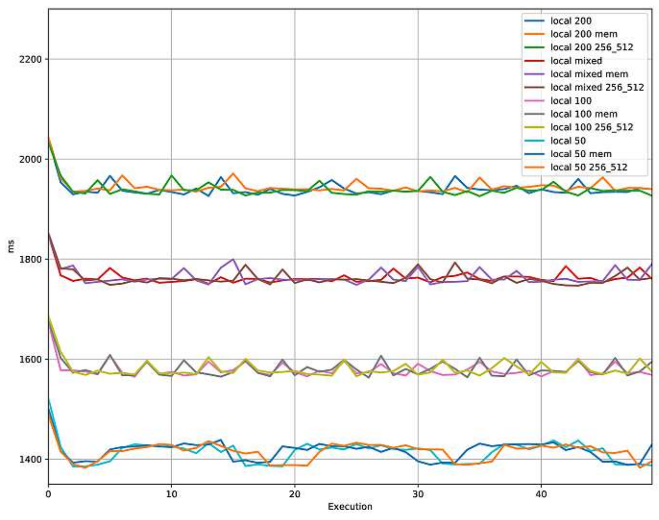

Figure A.18: Comparing all local versions.

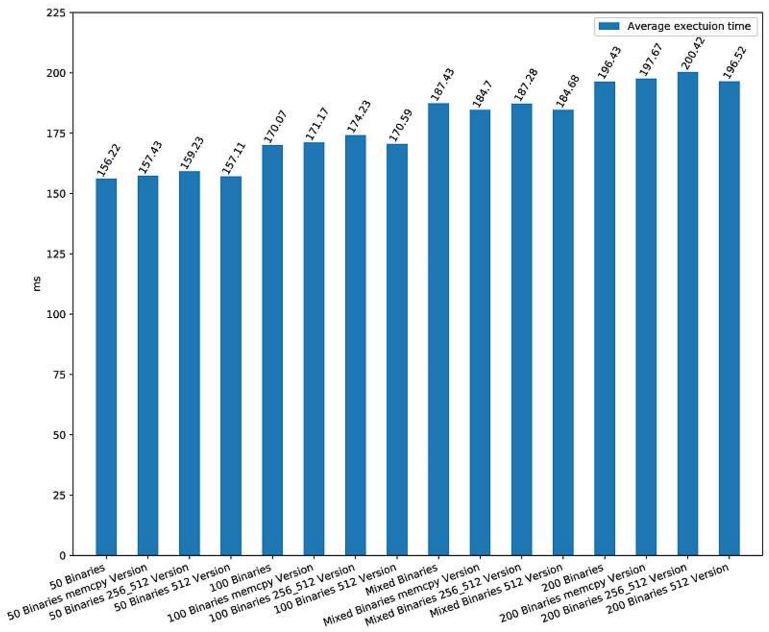

Figure A.19: Comparing various versions of the attestation process (Docker).

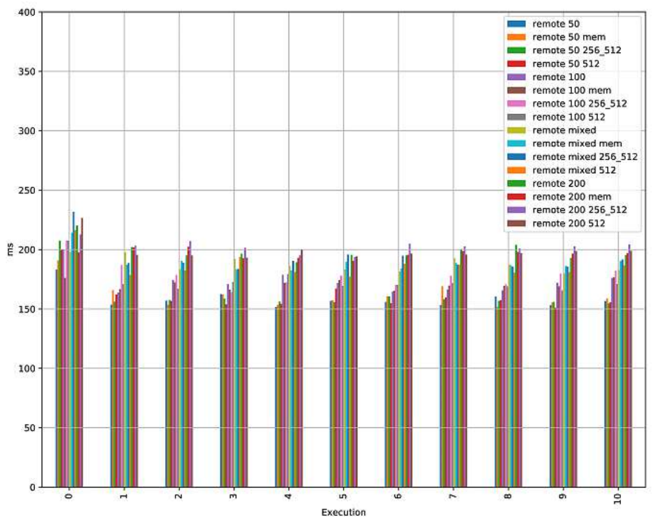

Figure A.20: Comparing different versions of the attestation process (Docker).

is to show that the amount of software selection has an impact on the execution time during the attestation process.

A.5.2 Attester

Figure A.22 illustrates the average time from the attester side for loading the key from the NVRAM, creating the TPM Quote, establishing the response struct, and convert into a byte sequence. Figure A.23 provides the same characteristics as Figure A.20. The only difference is the measurements are taken only on the attester side. The fascinating part is for the first execution it takes a little more than other execution. After the second execution on the x-axis, it is visible that each version is leveling off. Figure A.24 provides the same characteristics as figure A.21. The only difference is the measurements are taken only on the attester side. The fascinating part is for the first execution it takes a little more than other execution. After the second execution on the x-axis, it is visible that each version is leveling off. Further, the line chart with the 200 software selection has one outliner, which has a small effect for calculating the average execution time.

A.5.3 Verifier

Figure A.25 expresses the measurement for the average execution for verifying the TPM's signature, the nonce, the NIZK, and the TPM Quote. Figure A.26 shows how long on average the verifier takes to create an attestation request.

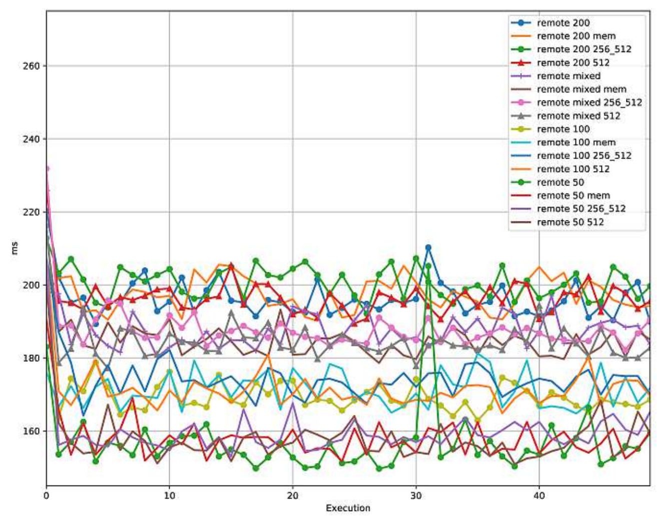

Figure A.21: Comparing different versions of the attestation process, illustrated as line chart (Docker).

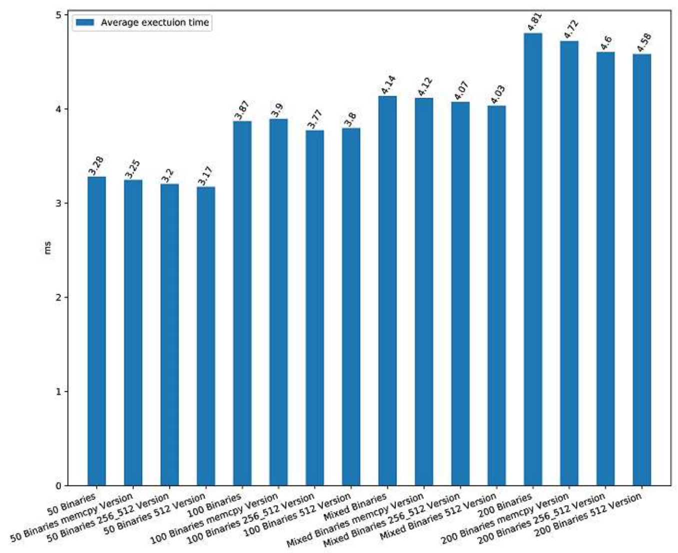

Figure A.22: Comparing average execution time of different versions on attester side (Docker).

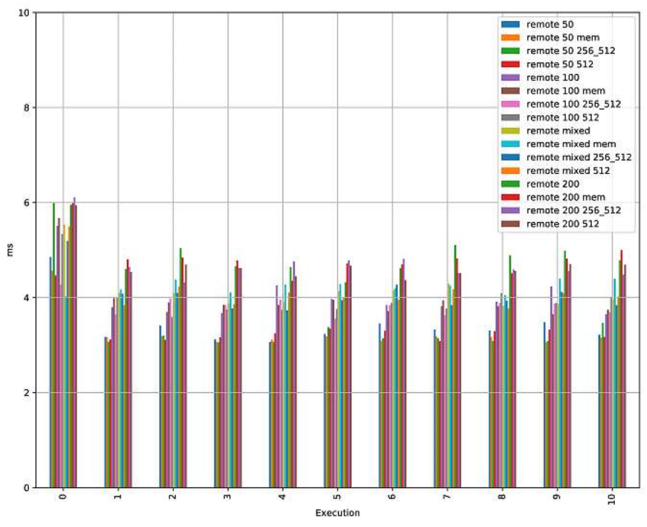

Figure A.23: Comparing different versions on the attester side (Docker).

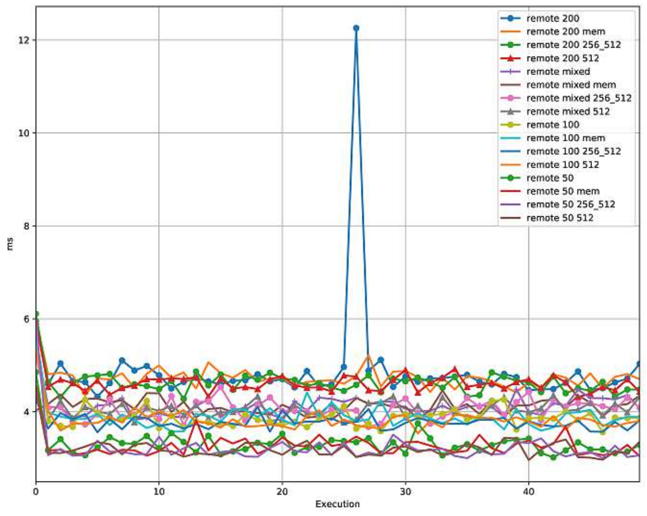

Figure A.24: Comparing different versions on the attester side, illustrated as line chart (Docker).

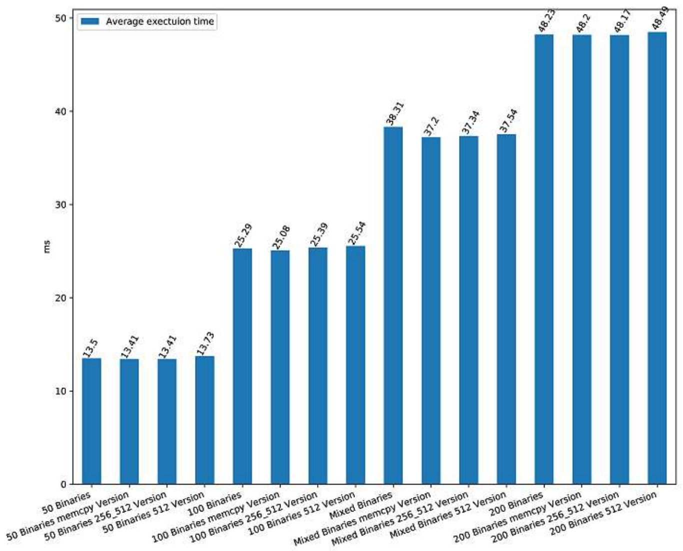

Figure A.25: Comparing average execution time of the different versions on the verification process (Docker).

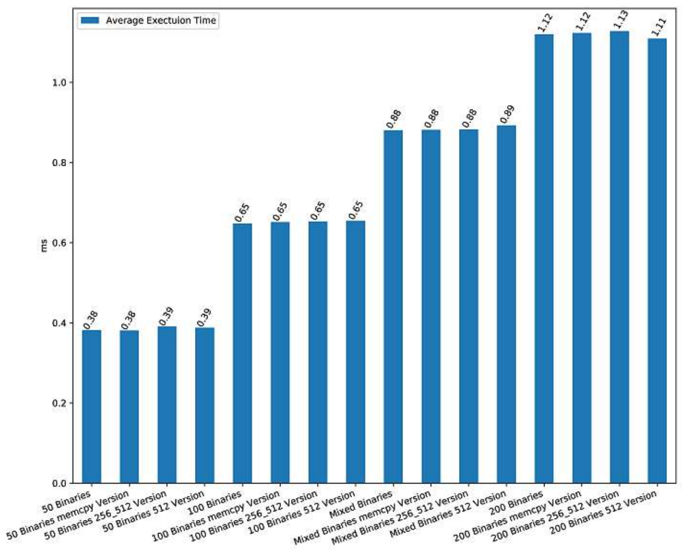

Figure A.26: Comparing average execution time of the different versions on attestation creation (Docker).

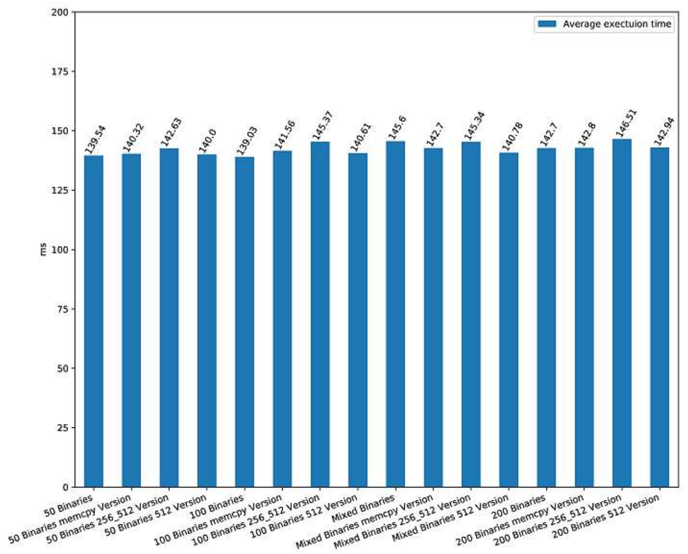

Figure A.27: Comparing average execution time of the different versions on communication duration (Docker).

A.5.4 Communication

The bar plot Figure A.27 shows the average execution time for the communication process between two parties during the attestation process. Further, the number of software selections during the attestation process does not affect. The communication measurement contains the socket establishment, TLS handshake, mutual authentication, and data transfer from both parties. In the case of the communication time, the Figure A.28 shows each execution time during 50 iterations between each version of the poof of concept implementation for 50 and 200 software selection variants. Further, Figure A.28 has a zigzag pattern, which does not level off. The zigzag pattern could be the reason if the socket is also listing to other ports.

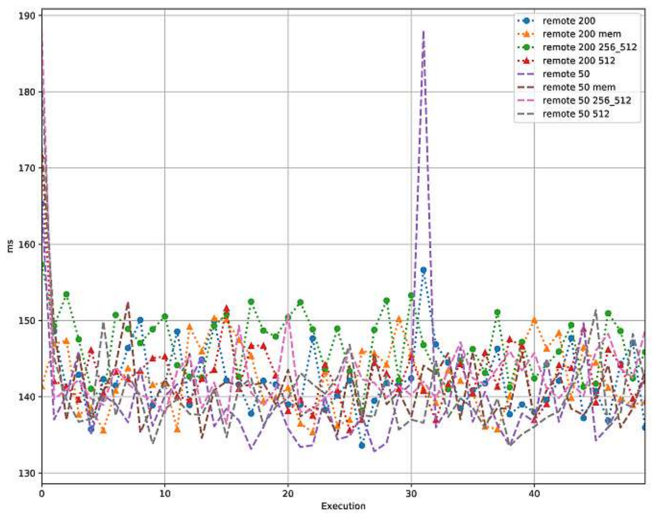

Figure A.28: Comparing different versions of the communication duration (Docker).

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

AK	Attestation Key
BA	Binary Attestation
CA	Certification Authority
CBOR	Concise Binary Object Representation
CDDL	Concise Data Definition Language
CRTM	Core Root of Trust for Measurement
EC	Elliptic Curve
ECC	Elliptic Curve Cryptrography
ECDLP	Elliptic Curve Discrete Logarithm Problem
EK	Endorsment Key
EMD	Electronic Medical Devices
HSM	Hardware Security Module
IAEA	International Atomic Energy Agency
IMA	Integrity Measurement Architecture
IoT	Internet of Things
NIZK	Non-Interactive Zero-Knowledge
PBA	Property-based Attestation
PCR	Platform Configuration Register
PK	Platform Key
PKI	Public Key Infrastructure
PoC	Proof-of-Concept
PPRA	Privacy-Preserving Remote Attestation
PPT	Probabilistic Polynomial Time
PUF	Physical Unclonable Function
RA	Remote Attestation

RIM	Reference Integrity Measurement
RPI3	Raspberry Pi 3 Model B V1.2
SK	Storage Key
SML	Stored Measurement Log
TCG	Trusted Computing Group
TPM	Trusted Platform Module
TTP	Trusted Third Party
ZK	Zero-Knowledge

Bibliography

[1] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474. DOI: `10.1109/SP.2014.36`.

[2] Stefan Berger. *Software TPM*. en. URL: `https://github.com/stefanberger/swtpm` (visited on 2021-04-18).

[3] Daniel J. Bernstein. “Curve25519: New Diffie-Hellman Speed Records”. en. In: *Public Key Cryptography - PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. DOI: `10.1007/11745853_14`. URL: `http://link.springer.com/10.1007/11745853_14` (visited on 2021-04-08).

[4] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. “High-Speed High-Security Signatures”. In: *CHES*. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 124–142. DOI: `10.1007/978-3-642-23951-9_9`. URL: `https://www.iacr.org/archive/ches2011/69170125/69170125.pdf`.

[5] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. *Elligator: Elliptic-curve points indistinguishable from uniform random strings*. Tech. rep. 325. 2013. URL: `http://eprint.iacr.org/2013/325` (visited on 2021-04-09).

[6] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. *Elligator: Elliptic-curve points indistinguishable from uniform random strings*. Cryptology ePrint Archive, Report 2013/325. `https://eprint.iacr.org/2013/325`. 2013.

[7] Daniel J. Bernstein and Tanja Lange. *Indistinguishability from uniform random strings*. en. URL: `https://safecurves.cr.yp.to/ind.html` (visited on 2021-04-07).

[8] Daniel J. Bernstein and Tanja Lange. *Ladders*. en. URL: `https://safecurves.cr.yp.to/ladder.html` (visited on 2021-04-07).

[9] Daniel J. Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. en. URL: https://safecurves.cr.yp.to (visited on 2021-04-07).

[10] Henk Birkholz, Christoph Vigano, and Carsten Bormann. *Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures*. RFC 8610. 2019-06. DOI: 10.17487/RFC8610. URL: https://rfc-editor.org/rfc/rfc8610.txt.

[11] M BLUM, A DE SANTIS, S MICALI, and G PERSIANO. "Noninteractive zero-knowledge". English. In: *SIAM journal on computing (Print)* (1991). ISSN: 0097-5397.

[12] Carsten Bormann. *CBOR — Concise Binary Object Representation | Overview*. URL: https://cbor.io/ (visited on 2021-04-11).

[13] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949. 2020-12. DOI: 10.17487/RFC8949. URL: https://rfc-editor.org/rfc/rfc8949.txt.

[14] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. *The Provable Security of Ed25519: Theory and Practice*. Tech. rep. 823. 2020. URL: http://eprint.iacr.org/2020/823 (visited on 2021-04-08).

[15] Greg Burd. *duration.h*. (Accessed on 23/02/2021). 2015. URL: https://gist.github.com/gburd/5469493#file-duration-h (visited on 2021-02-23).

[16] *Certificate-Policies*. URL: https://www.openssl.org/docs/manmaster/man5/x509v3_config.html#Certificate-Policies (visited on 2021-04-21).

[17] Ioannis Chatzigiannakis, Apostolos Pyrgelis, Paul G. Spirakis, and Yannis C. Stamatiou. "Elliptic Curve Based Zero Knowledge Proofs and their Applicability on Resource Constrained Devices". In: *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*. 2011, pp. 715–720. DOI: 10.1109/MASS.2011.77.

[18] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. "A Protocol for Property-Based Attestation". In: *Proceedings of the First ACM Workshop on Scalable Trusted Computing*. STC '06. Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 7–16. DOI: 10.1145/1179474.1179479. URL: https://doi.org/10.1145/1179474.1179479.

[19] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. "Property-Based Attestation without a Trusted Third Party". In: *Proceedings of the 11th International Conference on Information Security*. ISC '08. Taipei, Taiwan: Springer-Verlag, 2008, pp. 31–46. DOI: 10.1007/978-3-540-85886-7_3. URL: https://doi.org/10.1007/978-3-540-85886-7_3.

[20] Craig Costello and Benjamin Smith. *Montgomery curves and their arithmetic: The case of large characteristic fields*. Tech. rep. 212. 2017. URL: http://eprint.iacr.org/2017/212 (visited on 2021-04-23).

[21] Frank Denis. *Introduction*. URL: https://libsodium.gitbook.io/doc/ (visited on 2021-04-10).

[22] Frank Denis. *Ristretto*. URL: https://libsodium.gitbook.io/doc/advanced/point-arithmetic/ristretto (visited on 2021-04-10).

[23] Frank Denis. *Usage*. 2021. URL: https://doc.libsodium.org/usage (visited on 2021-04-18).

[24] Paul J. Drongowski. *Performance counter kernel module | Sand, software and sound*. en-US. 2013. URL: http://sandsoftwaresound.net/raspberry-pi/raspberry-pi-gen-1/performance-counter-kernel-module/ (visited on 2021-04-18).

[25] Michael Eckel. *Lecture notes in Hardware-Based System Security*. 2020.

[26] Michael Eckel and Björn Grohmann. "Contrained Disclosure of Sensitive System Operational State Information". unpublished. 2016.

[27] Pierre-Alain Fouque and Jean-Christophe Zapalowicz. "Statistical Properties of Short RSA Distribution and Their Cryptographic Applications". en. In: *Computing and Combinatorics*. Ed. by Zhipeng Cai, Alex Zelikovsky, and Anu Bourgeois. Vol. 8591. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 525–536. DOI: 10.1007/978-3-319-08783-2_45. URL: http://link.springer.com/10.1007/978-3-319-08783-2_45 (visited on 2021-04-12).

[28] Abdreas Fuchs and Tadeusz Struk. *tpm2-software/tpm2-tss*. original-date: 2015-06-30T16:21:57Z. 2021-04. URL: https://github.com/tpm2-software/tpm2-tss (visited on 2021-04-10).

[29] Torbjörn Granlund. *The GNU MP Bignum Library*. URL: https://gmplib.org/ (visited on 2021-04-10).

[30] Joshua Guttman, Amy Herzog, Jon Millen, Leonard Monk, John Ramsdell, Justin Sheehy, Brian Sniffen, G Coker, and P Loscocco. "Attestation: Evidence and Trust". In: *MTR080072 MITRE TECHNICAL REPORT, Center for Integrated Intelligence Systems Bedford*. 2008.

[31] Hala Hamadeh and Akhilesh Tyagi. "An FPGA Implementation of Privacy Preserving Data Provenance Model Based on PUF for Secure Internet of Things". In: *SN Comput. Sci.* 2.1 (2021), p. 65. DOI: 10.1007/s42979-020-00428-0. URL: https://doi.org/10.1007/s42979-020-00428-0.

[32] Mike Hamburg. *Decaf: Eliminating cofactors through point compression*. Tech. rep. 673. 2015. URL: http://eprint.iacr.org/2015/673 (visited on 2021-04-09).

[33] Feng Hao. *Schnorr Non-interactive Zero-Knowledge Proof*. RFC 8235. 2017-09. DOI: 10.17487/RFC8235. URL: https://rfc-editor.org/rfc/rfc8235.txt.

[34] *IAEA Safeguards Serving Nuclear Non-Proliferation*. Tech. rep. 2015. URL: `https://www.iaea.org/sites/default/files/safeguards_web_june_2015_1.pdf` (visited on 2021-04-21).

[35] *Inserting Custom OIDs into OpenSSL*. URL: `https://knowledge.digicert.com/quovadis/ssl-certificates/csr-generation/inserting-custom-oids-into-openssl.html` (visited on 2021-04-21).

[36] L. S. Jie and H. Y. Ping. "A Privacy-Preserving Integrity Measurement Architecture". In: *2010 Third International Symposium on Electronic Commerce and Security*. 2010, pp. 242–246. DOI: `10.1109/ISECS.2010.60`.

[37] Simon Josefsson and Jim Schaad. *Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure*. RFC 8410. 2018-08. DOI: `10.17487/RFC8410`. URL: `https://rfc-editor.org/rfc/rfc8410.txt`.

[38] Dmitry Kasatkin. *Integrity Measurement Architecture (IMA) / Wiki / Home*. URL: `https://sourceforge.net/p/linux-ima/wiki/Home/` (visited on 2021-04-26).

[39] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[40] Florian Kohnhäuser. "Advanced Remote Attestation Protocols for Embedded Systems". PhD thesis. Darmstadt University of Technology, Germany, 2019. URL: `http://tuprints.ulb.tu-darmstadt.de/8998/`.

[41] Christoph Krauß. *Lecture notes in Automotive Security*. 2020.

[42] Wu Luo, Wei Liu, Yang Luo, Anbang Ruan, Qingni Shen, and Zhonghai Wu. "Partial Attestation: Towards Cost-Effective and Privacy-Preserving Remote Attestations". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. ISSN: 2324-9013. 2016-08, pp. 152–159. DOI: `10.1109/TrustCom.2016.0058`.

[43] Wu Luo, Qingni Shen, Yutang Xia, and Zhonghai Wu. "Container-IMA: A privacy-preserving Integrity Measurement Architecture for Containers". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, 2019-09, pp. 487–500. URL: `https://www.usenix.org/conference/raid2019/presentation/luo`.

[44] Arcus Matthew. *Using the Cycle Counter Registers on the Raspberry Pi 3*. en. 2018-01. URL: `https://matthewarcus.wordpress.com/2018/01/27/using-the-cycle-counter-registers-on-the-raspberry-pi-3/` (visited on 2021-04-18).

[45] Eckel Michael, Richardson Michael, Roberts William, and Uiterwijk Patrick. *CHARRA: CHAllenge-Response based Remote Attestation with TPM 2.0*. 2019. URL: `https://github.com/Fraunhofer-SIT/charra` (visited on 2020-10-06).

[46] Hamburg Mike, Henry de Valence, Lovecruft Isis, and Arcieri Tony. *Ristretto - The Ristretto Group*. URL: https://ristretto.group/ristretto.html (visited on 2021-04-09).

[47] Victor S. Miller. "Use of elliptic curves in cryptography". In: *Conference on the theory and application of cryptographic techniques*. Springer, 1985, pp. 417–426.

[48] Intan Muchtadi-Alamsyah and Yanuar Bhakti Wira Tama. "Implementation of Elliptic Curve25519 in Cryptography". en. In: *Theorizing STEM Education in the 21st Century*. Ed. by Kehdinga George Fomunyam. IntechOpen, 2020-02. URL: https://www.intechopen.com/books/theorizing-stem-education-in-the-21st-century/implementation-of-elliptic-curve25519-in-cryptography (visited on 2021-04-07).

[49] Niels Möller. *Nettle / nettle*. en. URL: https://git.lysator.liu.se/nettle/nettle (visited on 2021-04-10).

[50] Svetlin Nakov. *Practical Cryptography for Developers*. 2018-11. ISBN: 978-619-00-0870-5. URL: https://cryptobook.nakov.com/.

[51] Gabriele Paoloni. "How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures". In: *Intel Corporation* 123 (2010).

[52] Trevor Perrin. *noiseprotocol/noise_spec*. en. 2016. URL: https://github.com/noiseprotocol/noise_spec (visited on 2021-04-20).

[53] Trevor Perrin. "The Noise Protocol Framework". en. In: (2018-07), p. 65.

[54] Lukas Prokop. "Elliptic Curve Cryptography: Theory for EdDSA". In: (2014). URL: https://lukas-prokop.at/proj/eddsa/static/curves/curves.pdf.

[55] Ahmad-Reza Sadeghi and Christian Stüble. "Property-Based Attestation for Computing Platforms: Caring about Properties, Not Mechanisms". In: NSPW '04. Nova Scotia, Canada: Association for Computing Machinery, 2004, pp. 67–77. DOI: 10.1145/1065907.1066038. URL: https://doi.org/10.1145/1065907.1066038.

[56] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. "Design and Implementation of a TCG-Based Integrity Measurement Architecture". In: *Proceedings of the 13th Conference on USENIX Security Symposium*. Vol. 13. SSYM'04. San Diego, CA, USA: USENIX Association, 2004, p. 16. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/design-and-implementation-tcg-based-integrity-measurement.

[57] Ioannis Sfyrakis and Thomas Gross. *A Survey on Hardware Approaches for Remote Attestation in Network Infrastructures*. 2020. arXiv: 2005.12453 [cs.CR].

[58] Susmit Sil. *Implementing Mutual SSL Authentication*. en. 2018-01. URL: https://blog.cloudboost.io/implementing-mutual-ssl-authentication-fc20ab2392b3 (visited on 2021-04-13).

[59] *SLB 9670VQ2.0 - Infineon Technologies*. AG, Infineon Technologies. URL: https://www.infineon.com/cms/en/product/security-smart-card-solutions/optiga-embedded-security-solutions/optiga-tpm/slb-9670vq2.0/ (visited on 2021-04-18).

[60] *SSL Library mbed TLS / PolarSSL*. ARM Limited. URL: https://tls.mbed.org/ (visited on 2021-04-13).

[61] William Stallings. *Cryptography and Network Security (4th Edition)*. USA: Prentice-Hall, Inc., 2005. ISBN: 0131873164.

[62] Wencheng Sun, Zhiping Cai, Yangyang Li, Fang Liu, Shengqun Fang, and Guoyan Wang. “Security and Privacy in the Medical Internet of Things: A Review”. In: *Security and Communication Networks* 2018 (2018), p. 5978636. ISSN: 1939-0114. DOI: 10.1155/2018/5978636. URL: https://doi.org/10.1155/2018/5978636.

[63] *TPM2.0 Mobile Reference Architecture*. Family 2.0, Level 00, Revision 142. Trusted Computing Group. 2014-12.

[64] *Trusted Platform Module Library - Part 1: Architecture*. Family 2.0, Level 00, Revision 01.59. Trusted Computing Group. 2019-11.

[65] *Trusted Platform Module Library - Part 3: Commands*. Family 2.0, Level 00, Revision 01.59. Trusted Computing Group. 2019-11.

[66] Henry de Valence, Jack Grigg, George Tankersley, Filippo Valsorda, and Isis Lovecruft. *The ristretto255 Group*. Internet-Draft draft-hdevalence-cfrg-ristretto-01. Work in Progress. Internet Engineering Task Force, 2019-05. 16 pp. URL: https://datatracker.ietf.org/doc/html/draft-hdevalence-cfrg-ristretto-01.

[67] Henry de Valence, Jack Grigg, George Tankersley, Filippo Valsorda, Isis Lovecruft, and Mike Hamburg. *The ristretto255 and decaf448 Groups*. Internet-Draft draft-irtf-cfrg-ristretto255-decaf448-00. Work in Progress. Internet Engineering Task Force, 2020-10. 26 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-00.

[68] D. Wood. “ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER”. In: 2014.

[69] Huixin Wu and Feng Wang. “A Survey of Noninteractive Zero Knowledge Proof System and Its Applications”. In: *TheScientificWorldJournal* 2014 (2014-05), p. 560484. DOI: 10.1155/2014/560484.

[70] Z. Xing-lan and L. Wei-wei. “Improving Privacy of Property-based Attestation without a Trusted Third Party”. In: *2011 Seventh International Conference on Computational Intelligence and Security*. 2011, pp. 559–563. DOI: 10.1109/CIS.2011.129.

[71] Yongxiong Zhang, Liangming Wang, Yucong You, and Luxia Yi. “A Remote-Attestation-Based Extended Hash Algorithm for Privacy Protection”. In: *2017 International Conference on Computer Network, Electronic and Automation (ICC-NEA)*. 2017-09, pp. 254–257. DOI: `10.1109/ICCNEA.2017.60`.

[72] Mimi Zohar, David Safford, and Reiner Sailer. *Using IMA for Integrity Measurement and Attestation*. Portland, OR, USA, 2009-09. URL: `https://blog.linuxplumbersconf.org/2009/slides/David-Stafford-IMA_LPC.pdf`.