# PATAT: An Open Source Attestation Mechanism for Trusted Execution Environments on TrustZone devices

Frank Nijeboer

May 27, 2024

**Abstract** As technology evolves, secure computing environments become increasingly critical. Arm TrustZone, a hardware-based security extension for Cortex processors, provides a trusted environment for applications requiring high levels of confidentiality and integrity. However, effective attestation mechanisms to verify the integrity of TrustZone applications have not been standardized yet.

In this research, we investigate the state of attestation mechanisms for Arm TrustZone and propose a novel mechanism, called PATAT, based on Merkle Trees — a data structure for secure data verification — and the Noise Protocol Framework, a framework for building cryptographic handshakes for secure communication. PATAT is designed to fit the Remote Attestation and Trusted Systems (RATS) Architecture, an architecture for attestation mechanisms defined in RFC 9334. We formally verify PATAT using the Tamarin prover, a tool for security protocol analysis, and implement a proof-of-concept to evaluate its performance.

**Keywords** attestation · TrustZone · Arm · security · cryptographic protocols · Tamarin

## 1 Introduction

In today's digital era, the rise of connected devices and the Internet of things (IoT) has led to an explosion of data being generated at the edge of the network. This has resulted in a significant shift towards distributed architectures that leverage the power of edge computing, enabling data to be processed near its source for faster response times and reduced latency [1]. As a result, the field of edge computing is expected to grow rapidly in the near future [2]. Arm chips such as the Cortex-A family [3] are often used in these edge devices due to their cost effectiveness and power efficiency. However, as the number of edge devices continues to grow, so does the risk of cyber threats, making security a critical concern for developers and manufacturers.

Hardware-based security mechanisms like Trusted Execution Environments (TEEs), have emerged as a solution to address security concerns in edge computing environments [4]. TEEs provide an isolated execution environment, separate from the operating system, which allows sensitive applications to run securely. TEEs can also run on Arm chips that feature the TrustZone technology [3] [5]. However, while the concept of attestation, where a (remote) party can verify the integrity of the code running in a TEE, is well-established in alternative technologies like Intel SGX, an open-source attestation implementation within TrustZone has yet to emerge, despite TrustZone being available since 2004 [4].

This work explores the state of attestation mechanisms for TrustZone in Section 5. This is then followed by Section 6 where we present PATAT: a novel attestation mechanism for TrustZone. PATAT is based on the concept of Merkle Trees and the Noise Protocol Framework with the aim of being easy to implement. This novel attestation mechanism is then accompanied by a formal proof using the Tamarin Prover in Section 7.1 and a proof-of-concept implementation in Section 7.2 and practical examination in Section 7.3.

This work's use case originated from Scalys: the developers of TrustBox [6], a networking device designed for use in zero-trust environments, powered by an NXP chip that features the Arm TrustZone [7]. These devices operate in various untrusted environments, necessitating additional security measures. While attestation can provide this security, the lack of a standardized attestation mechanism in TrustZone necessitates this research.

# 2 Background

This section is a brief introduction to some topics which are relevant to this research. In Section 2.1, we provide a high level overview of the TrustZone technology as well as some information about the computer chips which are equipped with TrustZone. Section 2.2 discusses the concept of a TEE. Then Section 2.3, introduces the concept of attestation. Section 2.4 discusses the set of features that distinguishes attestation mechanisms. Then, Section 2.5 describes cryptographic principles that are used in this work. Finally, Section 2.6 introduces the formal verification tool which we use in this work.

## 2.1 Arm TrustZone

The Arm TrustZone is a System-on-Chip (SoC) security solution, which is available on most devices with an Arm chip nowadays. It has already been available since 2004, but only in recent years, with the industry becoming more security focused has it seen more widespread adoption [4].

The TrustZone technology centers around the idea that there are two domains in the computer that must be isolated from each other: the *secure world* and the *normal world*. TrustZone not only controls the CPU state, but the concept of being *secure* or *normal* also extends to other parts of the SoC, such as peripherals and buses (the communication lanes between the parts of the SoC). Arm TrustZone provides hardware isolation between the *Secure* and *Normal* worlds. The processor's current state is determined by the Non-Secure (`NS`) bit that Arm has added to the processors. The *secure world* consists of everything that runs when the processor is in the secure (`S`) state and the *normal world* of everything when the state is `NS`. The hardware is created in such a way that the *normal world* is incapable of accessing the regions of the SoC that are considered *secure* [8] and the processor can only work in one of the states at a time. Switching between *secure* and *normal* depends on the type of chip that is used. This can be either a Cortex-A [3] or Cortex-M [5] chip . Since this research will mainly focus on the Cortex-A, the Cortex-M TrustZone will not be explained in-depth. Lastly, it is important to note for this research that the Arm TrustZone itself lacks any form of attestation.

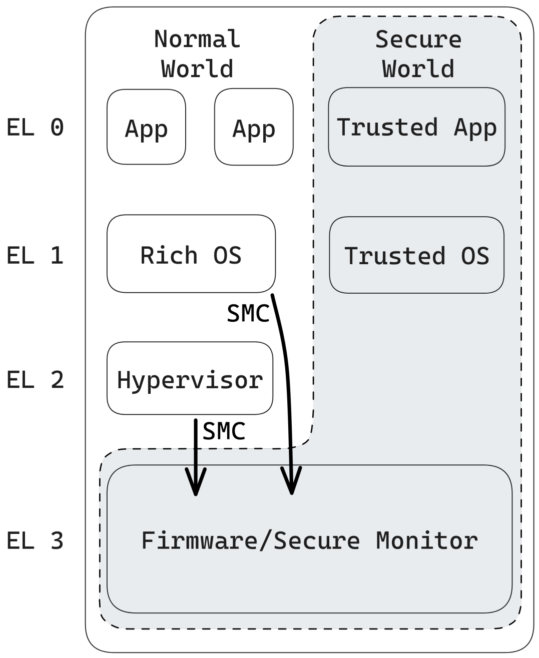


Figure 1: TrustZone in Cortex-A SoCs, adapted from [8]

### 2.1.1 TrustZone for Cortex-A

Cortex-A processors are made for devices that feature a full Operating System [3]. In this SoC, the *secure monitor* software provides the context switching functionality between worlds in the TrustZone. Code from both the Secure and the Non-Secure worlds can switch between the processor state by using the `smc` exception. Thus changing the state of the `NS` bit. This is shown in a schematic way in Figure 1.

Within these systems, the secure world extends beyond the processor: also the memory can be in either a *secure* or *non-secure* state. The TrustZone Address Space Controller (TZASC) handles this for DRAM and the TrustZone Memory Adapter (TZMA) does this for the SRAM and ROM [8].

## 2.2 Trusted Execution Environment

GlobalPlatform, a standardization organization, was the first to come up with the term Trusted Execution Environment (TEE) [9] [10]. They also create standards which developers and manufacturers of devices with a TEE can use by providing Application Programming Interfaces (APIs) and device specifications [4].

GlobalPlatform's definition of a TEE is "a secure area of the main processor of a connected device that ensures sensitive data is stored, processed and protected in an isolated and trusted environment" [11]. In another paper by Sabt, Achemlal and Bouabdallah [9] about the definition of a TEE, the authors denote a Trusted Execution Environment as a tamper-resistant processing environment with guarantees about: authenticity of the executed code, integrity of the runtime states, and confidentiality of data and state of the system.

Furthermore, they state that the content in a TEE should not be static but should allow for secure updates [9]. In a survey about TrustZone, Pinto and Santos [4] dedicate a significant portion to TEEs and their ability to address issues arising from the complexity of large (bloated) Trusted Computing Bases (TCBs). They explain that a TEE can host critical applications in an environment that is smaller and more secure than a Rich OS serving as a TCB. They define a TEE as "an isolated environment in which *Trusted Application (TA)s* can execute without the interference of the local (untrusted) OS" [4]. A *TA* is software that runs inside a TEE. These *Trusted Applications* are isolated from each other and from the Rich Execution Environment (REE), which only runs regular applications.

As seen above, the definitions of TEEs can vary in the details. For example, from GlobalPlatform's definition, it would seem as only the processor is placed in a secure state when entering the TEE. But, Pinto and Santos define a TEE as an *environment* and, as such, the peripherals and memory are also in a secure state inside the TEE. In this research, we will follow the latter definition, due to our focus on Arm devices which support secure states for peripherals and memory, as Section 2.1 shows.

An example of a TEE for TrustZone is OP-TEE (Open Portable Trusted Execution Environment) [12]. The Arm TrustZone itself is not a TEE, but the TrustZone enables the creation of a TEE through the isolation, authenticity, and integrity features that it provides. Examples of TEEs for other architectures are: Intel SGX [13] for Intel chips, and Multizone for RISC-V [14].

## 2.3 Attestation

Trusted Execution Environments make sure that the code and data inside the TEE cannot be reached from the regular OS. However, this does not provide safeguards against all attacks; attackers with physical access to the device can tamper with memory on the device and change data or firmware on the device. Furthermore, remote attackers can abuse flaws in device software, such as stack-buffer overflows and command injection [15] [16], to overwrite the code on the device to run malicious applications. Since the contents of a TEE cannot be observed from the outside, changes in the TEE cannot be detected.

Attestation mechanisms aim to prevent such attacks from succeeding. With an attestation architecture and attestation mechanisms, one party can prove to another party that it is running trusted (attested) software and that it has not been subject to any other form of tampering. Attestation mechanisms can aid in putting more trust in the Trusted Execution Environments and likewise, TEEs can strengthen attestation mechanisms [17].

### 2.3.1 Terminology

Existing works on attestation make use of different terminology for the same actors and parts in the attestation architecture. This work will adopt the terminology as described in RFC 9334 by the IETF [18], which Ménétrey et al. also explain in their work [17]. Therefore we have the following definitions for attestation:

**Attester** The actor that proves some properties about itself to another party.

**Claim** A *Claim* is a piece of asserted information, such as a cryptographic hash of the code on the device.

**Evidence** Information which the Attester uses to prove their identity to the Relying Party. It consists of a set of Claims which the Attester has gathered on the system, which are then signed such that the Relying Party can check its validity.

**Relying Party** The actor that (ultimately) decides whether the *Attester* can be trusted.

**Trusted Application** An Attester implementation in the form of an application.

**Verifier** A Verifier appraises a part of the *Evidence* of an *Attester*. They might e.g. be a third-party that appraises a piece of third-party code running on the *Attester* device.

### 2.3.2 Core Concepts for Attestation

As described by Coker et al. [19], attestation mechanisms should consist of building blocks on top of which the attestation mechanism can be built. These concepts are: Types of Evidence, Separate Domains and Trust Base, and they are explained in further detail in the following paragraphs.

**Types of Evidence** The number of evidence types which the *Attester* can use to prove their identity to the *Relying Party (RP)* has no limit. But it is important that the mechanism utilizes useful information during the attestation process. Most existing attestation mechanisms use the compiled application software as one of the *Claims* in some form, usually a hash.

Another example of a trivial type of *Claim* for a running TEE, would be to create a hash over the current memory contents. However, such information would not be useful to an attestation protocol, because this memory may change rapidly and unpredictably depending on the application. For these situations, Coker et al. suggest to use only certain parts of the memory which do not change often [19].

In addition to that, freshness of the total *Evidence* is also a goal according to Coker et al. That means that in addition to performing the measurement before executing a program the *Attester* must deliver *Evidence* as often as possible. With some attestation mechanisms this means that the *Relying Parties* can trigger a new measurement to get fresh measurement information. Ménétrey et al. also use freshness as one of the fields in their comparison between Attestation Mechanisms for TEEs [17].

**Separate Domains** Attestation measurement tooling must be able to provide accurate results about the state of the target, even when the target has been compromised. The measurement tool must have access to the target to be able to determine whether its state is still uncompromised. But the other way around should be impossible. This is to make sure that a compromised target cannot influence the results of the measurement tool.

Coker et al. state that using VMs can be a good way to achieve this separation [19]. They describe a measurement tool which resides in a VM and the target which resides in another VM. The Virtual Machine Monitor should then configure cross-VM visibility such that the measurement VM can inspect the target, and protect the measurement VM from the target. Creating separate TEE enclaves would be an alternative method for separating domains. The measurement tool in this case can be rooted in secure hardware, such as the CPU in the case of Intel SGX [13].

**Trust Base** Having domain separation and reasonable types of *Claims* will not be useful without a base of trust in the system. Without this trust base, the fundamental parts of the attestation architecture cannot be verified. Physical compromise of the hardware may enable attackers to overwrite firmware on the device in order to trick the parts of the attestation architecture into attesting parts of the system which are in fact compromised.

Such a trust base should start in the boot process and it should be hardware enforced. Secure Boot or Authenticated Boot are good options to use as a trust base, and many other works also utilize these technologies [9] [8] [20]. Secure Boot or Authenticated Boot can be offered by hardware vendors for their devices with an Arm chip [8] [21]. This secure boot process makes sure that only verified software/firmware can run on the SoC. Usually, the device verifies the firmware that it wants to boot with a public-key that is specific for a software vendor. The software vendor generates a signature of the firmware binary with their secret key and pushes both the firmware and the signature to the device. The device holds the public key, which should be stored in such a way that it can be read but not replaced by a public key from the attacker. UEFI systems also have a similar secure boot process, which can be enabled [22].

### 2.3.3 Governing Incentives

The realm of Attestation and Trusted Execution Environments (TEEs) has been subject to a number of incentives that aim to standardize the various aspects that are relevant to this area. Two notable examples of such incentives are the aforementioned Confidential Computing Consortium (CCC) [23] and the Internet Engineering Task Force (IETF) with their Remote Attestation Procedures (RATS) [18]. These initiatives aim to provide a common framework and guidelines that facilitate the development, deployment, and interoperability of TEE-based systems.

Despite the efforts of these entities, however, there remain many open questions that must be addressed before suitable standards can be established without the risk of unintended consequences. For example, there is the need to ensure that the standards are flexible enough to accommodate a variety of hardware platforms and use cases, while still providing robust security guarantees. Another area of concern is the question of interoperability between different attestation solutions, particularly as the industry continues to evolve and new technologies emerge. And as such, these standards have not yet been established. This still leaves room for industry and researchers as well as open source initiatives to experiment with attestation.

**RATS Concepts** The Remote Attestation Procedures (RATS) RFC 9334 [18] is a notable RFC which aims to bring concepts about attestation together in a centralized document to aid standardization across attestation mechanisms. Since we adopted the terminology from RATS, the terms can be found in Section 2.3.1. This section will shortly address some concepts within RATS.

RATS contains a dedicated section discussing Topological Patterns, which outline how *Attester*, *RP*, and *Verifier* exchange *Evidence*. These patterns describe two main communication models: the passport model and the background-check model, shown respectively in Figures 2 and 3. In the Passport Model, the *Attester* first sends evidence to a *Verifier* before sending the result to the *RP*. The Background Check Model the *Attester* sends *Evidence* to the *RP* who forwards (some) of the *Evidence* to

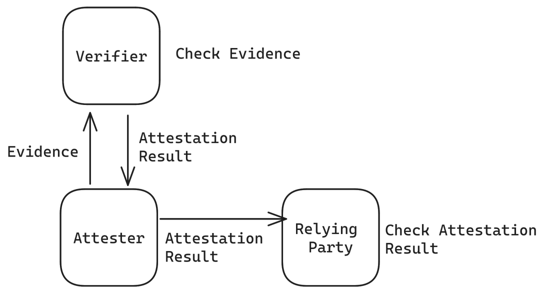


Figure 2: The RATS Passport Model

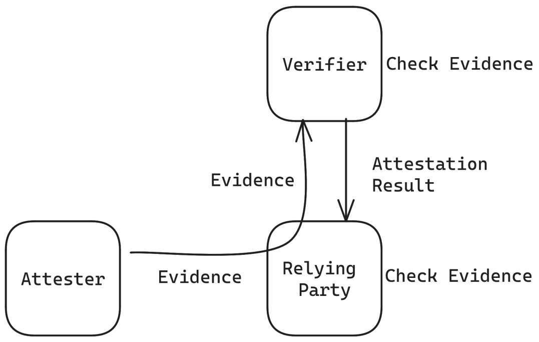


Figure 3: The RATS Background Check Model

the *Verifier*. The RFC also mentions the possibility of combining these models. Additionally, privacy considerations are emphasized for attestation implementations, as not all trust models permit the *Verifier* to trust the *Attester* with sensitive data, and vice versa. This highlights the importance of careful protocol design. At the time of writing there are no TrustZone attestation mechanism works which feature the RATS Procedures in their protocol.

## 2.4 Attestation Features

This section discusses the distinguishing features between attestation mechanisms. Creators of these mechanisms must make decisions about which features will be supported by the mechanism, and some features may or may not make it into the final version of a mechanism. Therefore, implementations of attestation mechanisms will each have their own set of supported features. A comparison between attestation mechanisms can be made by looking at the differences in feature sets. Section 5 in this work provides such a comparison. In this section, we present a list of features that we have defined by examining existing works on attestation mechanisms.

### 2.4.1 Functional Features

The term "Functional Features" refers to the specific functionality of an attestation mechanism that makes it appealing for software manufacturers to incorporate into their systems. These features are closely tied to the overall design of a particular attestation architecture and as such the implementation of Functional Features can vary significantly across different attestation mechanisms.

**Local Attestation (LA)** Local Attestation enables an *Attester* TEE to authenticate its identity to a *RP* TEE residing on the same device. The process can then, for example, be based on a key that is bound to the hardware on which the software runs [24].

Intel SGX is a good example of a local attestation mechanism since local attestation serves as one of the key pillars for SGX's security enclaves [13], and as such, has a well-researched implementation. In SGX, local attestation relies on the fact that both TEEs share the same CPU. This CPU is provisioned with a secret that serves as a message authentication code (MAC) for the attestation mechanism. The *Attester* TEE generates a MAC, which is then checked by the *RP* TEE to validate the identity of the *Attester*.

**Remote Attestation (RA)** Remote attestation is a natural extension of the concept of local attestation, since the latter is typically used to ensure that a device's hardware and software components have not been tampered with. However, local attestation is limited in its scope, as it only provides assurance about the trustworthiness of the local platform.

Remote attestation, on the other hand, enables a remote entity to verify the trustworthiness of a *Trusted Application (TA)*. Remote attestation can provide a higher level of assurance about the trustworthiness of a platform compared to local attestation, as it enables verification from a trusted third party. It can, for example, block execution of code if the remote device is not in a known state, possibly preventing malicious software from running. As an example of this concept, the CoCo project uses remote attestation to get decryption keys for software binaries, preventing them from running without attesting first [25].

Remote- and local attestation are not mutually exclusive. On the contrary: remote- and local attestation can complement each other to create a "chain of trust" that utilizes both mechanisms to achieve the desired level of security in a product. Some remote attestation mechanisms, such as Intel SGX [13], require local attestation as part of the remote attestation procedure.

**Mutual Attestation (MA)** Mechanisms which feature mutual attestation have the capability of doing attestation on both sides of the (remote) connection. Some *Trusted Applications* need this as a stronger assurance of trust.

**Secure Channels (SC)** Attestation mechanisms might have the option to create (or retain) a long lasting secure channel after the attestation was performed. Since this channel was created as part of the attestation, it has a higher level of trust compared to secure connections which are shared only through a secret key. This idea is also explored by using attestation in TLS connections [26]. These secure channels are most useful in cases where Remote Attestation is used, since this secure channel can be used to transfer application data that is not related to the attestation while still having the same level of trust as the attestation.

**Session Resumption (SR)** Mechanisms that provide this functionality do not need to perform the full attestation process each time. Such protocols typically feature a temporary key or "session cookie" that the parties can use to resume their connection without repeating all the steps normally necessary in the attestation process. This key usually expires after a set period, after which a full attestation must occur again. An example of this is detailed in the paper by Shepherd et al. [27], which we explore in more depth in Section 3.1.

**TEE Agnostic (TA)** Between different hardware implementations on which Trusted Execution Environments run, the TCB could differ drastically. The TEE agnostic feature shows the extent to which the attestation mechanism relies on specific hardware features that prevent it from running on a different hardware architecture. Usually, hardware-specific implementations allow for a greater amount of trust in the attestation mechanism, but at the cost of software portability between devices.

**Open Source (OS)** An attestation mechanism that fulfills this feature has an open source implementation. Although this is not a fundamental difference between mechanisms, an open source attestation mechanism may see greater adoption due to it being available to many software manufacturers without a Non-Disclosure Agreement or additional cost.

**Attester Anonymity (AA)** Attestation must strike a balance between privacy and the level of trust required between the *Attester* and the *RP*. If an attestation mechanism uses fewer uniquely identifying features in the *Evidence*, it will be harder to trace the attesting device (and possibly the device's owner). However, this may also result in less convincing *Evidence* for the *RP*.

### 2.4.2 Security Features

In the context of attestation mechanisms, security features refer to the features that impact the security of the scheme being evaluated. These features are used in ensuring the integrity and confidentiality of the attestation process, and help to ensure that attestation mechanisms are resistant to a range of potential attacks which could compromise the trustworthiness of the attestation.

**Message Confidentiality (MC)** To ensure confidentiality of the an attestation, the messages used in the process should all be encrypted. Disclosing information about the *Evidence* can be used by attackers to read potential private information.

**Message Verification (MV)** From a security perspective, message verification can help ensure the authenticity and integrity of messages exchanged during the attestation process, which prevents man-in-the-middle attacks.

**(Mutual) Key Establishment (KE)** This feature determines how the protocol establishes new keys during the attestation mechanism and what kind of key agreement protocol it uses for this.

**Forward Secrecy (FS)** This refers to the property of an attestation scheme that states that sessions in the past are not compromised when an attacker gets access to the session key in another communication session. Attestation mechanisms typically achieve this by using ephemeral session keys. When taken into consideration that devices which use attestation mechanisms could have lifespans of 10 years or more, forward secrecy becomes an important security feature to take into consideration. Lastly, this property is typically only useful in the context of remote attestation.

**Non-Repudiation (NR)** Non-Repudiation is the ability of an attestation mechanism to prevent the sender of a attestation from denying that they ever sent the attestation message. This also means that these attestations should arrive unaltered at their destination, or that receivers can detect modifications to the original messages. Attestation mechanisms can achieve non-repudiation by using digital signatures or other cryptographic techniques.

**Runtime Verification (RV)** Runtime Verification enables an attestation mechanism to monitor and verify the integrity and security of software and firmware components during runtime. This feature is necessary for detecting any tampering that may occur after the initial attestation process has completed. By continuously verifying the runtime environment, attestation mechanisms can ensure that any deviations from the expected behavior are detected. Additionally, runtime verification can also play a role in enabling the attestation mechanism to detect and respond to potential zero-day vulnerabilities. The downside is that runtime verification can be unique per piece of software, and a general solution might not work in each situation. Picking the wrong pieces of *Evidence* for Runtime Verification could even result in incorrect rejection of attestations that should be correct.

**Hardware Verification (HV)** This feature allows an attestation mechanism to detect tampering or unauthorized modifications to the hardware components, as opposed to only detecting software modifications. Usually mechanisms which implement this take some values read from hardware and use those as a piece of *Evidence* during the attestation. In the case that unique hardware features are used, this contradicts the Attester Anonimity feature, which would make this a trade-off, as described in the explanation of that feature.

**Root of Trust (RT)** This is not so much a feature as it is a question of where the root of trust lies in the attestation mechanism. Some attestation mechanisms require a root of trust at the hardware level, such as a unique hardware key or a TPM, whereas others might place their trusted components in kernel modules on the device, which is more software-based.

**Freshness (FR)** Freshness of attestation evidence refers to ensuring that the evidence presented for attestation is recent and not reused. Every attestation protocol should strive for freshness to prevent replay attacks.

## 2.5 Cryptography

This work introduces a novel attestation mechanism for TrustZone. It relies on cryptographic principles such as Merkle Trees, AEAD encryption, and Diffie-Hellman key exchanges. Therefore, this section outlines these concepts to provide the reader with basic information about them.

### 2.5.1 Merkle Trees

A Merkle Tree is a form of Tree-Based Data Structure where each leaf is the hash of its actual value, and each node is the hash of its children, concatenated [28]. They are sometimes also referred to as Hash Trees.

Despite its simplicity, Merkle Trees are useful in many different scenarios; ranging from Blockchain [29], and Digital Signatures [28] to Certificate Transparency Logs [30].

Figure 4 shows a simple example of a Merkle Tree. The leafs of the tree are the hashes of $A$, $B$, $C$ and $D$ ($H_{\{A,B,C,D\}}$). Then, those

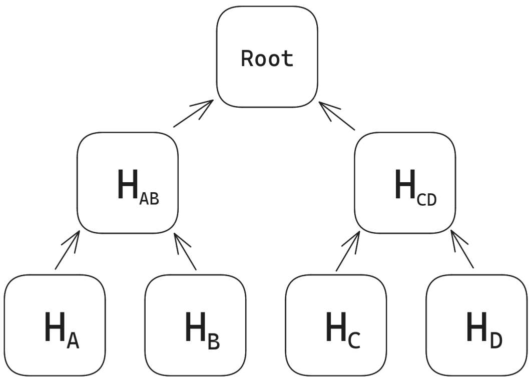


Figure 4: A simple Merkle Tree

hashes are concatenated in their parent node and the concatenation is hashed as well (e.g. $H_{AB} = H(H_A|H_B)$). Finally, these parents are also concatenated and hashed in the root of the tree. The idea is that when a single leaf in the tree changes, the root of the tree will also change. Therefore, inconsistensies in the leaf data will result in a different root.

**Merkle Proof** A Merkle Proof can be utilized to establish the inclusion of a specific value within a Merkle Tree root. In the example proof in Figure 5, the objective is to demonstrate that $A$ is part of the tree, and the prover has already transmitted the Merkle Tree root to the verifier. The prover must subsequently transmit $H_B$ and $H_{CD}$ to the verifier. Now they can independently compute $H_A$ and $H_{AB}$ since $H_B$ has been provided. Subsequently, by hashing $H_{AB}$ and the transmitted $H_{CD}$, the verifier can compute the root of the tree and check if it matches to the previously received root.

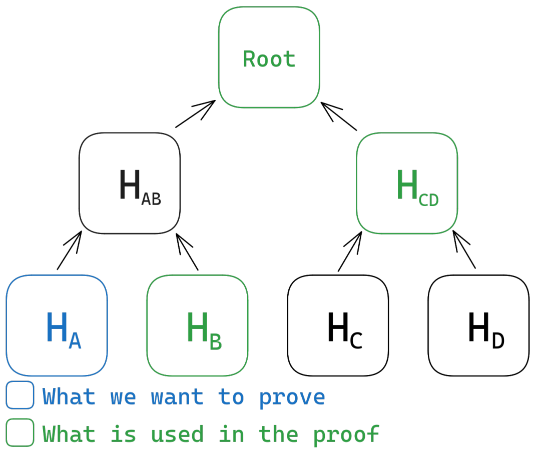


Figure 5: A simplified Merkle Tree Proof

Notice that the prover is not necessarily required to sent the plaintext value of $A$ to the verifier. If the verifier knows the expected value $A$, they can calculate the hash of $A$ themselves.

### 2.5.2 Diffie-Hellman

One of the key components of secure communication is the exchange of cryptographic keys between parties. The Diffie-Hellman key exchange algorithm is a function which can be used by 2 parties to securely arrive at a shared key over an insecure channel. Each party generates its own secret value and uses it along with a public value to compute a shared secret. Even if an eavesdropper intercepts the public values, they cannot easily derive the shared secret without knowing the parties' secret values. This shared secret can then be used for secure communication or as a basis for further cryptographic operations. Diffie-Hellman key exchanges are built on a variant of the discrete logarithm problem [31]. A simple example of a Diffie-Hellman exchange is a communication between Alice and Bob, where they publicly agree to use modulus $p$ and base $g$. Then Alice chooses the secret value $a$ and Bob chooses $b$. Alice sends $g^a \bmod p$ to Bob, who sends $g^b \bmod p$ to Alice. Alice computes the secret key as $(g^a)^b \bmod p = g^{ab}$ and Bob as $(g^b)^a \bmod p = g^{ba} \bmod p$ and since $g^{ab} \bmod p = g^{ba} \bmod p$, they now have a shared secret key. If the values are chosen sufficiently large, it is very hard to compute the secret key for an eavesdropper.

### 2.5.3 AEAD Encryption

AEAD encryption, which stands for Authenticated Encryption with Associated Data, is a cryptographic primitive that provides both confidentiality and integrity for data. Each message encrypted with Authenticated Encryption (AE) contains an Authentication Tag which the receiver can use to verify the message on decryption. To get from AE to AEAD encryption, an AE scheme also needs to provide the option to send Associated Data. This data is not confidential, but it is protected against tampering by an attacker. The receiver of an AEAD message can decrypt the confidential data and verify the integrity of the Associated Data with the same key [32]. Examples of AEAD encryption algorithms are ChaCha20Poly1305 [33] and AES-GCM [34].

## 2.6 Tamarin Prover

In addition to creating a new protocol, this work also introduces a method for formally verifying the protocol using the Tamarin Prover. The Tamarin Prover [35] is a security protocol verification tool that "supports falsification and unbounded verification of security protocols in the symbolic model" [36]. It was developed by David Basin and Cas Cremers, who also created the Scyther tool [37] which is used in other research involving attestatino mechanisms in TrustZone [27]. This section aims to provide the reader with a basic understanding of the concepts within the Tamarin Prover.

In the Tamarin Prover, the state of the system, e.g. encryption keys and the current phase of a secure handshake, is modeled as a collection of *facts*. This collection of facts, the *multiset*, can be modified using *multiset rewriting rules*. In the Tamarin Prover, a security protocol is represented as a set of multiset rewriting rules. These rules take a certain subset of the facts in the system as input and produce a new set of facts as output.

Tamarin provides built-in facts for generating fresh values and sending data over the network. Generating a fresh value is done through the $Fr()$ fact. Interactions with the network are represented by the $Out(x)$ and $In(y)$ facts, which model the sending of data $x$ out to the network and the receiving of data $y$ from the network. By default, the network operates under the assumption of the Dolev-Yao adversary model. This means that the adversary is presumed to have the capability to intercept, alter, and read all messages exchanged between parties. However, they cannot break the cryptography used in the messages. In general, facts can only be consumed once, but in addition to regular facts Tamarin also knows persistent facts. These can be consumed an arbitrary number of times.

In addition to the facts and rules, Tamarin also allows for the use of functions which let users model concepts like encryption and hash functions. The built-in functions used in this work are *hashing* and *diffie-hellman*.

The previous concepts form the basics required to formulate a *lemma*. Lemmas are used to prove properties desired in the overall protocol. They are typically formulated by defining variables and points in time, indicated by a leading #, followed by specific action facts. An example is the following lemma:

```
lemma my_lemma:
        "
        All x y #m #n #o.
        Action1(x) @m
        & Action2(y) @n
        ==>
        Action3(x, y) @o
        "
```

Everything preceding the arrow represents our assumptions regarding the action facts that have occurred, while everything following the arrow signifies the conclusion we want the Tamarin Prover to prove. In this case, we want to prove that if both `Action1` and `Action2` have occurred, then `Action3` must also have occurred with `x` and `y` as input. If the assumption turns out to be incorrect, Tamarin can create a graphical representation of a counterexample.

With the rules and facts, a user of Tamarin can create a model of their envisioned protocol. Using lemmas, they can then prove properties they wish to verify in the protocol. The `tamarin-prover` command-line tool can be instructed to verify these lemmas using the `--prove` argument followed by the path to the file containing the model description.

## 3 Related Work

In the following section, we will discuss other works that have explored attestation in the context of the Arm TrustZone and created their own protocols. Section 5 provides a comparison between these protocols, including the ones discussed here. These two works are highlighted because they, like this work, provide a formal proof for their attestation mechanisms. This section aims to give the reader a more in-depth understanding of how other works go about designing their protocols.

### 3.1 Shepherd et al.

Shepherd et al. [27] address the challenge of trusting data from unattended sensing devices. In their work they present a new trusted channel protocol for performing attestation and setting up a secure channel between two remote TEEs on devices for use in trusted sensing devices (e.g. IoT health devices). Their end result is a bi-directional attestation mechanism, with the capability to also do only uni-directional attestation for situations where bi-directional attestation is not possible.

#### 3.1.1 System Design

From an operational perspective the attestation architecture runs inside a Trusted Execution Environment (as opposed to the TPM based attestation mechanisms which they state are not sufficient for their use case). They use the 'quote' abstraction in their architecture to send the current state of the TEE. In this work, we refer to a 'quote' as *Evidence*. The mechanism is TEE-agnostic so it does not require TEE features specific to any manufacturer. What they do suggest, is using a Trusted Measurer (TM) to sign the *Evidence* with a key that resides on the device. The Trusted Measurer should be verified in the authenticated boot process. Lastly, they mention that the use of a TPM could increase hardware level tamper resistance, but that would go at the cost of additional required hardware.

We will now discuss the bi-directional protocol. We also have a more formal notation in Protocol 1. Here, there are two *Trusted Applications* on both sides of a connection; we call these $A$ and $B$. One of these initiates the protocol by sending the identifier (ID) of both *Trusted Applications* ($ID_A$ and $ID_B$ respectively), a nonce $n_A$, a Diffie-Hellman exponentiation $G_A$, an attestation request for the other application $AR_B$ and a session cookie $S_{cookie}$, which is a hash of all the attributes that $A$ sent to $B$ except the attestation request. The parties can use this cookie to pick up an already established attestation session, without performing the full attestation again. This increases efficiency, because it means that the devices do not have to calculate new DH exponentiations. It does, however, mean that the loss of a session key can have a larger impact compared to systems which do not feature session resumption.

After $A$ has sent the first message, $B$ will respond with a message that consists of both *IDs*, a nonce which they generated ($n_B$) an-

**Protocol 1:** Bi-Directional Trust Protocol (BTP) by Shepherd et al. [27]

(1) $TA_A \rightarrow TA_B : ID_A \mid\mid ID_B \mid\mid n_A \mid\mid G_A \mid\mid AR_B \mid\mid S_{cookie}$
$S_{cookie} = H(G_A \mid\mid n_A \mid\mid ID_A \mid\mid ID_B)$
(2) $TA_B \rightarrow TA_A : ID_B \mid\mid ID_A \mid\mid n_B \mid\mid G_B \mid\mid [\sigma_{TA_B}(X_{TA_B}) \mid\mid \sigma_{TA_B}(V_{TA_B})]^{K_E}_{K_{MAC}} \mid\mid AR_A$
$X_{TA_B} = H(ID_A \mid\mid ID_B \mid\mid G_A \mid\mid G_B \mid\mid n_A \mid\mid n_B)$
$V_{TA_B} = Q_{TA_B} \mid\mid n_B \mid\mid n_A$
(3) $TA_A \rightarrow TA_B : [\sigma_{TA_A}(X_{TA_A}) \mid\mid \sigma_{TA_A}(V_{TA_A})]^{K_E}_{K_{MAC}} \mid\mid S_{cookie}$
$X_{TA_A} = H(ID_A \mid\mid ID_B \mid\mid G_A \mid\mid G_B \mid\mid n_A \mid\mid n_B)$
$V_{TA_A} = Q_{TA_A} \mid\mid n_B \mid\mid n_A$

other Diffie-Hellman exponentiation $G_B$. Then $B$ also sends a part which is encrypted with symmetric key $K_E$ and has a MAC with key $K_{MAC}$, denoted by: $[\sigma_{TA_B}(X_{TA_B})||\sigma_{TA_B}(V_{TA_B})]^{K_E}_{K_{MAC}}$. Where $X_{TA_B}$ is a hash of the IDs, nonces and DH exponentiations and $V_{TA_B}$ is *Evidence* from $B$'s TEE, concatenated with both nonces. $\sigma_x(m)$ denotes that message $m$ is signed with $x$'s public/private key pair.

In the final step of the protocol, $A$ responds with an encrypted message which contains: $[\sigma_{TA_A}(X_{TA_A})||\sigma_{TA_A}(V_{TA_A})]^{K_E}_{K_{MAC}}$ along with $S_{cookie}$, where $X_{TA_A}$ should be the same as $X_{TA_B}$ and $V_{TA_A}$ is now *Evidence* generated by the TEE of $A$ along with both nonces. Note that the protocol supports both bi-directional as uni-directional attestation, and in the latter case, the *Evidence* by $A$ is not required.

A secure connection between the trusted sensing device and the remote server has now been established. According to Shepherd, this connection provides stronger security guarantees than a regular TLS connection because it operates within a Trusted Execution Environment, which is entirely verified by a Trusted Measurer. The sensing device now possesses a secure environment (a TEE) that has been attested, enabling it to transmit sensing data securely. As a result, the remote server enjoys a higher level of trust compared to a regular connection that relies solely on a secret key, as the data comes from a source that has been attested.

#### 3.1.2 Limitations

In their article, Shepherd et al. provide a clear explanation of their protocol, which has the ability to function in both bi-directional and uni-directional modes, a promising feature. However, the authors acknowledge that their protocol yields a 4x overhead compared to conventional TLS with Diffie-Hellman and RSA. This could have an impact on tightly constrained devices. Furthermore, due to its multi-architecture compatibility, the protocol is unable to leverage advanced hardware features that offer stronger security assurances for uncompromised devices. Lastly, the protocol requires that both devices always have an active network connection to satisfy the protocol. This makes sense for the use case of trusted sensing devices, but this makes the protocol not suitable for certain other deployment scenarios.

### 3.2 Ménétrey et al.

Ménétrey et al. [38] implemented a secure WebAssembly runtime environment for TrustZone, called WaTZ (WebAssembly TrustZone). WebAssembly (Wasm) [39] is a binary instruction format designed for efficient execution of code on modern web browsers and other platforms. Since many programming languages can now compile to Wasm, WaTZ offers a secure runtime for numerous applications. As part of WaTZ, the authors developed an attestation mechanism. We will now focus on that attestation mechanism. The goal of this mechanism is to provide a way to attest to the code in the WaTZ environment while keeping the implementation of the connection to the remote server small. After the attestation of the WaTZ environment the *RP* has a stronger guarantee that the code running in the environment is secure.

#### 3.2.1 System Design

In WaTZ, a kernel module generates *Evidence* which includes:

- An anchor value that binds the parameters to a session

- The WaTZ version number
- The hash of the Wasm bytecode
- The public key of the attestation service
- The signature of the evidence

We rewrote their (SGX inspired) protocol in the same style as the previous section, which is shown in Protocol 2.

The protocol starts with the *TA* sending the public part of a session key-pair that the *TA* generated ($G_{TA}$) to the *RP*. Then, the *RP* responds by generating their own session key pair (with public part $G_{RP}$) Both the session key pairs are combined to create a *Key Derivation Key* which is subsequently derived in two shared secret keys: $K_m$ and $K_e$ for generating MACs and encrypting messages with a symmetric key, respectively. *RP* sends a message to *TA* with the $G_{RP}$, its public key $Pk_{RP}$ and a signature of the session keys. After receiving and verifying the message, *TA* gathers evidence locally and responds with a message that contains $G_{TA}$, the evidence (with a so-called *anchor* which is the hash of the public session keys) along with the *TA*'s public key, a signature of the evidence and a MAC of the entire message. *RP* receives and verifies this message again. Then it checks the evidence that it received and it makes the decision whether the device is in a known state. If that is the case, *RP* can send a new message with arbitrary confidential data that is encrypted with AES-GCM.

#### 3.2.2 Limitations

The authors have verified the remote attestation mechanism with Scyther to ensure its formal correctness and have provided a useful example of how a remote attestation mechanism could work in TrustZone. However, there are several limitations to this mechanism that should be considered.

First, the protocol is specific to WaTZ and as such can only support Wasm applications. This may not be easily adaptable for other applications, since these may have specific requirements and build environments. Second, the protocol sends the messages in plain-text over the network. More privacy oriented applications might require higher levels of confidentiality. The authors also mention this in their paper and state that these messages could also be encrypted with AES-GCM. Finally, the Wasm application does not perform local attestation and relies on Secure Boot to ensure the device is uncompromised. If the software does not have network access, it cannot determine whether it is running on an uncompromised device.

## 4 Research Questions

The Related Work and Background demonstrate that there is a lack of a well-documented, standardized attestation mechanism for applications running on TrustZone-enabled devices, which enables remote attestation with an open-source implementation or an easy to implement concept using open source building blocks.

To address this challenge, this research aims to design a new attestation protocol by comparing existing attestation mechanisms for the TrustZone. The proposed protocol will be designed to fit the RATS model [18] and will take into account the limited computational power of some TrustZone enabled devices to aid in the use case which is outlined in the Introduction. We have formulated a set of research questions that will guide our investigation and design process.

### 4.1 Comparing Mechanism Designs

**RQ1** How do existing works in research and open source projects compare to each other in the context of the feature sets that the specific mechanisms support?

### 4.2 Protocol Design

**RQ2** What would a design for a TrustZone-based attestation mechanism that supports both local and remote attestation look like?

**RQ2.1** How can we make use of existing open source projects to implement the aforementioned attestation mechanism protocol?

**Protocol 2:** Remote attestation protocol by Ménétrey et al. in WaTZ [38]

(1) $TA \rightarrow RP : G_{TA}$

(2) $RP \rightarrow TA : X_{RP} \mid\mid MAC_{K_m}(X_{RP})$
$X_{RP} = G_{RP} \mid\mid Pk_{RP} \mid\mid \sigma_{RP}(G_{RP} \mid\mid G_{TA})$

(3) $TA \rightarrow RP : Y_{TA} \mid\mid MAC_{K_m}(Y_{TA})$
$Y_{TA} = G_{TA} \mid\mid evidence \mid\mid \sigma_{TA}(evidence)$
$evidence = (H(G_{TA}||G_{RP})||Pk_{TA}||...)$

(4) $RP \rightarrow TA : iv \mid\mid AES - GCM_{K_e}(data)$

## 4.3 Evaluating the protocol

**RQ3** What is the performance of the proposed protocol in terms of speed, memory footprint and security?

**RQ3.1** How can we formally verify the correctness of the protocol?

**RQ3.2** How does the performance of a device running our protocol compare to the scenario where the device does not use it?

# 5 Comparing Protocols

To gain a good grasp of the current state of attestation mechanisms in Arm TrustZone for answering RQ1, we conducted a literature review on existing attestation mechanisms available for Arm TrustZone. We gathered references from the online database Google Scholar [40] using the search term “"TrustZone" "Attestation"”. We filtered these results to only include works that implemented their own protocols. From this list, we selected the first 15 papers. In this section, we discuss these papers in less detail compared to Section 3, where the goal was to provide the reader with a more in-depth understanding of those two mechanisms. Here, the works are compared based on the feature sets described in Sections 2.4.1 and 2.4.2. The results are presented in the following sections.

## 5.1 Functional Features

Table 1 shows the results of our comparison as answer to RQ1 with regards to the Functional Features from Section 2.4.1.

First, we can observe that most of the mechanisms discussed in the research support Remote Attestation (except for SofTEE). On the other hand, only half of the examined mechanisms from the papers support Local Attestation. However, all mechanisms that do not support local attestation do support remote attestation. Among the mechanisms supporting remote attestation, only three provide mutual attestation, while six out of fourteen support Secure Channels with the remote server. There is only one mechanism that supports session resumption. Additionally, there are only five TEE-agnostic mechanisms; the others are TrustZone-only. Furthermore, six of the papers also offer an Open-Source implementation for their mechanism or enclave implementation. Lastly, measuring attester anonymity is challenging, resulting in three unknowns in this regard. Five of the papers lack this anonymity, potentially allowing user data to be sent during attestation. In contrast, the remaining papers can usually uniquely identify a device (this seems to be inherent to many attestation mechanisms) without obtaining user data.

## 5.2 Security Features

The security features are challenging to combine into one table since the protocols differ significantly in their core. Consequently, the *KE* and *RT* columns contain text instead of symbols. Moreover, two papers (ERAMO [44] and SANCTUARY [20]) leave the choice of the key algorithm to the system implementer. Hence, certain fields in the table are marked with “✍” to indicate that the answer for that particular field depends on the specific implementation.

The tables uses some abbreviations for brevity which we explain more in-depth here:

**KDF (Key Derivation Function)** A key derivation function is a cryptographic algorithm

Table 1: Functional Features Table

| **Paper Name** | *LA* | *RA* | *MA* | *SC* | *SR* | *TA* | *OS* | *AA* | *Ref* |
|---|---|---|---|---|---|---|---|---|---|
| AdAttester | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗✓ | [41] |
| C-FLAT | ✓ | ✓ | ✗ | ✗ | ✗ | ? | ✓ | ✗ | [42] |
| End-to-End Security for Distributed... | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ? | [43] |
| ERAMO | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | [44] |
| Establishing Mutually Trusted Channels | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ? | [27] |
| Komodo | ✓ | ✓ | ✗ | (✓) | ✗ | ✗ | ✓ | ? | [45] |
| Practical Runtime Attestation for ... | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | [46] |
| Remote Attestation for Embedded ... | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | [47] |
| SANCTUARY | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | [20] |
| SecTEE | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ? | ✓ | [48] |
| SofTEE | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | [49] |
| SWATT | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | [50] |
| TrustZone based Attestation in ... | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | [51] |
| VRASED | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | [52] |
| WaTZ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | [38] |
| This work (PATAT) | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | |

✓ – Supported. (✓) – Supported by design, but no implementation. ✗ – Unsupported.
? – Unclear whether supported.

Table 2: Security Features Table

| **Paper Name** | MC | *MV* | *KE* | *FS* | *NR* | *RV* | *HV* | *RT* | *FR* | *Ref* |
|---|---|---|---|---|---|---|---|---|---|---|
| AdAttester | ✓ | ✓ | Device Keys | ✗ | ✓ | ✓ | ✓ | Secure Boot | ✓ | [41] |
| C-FLAT | ✓ | ✓ | ? | ✗ | ✓ | ✓ | ✗ | TrustZone | ✓ | [42] |
| End-to-End Sec... | ✓ | ✓ | KDF which? | ✓ | ✓ | ✗ | ✓* | HUK | ✓ | [43] |
| ERAMO | ✓ | ✍ | ✍ | ✍ | ✍ | ✓ | ✓* | TrustZone | ✓ | [44] |
| Establishing M... | ✓ | ✓ | DH | ✓ | ✓ | ✗ | ✍ | TM in TrustZone | ✓ | [27] |
| Komodo | ✓ | ✓ | MAC with secret | ✓ | ✓ | ✗ | ✓* | Hardware chain | ✓ | [45] |
| Practical Runt... | ✓ | ✓ | KDF for asym | ✗ | ✓ | ✗ | ✗ | TrustZone & HUK | ✓ | [46] |
| Remote Attesta... | ✓ | ✓ | Key from TPM | ✗ | ✓ | ✗ | ✗ | TZ, kernel & SB | ✓ | [47] |
| SANCTUARY | ✓ | ✍ | ✍ | ✍ | ✍ | ✗ | ✓ | TrustZone & HUK | ✓ | [20] |
| SecTEE | ✓ | ✓ | DH | ✓ | ✓ | ✗ | ✗ | TrustZone & HUK | ✓ | [48] |
| SofTEE | ✓ | ✓ | Key in security monitor | ✓ | ✓ | ✗ | ✗ | TPM | ✓ | [49] |
| SWATT | ✓ | ✓ | Random generated MAC | ✓ | ✗ | ✓ | ✗ | Random MAC | ✓ | [50] |
| TrustZone base... | ✓ | ✓ | Same root key | ✓ | ✓ | ✓ | ✗ | TrustZone & HUK | ✓ | [51] |
| VRASED | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | HUK and form. ver. | ✓ | [52] |
| WaTZ | ✍ | ✓ | ECDHE | ✓ | ✓ | ✗ | ✓* | HUK & SB | ✓ | [38] |
| This work (PATAT) | ✓ | ✓ | DH | ✓ | ✓ | ✓ | ✍ | TrustZone & HUK | ✓ | |

✓ – Supported. ✓* Supported with a caveat. (✓) – Supported by design, but no implementation. ✗ – Unsupported.
? – Unclear whether supported. ✍ – Depends on chosen algorithms/implementation

that derives a secret key from another secret value, such as a master secret key. Such algorithms typically make use of a pseudorandom function, with the master secret acting as input to calculate the eventual key. One of its primary uses is to prevent the exposure of the actual secret key in the event that a derived key is leaked.

**DH (Diffie Hellman)** Diffie-Hellman functions are explained in Section 2.5.2. They are used for secure key exchange between parties.

**ECDHE (Elliptic Curve Diffie-Hellman Ephemeral)** ECDHE is a modern variant of the Diffie-Hellman key exchange, using an elliptic curve (EC) public-private key pair. Unlike e.g. RSA, which relies on factorization of large numbers, EC cryptography is based on scalar multiplication on points on an elliptic curve. At its core it works because it is considered a "hard" problem to find the private key from the public key and the public parameters of the curve. This approach allows for the use of smaller key sizes compared to alternative public-key cryptography methods [53].

**TZ (TrustZone)** Some attestation mechanisms leverage the unique features provided by TrustZone to serve as a root of trust. This can be achieved, for example, by employing a Trusted Measurer within a TrustZone application. Given the properties of TrustZone, the Trusted Measurer is expected to remain secure.

**HUK (Hardware Unique Key)** Attestation mechanisms may require devices to possess a hardware unique key serving as a root of trust, enabling the generation of additional keys using a KDF. Typically, these devices are prepared in the factory by fusing the key onto the board to prevent tampering. Furthermore, access to these keys is generally restricted to trusted components of the software residing on these devices.

**SB (Secure Boot)** Secure Boot [22] is a computer security feature that ensures only trusted software is loaded during the boot process. It works by verifying the digital signature of each piece of code, from the firmware to the operating system kernel, before allowing it to run. This prevents unauthorized or malicious software from executing during startup, thereby enhancing the overall security of the system.

In the comparison, we first note that all mechanisms perform message verification and depending on the implementation, WaTZ is the only mechanism to omit message confidentiality. However, one of them lacks non-repudiation: in SWATT, the challenge key is transmitted in plain text over the network without an additional key, allowing others to intercept it and generate their own responses. Additionally, four mechanisms lack forward secrecy in the transmitted messages (SofTEE does not support remote attestation, hence it is marked with an (✗)). Furthermore, six mechanisms support runtime verification. Lastly, we observe that only three mechanisms support hardware verification, while others rely on secure boot for this security feature.

## 6 Protocol Design

Answering RQ2 requires us to come up with a protocol which can be implemented as an open source library for use on TrustZone devices (most notably Cortex-A devices). We take the learnings from Section 5 to create our own protocol design, while making sure that this remains easy to implement with open source building blocks, which answers RQ2.1. We call this the Protocol for ATestation in Arm Trustzone (PATAT).

### 6.1 Goals & Assumptions

With the use-case of Scalys described in the Introduction in mind, the goals of our newly created attestation mechanism are as follows:

- Implement the concepts from RATS [18] as described in Section 2.3.3.
- Allow third parties to perform attestation for their own applications but not necessarily for the device manufacturer features.
- Allow device manufacturers to perform attestation on the device features such as its

firmware but not necessarily on third-party applications.

- Ensure that the device manufacturer and app developer are not required to share sensitive information about the attestation.
- Provide an option to retain a secure connection between the *TA* and the *RP* after successful attestation.
- Ensure that the mechanism is relatively easy to implement in any programming language.

These goals aim to make the protocol attractive to software manufacturers who want to run part of their application on a TrustZone-enabled device, with the added benefit of allowing attestation. They ensure that software vendors do not have to be responsible for the attestation of firmware and hardware, which may be outside their expertise. Additionally, these goals are designed to appeal to hardware manufacturers who create their own hardware equipped with TrustZone-enabled SoCs. The option to retain a secure connection after successful attestation enhances efficiency by allowing an established secure connection to be reused instead of being dropped. The final goal, making the mechanism easy to implement in any programming language, aims to reduce development costs, thereby increasing the adoption rate among software vendors.

We assume that, the *RP* has a secure connection to the *Verifiers*. Man-in-the-middle attacks and other interference should not be possible here. An option could be to set up an IPSEC connection with pre-shared symmetric keys over a secure channel, but the exact implementation is out of scope for this work. These parties can also opt to use the handshake from this work as a means to setup a secure connection. We do not define this in detail here to allow for flexibility in setting this up later and keeping the proof of the protocol simple.

## 6.2 Proposed Protocol High Level

This section describes this work's protocol on a high level. It starts by describing the way in which *Evidence* is packaged in Section 6.2.1 followed by an explanation of the attestation flow in Section 6.2.2.

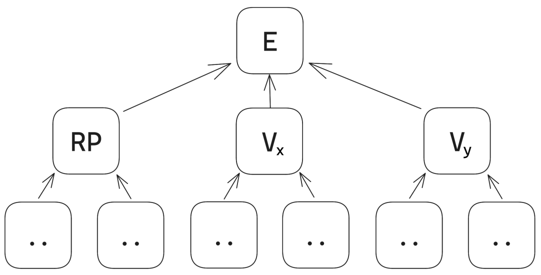


Figure 6: Merkle tree with evidence for the protocol

### 6.2.1 Evidence Format

Figure 6 shows our implementation of a Merkle Tree to use as *Evidence* in our proposed protocol. In this specific implementation of the Merkle Tree, we have a subtree for each role in the attestation process. Figure 6, e.g. has a subtree with information that the *RP* knows, and 2 *Verifier* subtrees: $V_x$ and $V_y$. Together, they form the root of the tree: $E$. The root of the tree, $E$, will be sent by the *Attester* application/device to the *RP*. Structuring the data which is sent to the *RP* in this way makes sure that *Verifiers* and the *RP* do not learn private information about each other, since that information is stored in the Merkle Tree and thus only available in a Merkle Proof.

### 6.2.2 Attestation Flow

Figure 7 shows a high level overview of the attestation flow in the PATAT Protocol. We will now provide a description to the numbers in the image.

0. The *TA* and *RP* perform a handshake to set up a secure connection for performing the actual attestation. During this handshake, basic information about the *TA* may also be communicated, such as software version numbers used during verification later.

1. The *TA* assembles *Claims* into our proposed *Evidence* Merkle Tree denoted as the box with the blue key in the Figure. This tree is split into a subtree with *Claims* related to the *RP* (e.g., device manufacturer) and another subtree with *Claims* related to the *Verifier* (third-party application developer). It sends this as `E` to the *RP*.

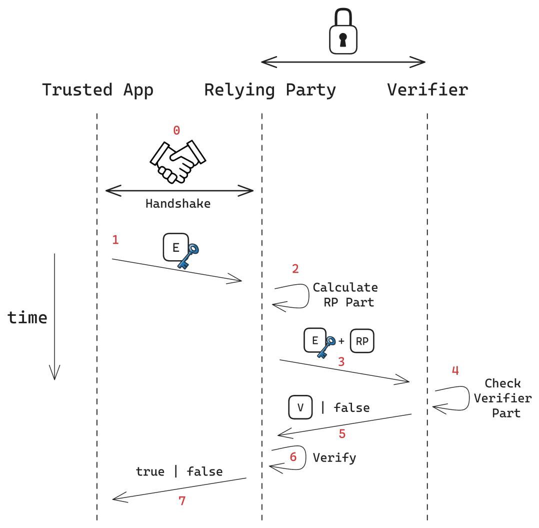


Figure 7: The Attestation flow in the proposed protocol

**Algorithm 3:** The Hash Key Derivation Function

```
Function HKDF($old\_chaining\_key, input\_data$):
    $temp\_key = \texttt{HMAC}(old\_chaining\_key, input\_data)$
    $new\_chaining\_key = \texttt{HMAC}(temp\_key, \text{0x01})$
    $new\_symmetric\_key = \texttt{HMAC}(temp\_key, output\_1 \mid\mid \text{0x02})$
    return $new\_chaining\_key, new\_symmetric\_key$
```

2. The *RP* calculates the hash of its part of the Merkle Tree with the information received about the device during the handshake.

3. Over the pre-existing connection between the *RP* and the *Verifier*, the *Evidence* Merkle Tree and the *RP*'s side of the Merkle Proof are sent to the *Verifier*. Additionally, the *Verifier* is provided with the same basic (non-private) information about the *TA*.

4. The *Verifier* uses the *RP*'s part of the Merkle Tree to perform a Merkle Proof on the *Evidence* Merkle Tree, checking the *Claims* sent along the Merkle Tree.

5. If the *Verifier* has successfully checked the *Evidence*, it responds to the *RP* with the *Verifier*'s proof of the Merkle Tree. Otherwise, it returns that the validation failed.

6. The *RP* checks the *Evidence* with the information from the *Verifier* to validate the total attestation.

7. It responds to the *TA* whether the attestation was successful or it breaks the secure connection if the attestation failed.

## 6.3 Proposed Protocol Handshake

Before sending the Merkle Tree, described in Section 2.5.1 to the *RP* a secure connection should be set up between the *TA* and the *RP*. The comparison study outlined in Section 5 reveals that numerous existing protocols employ either a symmetric encryption algorithm or a variant of Diffie-Hellman key exchange. The symmetric key approach poses difficulties with regards to scalibility. Moreover, it is desirable for the protocol to be capable of generating a new key for each attestation-performing application. Finally, a secure channel should be established after the handshake to enable fast and secure message exchange. A promising framework for designing new handshake protocols is the Noise Protocol Framework developed by Trevor Perrin [54].

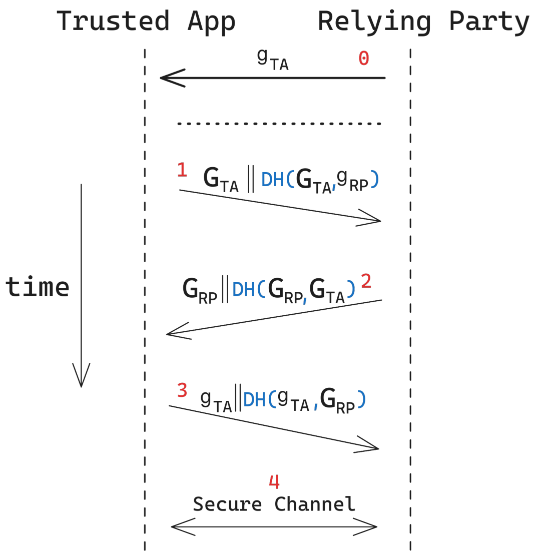
The Noise Protocol Framework is a framework for constructing secure cryptographic protocols that support authentication, forward secrecy, and identity hiding. It is not a handshake protocol by itself, but a framework to develop handshake protocols with certain security guarantees. Additionally, protocols created using the Noise Framework are simple and require only a hash function, a symmetric cipher function (in AEAD mode), and a Diffie-Hellman function. The simplicity of this framework makes it a logical choice for adopting it in our attestation mechanism, particularly considering the constraints that are associated with developing embedded systems (where ARM devices are often used). The framework has also been investigated in research [55] and been used to create secure handshake protocols for services such as WhatsApp [56], Wireguard [57] and the Lightning Network [58]. Lastly, due to its relatively simple nature, noise protocols can be proven in a formal manner, which is useful in answering RQ3.1. We will now continue by explaining the handshake mechanism.

In this handshake protocol, we will use the `XK` variant of the Noise Protocol Framework. This variant allows the *TA* to send their static public key to the *RP* during the handshake (`X` means transmitted), while already having knowledge of the *RP*'s static public key (`K` means known). The handshake is shown in Protocol 4. It is also displayed in simplified form in Figure 8 (note that the optional encrypted payload for each message has been ommited for clarity). The next paragraph aims to explain in more detail what has been described in the protocol notation.

In the following section $G_x$ denotes a public ephemeral key from $x$ and $g_x$ denotes static public key from $x$.

During the setup of the *TA*, it should have received the *RP*'s static public key (0). Manufacturers of devices should be free to decide in what manner this key ends up on the device with the *TA*. However, the key must remain protected from tampering.

Then, the handshake starts with the *TA* generating an ephemeral Diffie-Hellman key and sending it to the *RP* in (1). Along with that, the *TA* can also encrypt some additional payload data with a temporary key that has been generated from Algorithm 3 by using the previous chaining key *ck* and the Diffie-Hellman calculation between the private part of the *TA*'s Diffie-Hellman key and the (already known)



Trusted App
Relying Party
$g_{TA}$
0
1 $G_{TA} \| DH(G_{TA}, g_{RP})$
time
$G_{RP} \| DH(G_{RP}, G_{TA})$ 2
3 $g_{TA} \| DH(g_{TA}, G_{RP})$
4
Secure Channel

Figure 8: Simplified representation of the protocol handshake ($G_x$ denotes a public ephemeral key from $x$ and $g_x$ denotes static public key from $x$).

public part of the *RP*'s static *DH* key. However, the payload in this first message should not be treated as trusted input by the server, because the plaintext is encrypted with an ephemeral key. Therefore, the server cannot yet know if it is interacting with a genuine *TA*. Since the encryption is a form of AEAD, we also encrypt with associated data and a nonce that increments after encryption and resets after every *HKDF* function. Each time encryption is mentioned from now on, we refer to AEAD encryption with $h$ as associated data and this nonce. In this protocol, that is $h$; a hash chain of data which both the *TA* and *RP* learn during the handshake. We denote AEAD encryption with these values as $[payload]^h_{key}$. In the case of this first message, the previous $h$ value is hashed together with the public part of the Diffie-Hellman key to act as associated data. After the encryption of the payload, $h$ is updated by hashing its previous value with the ciphertext. Note that in the case that there is no payload, we will still be performing AEAD, just with 0 bytes as the input. Once the *RP* receives the message from the *TA*, they will first calculate the value of $h$ with the *TA*'s public key, then decrypt the payload and verify the AD and lastly calculate the current value of $h$ by hashing with the ciphertext.

For step (2), the *RP* also generates an

**Protocol 4:** Protocol Handshake

```
(0) $RP \rightarrow TA : S_{RP}$
(1) $TA \rightarrow RP : G_{TA} \mid\mid ciphertext$
    $h = \texttt{HASH(}h \mid\mid G_{TA}\texttt{)}$
    $ck, k1 = \texttt{HKDF(}ck,\ \texttt{DH(}g_{TA}, S_{RP}\texttt{))}$
    $ciphertext = [payload]^{h}_{k1}$
    $h = \texttt{HASH(}h \mid\mid ciphertext\texttt{)}$
(2) $RP \rightarrow TA : G_{RP} \mid\mid ciphertext$
    $h = \texttt{HASH(}h \mid\mid G_{RP}\texttt{)}$
    $ck, k2 = \texttt{HKDF(}ck,\ \texttt{DH(}g_{RP}, G_{TA}\texttt{))}$
    $ciphertext = [payload]^{h}_{k2}$
    $h = \texttt{HASH(}h \mid\mid ciphertext\texttt{)}$
(3) $TA \rightarrow RP : encrypted\_key \mid\mid ciphertext$
    $encrypted\_key = [S_{TA}]^{h}_{k2}$
    $h = \texttt{HASH(}h \mid\mid encrypted\_key\texttt{)}$
    $ck, k3 = \texttt{HKDF(}ck,\ \texttt{DH(}s_{TA}, G_{RP}\texttt{))}$
    $ciphertext = [payload]^{h}_{k3}$
    $h = \texttt{HASH(}h \mid\mid ciphertext\texttt{)}$
(4) $TA \rightarrow RP : [payload]_{c1}, RP \rightarrow TA : [payload]_{c2}$
    $c1, c2 = \texttt{HKDF(}ck,\ \_\texttt{)}$
```

ephemeral DH key. This key will be the first part of the message which it will send back to the *TA*. The *RP* further chains the *h* value with the generated ephemeral public key and calculates a new key to encrypt the payload for this message by 'mixing in' their ephemeral key with the *TA*'s ephemeral public key. Then a new value for *h* is calculated by adding the new ciphertext to the hash chain. The *TA* receives the ephemeral key along with the ciphertext and mirrors these steps to decrypt the payload.

Step (3) is a bit different, since the *TA* will send their static public key in encrypted form to the *RP*. To encrypt the static public key, they use the same key as used in the payload encryption from step (2) and the updated *h*. Then, the encrypted key is hash-chained to *h* and a new DH calculation occurs between the *TA*'s static key and the *RP*'s ephemeral key. Again, an optional payload is encrypted with the new key and afterwards *h* gets a new value.

Step (4) finishes the handshake by creating 2 "CipherStates": $c1$ for traffic from *TA* to *RP* and $c2$ for messages from *RP* to *TA*. From this moment on, all messages are encrypted symmetrically with these 2 keys and associated data set to zero length and an incrementing nonce per cipher state. So secret data can now be exchanged safely between *TA* and *RP*. This includes the Merkle Tree as well as any other secret data that may be exchanged between these parties.

## 7 Examining the Protocol

In this section, we delve into the security and performance aspects of the proposed protocol, as described in Section 6. We start by demonstrating the correctness of the handshake in Section 7.1, followed by a Proof-of-Concept implementation of the protocol in Section 7.2 and an analysis of the protocol's performance in Section 7.3.

### 7.1 Protocol Proof

As a partial answer to RQ3 and, more specifically, RQ3.1, we verified the formal correctness of the PATAT protocol. To achieve this, we modeled it in the Tamarin Prover of which the basics already been explained in Section 2.6. At a high level, we created two Tamarin files with the proofs: one for verifying the *Integrity* of the protocol and one for verifying the *Confidentiality* of the protocol. With these proofs, we aim to provide a formal way of demonstrating that messages in the PATAT Protocol cannot be tampered with or read without compromising secret keys. The complete Tamarin Model for PATAT, along with the output of the Tamarin Prover, can be found on GitLab [59].

### 7.1.1 Modeling Functions

We began the model by defining the requirements for built-in hash and Diffie-Hellman functionality. Additionally, we created custom functions for AEAD encryption, decryption, and verification. We also defined a function with 2 parameters for HMAC. For the AEAD functions, we included specific equations in Tamarin to represent how these functions work. The HMAC function doesn't need an equation because it is modeled as a one-way function. However, since Tamarin only recognizes state, not functions, equations are required for modeling the encryption and AEAD verification processes. These equations are as follows:

```
enc = aead(k, n, a, p)
decrypt(enc, k, n, a) = p
verify(enc, k, n, a) = true
```

### 7.1.2 Tamarin Rules

With the functions from the previous section, we can create the `rules`. To provide the model with an opportunity to "win", we begin by creating a rule that takes the persistent fact `KeyPair`, which contains a public and private key. This implies that a `KeyPair` should already exist in the multiset. The arrow indicates that the previous state is transformed into the state after the arrow, signifying that the `Out` fact containing the private key is now part of the multiset, indicating that the private key has been revealed. The arrow also carries the name of the transformation `RevealSecretKey` as the action fact, which we can use later in the lemmas.

```
rule reveal_private_key:
    [!KeyPair(pubkey, ~privkey)]
--[RevealSecretKey(pubkey)]->
    [Out(~privkey)]
```

We created a similar rule for revealing the ephemeral keys to the multiset. Then, we modeled the setup stage of the protocol by creating an `initialize` rule for both the *RP* and the *TA*. The *TA* variant takes the a freshly generated ID, a generated static `KeyPair`, the server's public key, and prologue data that has been agreed on beforehand. This becomes the following Tamarin block:

```
Fr(~ta_id),
!KeyPair(ta_pub_s, ~ta_priv_s),
In(server_pub_s),
In(prologue)
```

Then, the rule outputs the following facts:

```
!StaticKey(
    ~ta_id, ta_pub_s, ~ta_priv_s
),
TAInitializedState(
    ~ta_id, h3, ck1, server_pub_s
)
```

The variables in these output facts have the following assignments:

```
ta_pub_s = 'g'^~ta_priv_s
h1 = h('PROTOCOL_NAME')
ck1 = h1
h2 = h(<h1, prologue>)
h3 = h(<h2, server_pub_s>)
```

With these assignments, we have modeled the steps from the handshake setup from Section 6.3. The *RP* `initialize` rule is modeled very similarly to the rule above.

With the setup modeled, we can move on to the actual handshake. We split this part up into 4 rules, one for each message from each actor in the handshake. We show the rule for the *TA*'s first message as an example in Listing 1. The others can be found in the complete model on GitLab [59].

In the rule, we model the input as a fresh ephemeral key and a previously generated static keypair associated with the *TA*'s ID. The rule also takes the `TAInitializedState` fact from the `initialize` rule. Additionally, we model the payload that the *TA* will send as a fresh random value. We model the payload this way in the *Confidentiality* version because the adversary should not be aware of its contents. In the *Integrity* version, we always take the payloads from the network, as this modeling accounts for the possibility that an attacker may have knowledge of or influence over the payload, thereby creating a stronger adversary model.

The output of this rule includes a persistent `EphemeralKey` fact, representing that the *TA* has generated a new ephemeral key pair. Along with it is a `TAAfterFirstMessageState` fact

Listing 1: Tamarin rule for the *TA*'s first message

```
rule ta_first_message:
    let
        ta_public_s = 'g'^~ta_private_s
        ta_public_e = 'g'^~ta_private_e
        h4 = h(<h3, ta_public_e>)
        dh_es = (server_public_s^~ta_private_e)
        temp_k = hmac(ck1, dh_es)
        ck2 = hmac(temp_k, '0x01')
        k1 = hmac(temp_k, <ck2, '0x02'>)
        n1 = '0'
        ciphertext1 = aead(k1, n1, h4, ~payload1)
        h5 = h(<h4, ciphertext1>)
        message = <ta_public_e, ciphertext1>
    in
        [
            Fr(~ta_private_e),
            !StaticKey(~ta_id, ta_public_s, ~ta_private_s),
            TAInitializedState(~ta_id, h3, ck1, server_public_s),
            Fr(~payload1)
        ]
    --[
        SendMessage(~ta_id, 'payload1', ~payload1),
        SendCiphertext(~ta_id, 'payload1', ciphertext1),
        SetOwnEphemeralKey(~ta_id, ta_public_e),
        RunningTA(~ta_id, ~ta_private_s),
        TAKeyUsed(~ta_id, 'm1', k1, n1, h4)
    ]->
        [
            !EphemeralKey(~ta_id, ta_public_e, ~ta_private_e),
            TAAfterFirstMessageState(~ta_id, h5, ck2, k1),
            Out(message)
        ]
```

with the current state of the variables `h`, `ck`, and `k` in the handshake. Finally, we model the message in an `Out` fact, indicating that the adversary can also interact with it. The action facts in the middle are used later in the lemmas for proving the protocol. In the `let` block, we model the transformation of the variables during the handshake and the encryption of the `message` with AEAD encryption. The other three rules, representing the different messages sent during the handshake, are modeled similarly.

#### 7.1.3 Tamarin Lemmas

The rules from the section model the PATAT protocol and Tamarin action facts were assigned in the rules. These action facts are used to model the lemmas which are used to provide the actual proofs for the PATAT Protocol.

In Listing 2, we have included one of the lemmas used in the Tamarin proof for the confidentiality of the handshake messages. This lemma aims to prove that even if an adversary gains access to the *RP*'s ephemeral secret key, the confidentiality of the messages is maintained. We start by defining the variables used in the lemma. Then, we denote in the `ReceiveMessage` action fact that the *TA* has received the second message. The section after that describes the other setup steps that have occurred during the handshake. We also specify that only the *RP*'s ephemeral secret key was compromised by the adversary. Using the `==>` arrow, we indicate that these action facts together imply that the adversary is not aware of the payload. When we run the Tamarin Prover, it will either verify or falsify this lemma, allowing us to determine whether this confidentiality assumption holds.

#### 7.1.4 Proof Results

Running the verification on all the lemmas in Tamarin outputs for each whether they are falsified or verified. Table 3 shows the results of the integrity and confidentiality lemmas. In the table we see that the security of the handshake messages remains, even when some of the intermediate keys have been stolen by an attacker. The table does not include cases where both *TA* and *RP* lose their keys because it is trivial to see that such cases cannot be secure. The proof shows that payload secrecy for all handshake messages is guaranteed when all the keys remain secure. The integrity for the first message cannot be guaranteed, since that message only includes an ephemeral key from the TA. The following handshake messages all have a static key mixed in, which guarantees the integrity of the handshake messages. An interesting finding is that the third handshake message's secrecy can be guaranteed when two keys have been stolen by the attacker, except for the case in which both ephemeral keys are compromised.

### 7.2 Protocol Implementation

We implemented our protocol in Rust in the OP-Tee [12] environment. For that, we made use of the OP-TEE Rust SDK available from Apache [60]. With this, we created a client application in OP-TEE which spawns a *TA* that sets up a connection with a remote *RP*. The cryptographic operations in the implementation make use of the cryptographic primitives exposed in the Teaclave SDK. We emulated a system with TrustZone using QEMU [61] on a regular `x86_64` machine to test the protocol in a controlled environment.

For the implementation, we use a 32-byte hash size and set the Diffie-Hellman output length to 32 bytes with a 2048-bit key. For symmetric encryption, we chose ChaCha20Poly1305 due to its faster software implementation compared to AES-GCM. This algorithm uses a key size of 32 bytes (256 bits). Consequently, the Noise protocol handshake pattern string is `Noise_XK_25519_ChaChaPoly_SHA256`. This string is used at the beginning of the protocol to ensure compatibility with any implementation of the Noise spec. This approach allows for easier implementation if the protocol needs to be developed in another programming language that also has a ready-made Noise implementation. Due to the limited size of the computing base in the TA, not all modern Rust libraries can be used, making it more challenging to quickly develop a Proof-of-Concept application.

To provide additional verification for our implementation on the *TA* side, we implemented the *RP* side of the protocol in Rust for an `x86_64` machine. For this, we used the snow package in Rust [62] and the built-in hash meth-

Listing 2: An example of a lemma.

```
lemma payload_payload2_confidentiality_lost_single_rp_eph_still_safe:
    "
    // For all the values that these variables can hold...
    All ta_id ta_pubkey payload2 server_pubkey
        server_id ta_e_pubkey server_e_pubkey
        #a #b #c #d #e #f #g #i #j.

    // If the RP receives payload2...
    ReceiveMessage(ta_id, 'payload2', payload2) @a

    // After a correct setup...
    & SetOwnStaticKey(ta_id, ta_pubkey) @b
    & AcceptStaticPubKey(server_id, ta_pubkey) @c
    & SetOwnStaticKey(server_id, server_pubkey) @d
    & AcceptStaticPubKey(ta_id, server_pubkey) @e
    & SetOwnEphemeralKey(ta_id, ta_e_pubkey) @f
    & AcceptEphemeralKey(server_id, ta_e_pubkey) @g
    & SetOwnEphemeralKey(server_id, server_e_pubkey) @i
    & AcceptEphemeralKey(ta_id, server_e_pubkey) @j

    // And TA Ephemeral key is safe
    & (not Ex #n. RevealEphemeralKey(ta_e_pubkey) @n)

    // But RP Ephemeral secret key is stolen
    & (Ex #n. RevealEphemeralKey(server_e_pubkey) @n)

    // And TA Long term key is safe
    & (not Ex #n. RevealSecretKey(ta_pubkey) @n)

    // And RP Long term key is safe
    & (not Ex #n. RevealSecretKey(server_pubkey) @n)
        ==>
    // Then an attacker should not know the contents of
    // the second handshake message
    (not Ex #x. K(payload2) @x)
    "
```

Table 3: Formal Proof Results

| Lost keys | Final | 1C | 1I | 2C | 2I | 3C | 3I |
|---|---|---|---|---|---|---|---|
| None | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| $G_{TA}$ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| $G_{RP}$ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| $g_{TA}$ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| $g_{RP}$ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| $G_{TA}\&G_{RP}$ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| $G_{TA}\&g_{RP}$ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| $g_{TA}\&G_{RP}$ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| $g_{TA}\&g_{RP}$ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |

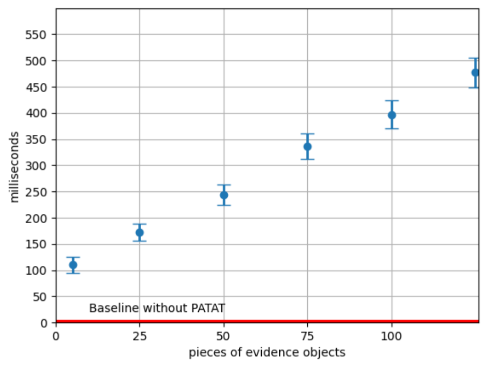


Figure 9: Measurements of the protocol runs

ods for creating the Merkle Tree data format. Using a tried and tested implementation of the Noise protocol and Merkle Tree on this side of the connection grants us additional confidence in the *TA*'s implementation when the attestations are verified. The application in the QEMU environment can communicate with the `x86_64` machine, acting as the *TA* and *RP*, respectively. Both sources are available on GitHub [63] [64].

### 7.3 Protocol Measurement

We evaluated our protocol by measuring the time it takes to run *Trusted Applications* with this work as remote attestation protocol. The measurements were done in a QEMU environment, simulating an Arm Cortex V8 chip. The QEMU environment ran on Fedora 39, powered by an AMD Ryzen™ 9 5900X and 32GB DDR4 RAM. The measurements were performed six times, varying the amount of *Claims* used. Starting with 5 pieces of evidence strings, then 25 incrementing in steps of 25 until 125. The evidence strings were hard-coded to ensure that the protocol's performance was assessed independently of the measuring process.

To answer RQ3.2 each configuration was tested 1000 times. Figure 9 shows the mean and standard deviation of the time elapsed (in ms) for each amount of *Claims* in the *Evidence*. The red line shows mean time to run the application for which no attestation was performed to run the application in the TEE. This acts as a baseline to compare the execution times of the other runs.

We also measured the payload sizes for sending the *Evidence* during our measurements. These lie between 183 bytes for the smallest number of *Evidence* objects and 311 for the maximum number that we used in the runs.

## 8 Discussion

This section discusses the protocol's design from Section 6, its verification from Section 7.1 and the protocol PoC implementation and examination from Section 7.2.

### 8.1 Protocol Considerations

The protocol provides a novel way of performing remote attestation. Its main contribution is the elegant combination of Merkle Trees as the *Evidence* format and the use of the simple-to-implement Noise protocol for setting up a secure connection. Given that these concepts are already known and implementations already exist, they can be combined relatively easily to create the PATAT Protocol.

Although this work primarily focuses on remote attestation, the protocol could technically also be performed locally on the device. Encryption and TrustZone concepts ensure that no malicious software on the device can eavesdrop on the connection between the trusted part, acting as the *RP*, and the *TA*. A hybrid between local and remote attestation is also possible, and future work could explore the possibility of creating a Merkle Tree chain that holds results of previous remote attestations and uses them for local attestation.

Furthermore, since the handshake is performed using Diffie-Hellman, some applications that must be quantum-proof might hesitate to apply the PATAT Protocol. For such cases, it is relatively easy to extend the protocol with a symmetric key. However, this would require an extension to the proof from Section 7.1 and lead to slightly more overhead in key distribution.

### 8.2 Verification Results

The results from the protocol verification, outlined in Section 7.1, indicate that the protocol maintains security under normal conditions. Additionally, the analysis confirms that the security assumptions regarding authenticity and secrecy remain intact even if certain protocol keys are compromised. The only exception is when both ephemeral keys are lost to an attacker. It should be noted, however, that the ephemeral keys are designed to be neither stored nor reused. The Noise protocol's security considerations also explicitly state that reuse of ephemeral keys is prohibited. Hence, this scenario is not likely to occurr in practice.

Next to that case, the proof shows that the authenticity of the first message cannot be guaranteed. Therefore, any additional data that is sent along that message should not be trusted until the long term key is mixed. This outcome was anticipated, given that the initial message contains only an ephemeral key integrated into its encryption, which cannot be authenticated by the *RP*. Thus, there is a potential vulnerability that could be exploited by an active attacker if data from message 1 is handled as trusted data.

Furthermore, if we consider the scenario where the ephemeral key of the *TA* is compromised, the attacker gains the ability to intercept the second message. Given that the second message originates from the *RP* and is directed to the *TA*, the authenticity is perceived from the *TA*'s perspective. Consequently, assuming that the *TA* initiated the first message and its ephemeral key was indeed compromised, the *TA* cannot be sure that the second message came from the *RP*. Even if they were the ones that sent the message, the attacker with the ephemeral key is able to read its contents. Again, this was expected, as the ephemeral key from the *TA* is the only key they mixed in, and its compromise inevitably compromises the security of the second message as well.

Lastly, we see that in all the other cases, confidentiality and integrity of the messages hold. Which gives us a set of practical security considerations for this protocol:

1. The first message must not include any confidential payload.

2. Trust in any payload data from the first message is only justified after verification of message 3.

3. Message 2 should not contain secret information, since the receiver has not been verified.

4. Message 3 can be used to send some secret information before the actual secret channel has been setup. This message has similar security guarantees as the secure channel.

5. Ephemeral keys must not be reused.

## 8.3 Implementation

The implementation discussed in Section 7.2 provides a way to perform attestation on itself, but it is unfinished. Currently it can perform a handshake to setup a connection from the *TA* to a server and combine different *Claims* into Merkle Tree *Evidence*. The *RP* can then verify that result and respond.

What is missing at this point is a trusted way of measuring the system for *Claims*. Having a separate trusted measurer is important, as we outlined in Section 2.3.2, so this must be implemented before using the protocol in production environments. This Proof-of-Concept (PoC) can serve as a reference or basis for another implementation. Future improvements in the Rust SDK for OP-TEE could also help make it easier to maintain.

Furthermore, the protocol can be implemented in other languages as well, due to its simplicity and the availability of existing libraries that provide the building blocks for this work. Such works should also aim to implement a Trusted Measurer to acquire evidence for the protocol and a mechanism to prevent *Trusted Applications* from running without performing attestation. Furthermore, the PoC shows a way to perform remote attestation, whereas future implementation may also consider implementing local attestation. The nature of this work does allow for such a use-case which can also be implemented in another work.

## 8.4 Measurement Results

The measurements outlined in Section 7.3 demonstrate that the protocol does not introduce severe overhead on a fast connection. The measurements extend up to 125 distinct pieces of *Evidence*, whereas a range between 10 to 20 would be more realistic, particularly considering the limited computational capacities of embedded devices. Nevertheless, these results show that the protocol itself can sustain higher numbers of evidence if necessary. Compared to the baseline without attestation it does introduce some overhead, but remote attestation always introduces some overhead due to the additional steps and network traffic.

As also discussed in that Section, the payload sizes lie between 183 and 311 bytes. The maximum number of bytes in a single TCP packet are 1500 bytes, so these payloads fit in one packet. However, future improvements to the implementation and packaging of the *Evidence* may be possible.

However, it is important to acknowledge that in real-world applications, the measurement process itself can also be time-consuming. Future implementations need to account for this aspect when integrating Trusted Measurers, as they could significantly influence the overall performance of the attestation process.

## 9 Conclusion

In this study, we have examined existing attestation mechanisms and devised a novel attestation protocol tailored for the Arm TrustZone environment. Noteworthy is the fact that our protocol leverages established technologies such as the Noise Protocol and Merkle Trees, further validated through formal verification via the Tamarin Prover. This protocol aligns closely with the specified goals outlined in Section 6.1. Section 5 provides an analysis of existing TrustZone attestation mechanisms, categorized based on predefined Functional and Security Features. Building upon this groundwork, Section 6 presents the PATAT Protocol, a novel attestation protocol using the Noise Protocol and Merkle Trees. Section 7.1 demonstrates its formal verification using the Tamarin Prover, affirming its integrity in maintaining both secrecy and authenticity. Lastly, Section 7.2 describes the implementation of a Proof-of-Concept application and the performance of said application which show that our work runs under 500ms when utilizing 125 or less pieces of *Evidence*.

## References

[1] F. Almeida, J. Duarte Santos, and J. Augusto Monteiro, "The Challenges and Opportunities in the Digitalization of Companies in a Post-COVID-19 World," *IEEE Engineering Management Review*, vol. 48, no. 3, pp. 97–103, 2020, ISSN: 1937-4178. DOI: `10.1109/EMR.2020.3013206`.

[2] M. Satyanarayanan, "The Emergence of Edge Computing," *Computer*, vol. 50, no. 1, pp. 30–39, Jan. 2017, ISSN: 0018-9162. DOI: `10.1109/MC.2017.9`. [Online]. Available: `http://ieeexplore.ieee.org/document/7807196/`.

[3] A. Ltd, *TrustZone for Cortex-A – Arm®*, Arm | The Architecture for the Digital World. [Online]. Available: `https://www.arm.com/technologies/trustzone-for-cortex-a`.

[4] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Computing Surveys*, vol. 51, no. 6, 130:1–130:36, Jan. 28, 2019, ISSN: 0360-0300. DOI: `10.1145/3291047`. [Online]. Available: `https://doi.org/10.1145/3291047`.

[5] A. Ltd, *TrustZone for Cortex-M – Arm®*, Arm | The Architecture for the Digital World. [Online]. Available: `https://www.arm.com/technologies/trustzone-for-cortex-m`.

[6] *Networking ecosystem ⋆ Scalys - NXP Layerscape TrustBox Edge*, Scalys. [Online]. Available: `https://scalys.com/solutions/networking-ecosystem/`.

[7] *Trustbox Family ⋆ Scalys - Cyber secure IoT Edge devices*, Scalys. [Online]. Available: `https://scalys.com/solutions/networking-ecosystem/trustbox-edge/`.

[8] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, Pittsburgh, PA, USA: IEEE, Nov. 2016, pp. 445–451, ISBN: 978-1-5090-4607-2. DOI: `10.1109/CIC.2016.065`. [Online]. Available: `http://ieeexplore.ieee.org/document/7809736/`.

[9] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted Execution Environment: What It is, and What It is Not,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland: IEEE, Aug. 2015, pp. 57–64, ISBN: 978-1-4673-7952-6. DOI: 10.1109/Trustcom.2015.357. [Online]. Available: http://ieeexplore.ieee.org/document/7345265/.

[10] *Trusted Execution Environment (TEE) Committee*, GlobalPlatform. [Online]. Available: https://globalplatform.org/technical-committees/trusted-execution-environment-tee-committee/.

[11] *Introduction to Trusted Execution Environments*, GlobalPlatform. [Online]. Available: https://globalplatform.org/resource-publication/introduction-to-trusted-execution-environments/.

[12] *Open Portable Trusted Execution Environment*, Linaro. [Online]. Available: https://www.op-tee.org/.

[13] V. Costan and S. Devadas, *Intel SGX Explained*, 2016. [Online]. Available: https://eprint.iacr.org/2016/086, preprint.

[14] *MultiZone Security TEE for RISC-V*, Hex Five Security, Jun. 30, 2019. [Online]. Available: https://hex-five.com/multizone-security-tee-riscv/.

[15] *CWE - CWE-121: Stack-based Buffer Overflow (4.10)*. [Online]. Available: https://cwe.mitre.org/data/definitions/121.html.

[16] *CWE - CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') (4.10)*. [Online]. Available: https://cwe.mitre.org/data/definitions/78.html.

[17] J. Ménétrey, C. Göttel, A. Khurshid, *et al.*, “Attestation Mechanisms for Trusted Execution Environments Demystified,” in *Distributed Applications and Interoperable Systems*, D. Eyers and S. Voulgaris, Eds., vol. 13272, Cham: Springer International Publishing, 2022, pp. 95–113, ISBN: 978-3-031-16092-9. DOI: 10.1007/978-3-031-16092-9_7. [Online]. Available: https://link.springer.com/10.1007/978-3-031-16092-9_7.

[18] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, “Remote ATtestation procedureS (RATS) Architecture,” RFC Editor, RFC9334, Jan. 2023, RFC9334. DOI: 10.17487/RFC9334. [Online]. Available: https://www.rfc-editor.org/info/rfc9334.

[19] G. Coker, J. Guttman, P. Loscocco, *et al.*, “Principles of remote attestation,” *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, Jun. 2011, ISSN: 1615-5262, 1615-5270. DOI: 10.1007/s10207-011-0124-7. [Online]. Available: http://link.springer.com/10.1007/s10207-011-0124-7.

[20] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, “SANCTUARY: ARMing TrustZone with User-space Enclaves,” in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019, ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23448. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_01A-1_Brasser_paper.pdf.

[21] *ARM Security Technology Building a Secure System using TrustZone Technology*. [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Software-Architecture/Booting-a-secure-system/Secure-boot.

[22] R. Wilkins and B. Richardson, “UEFI secure boot in modern computer security solutions,” in *UEFI Forum*, 2013.

[23] *About – Confidential Computing Consortium*. [Online]. Available: https://confidentialcomputing.io/about/.

[24] J. Ménétrey, C. Göttel, M. Pasin, P. Felber, and V. Schiavoni, *An Exploratory Study of Attestation Mechanisms for Trusted Execution Environments*, Apr. 15, 2022. arXiv: 2204.06790 [cs]. [Online]. Available: http://arxiv.org/abs/2204.06790, preprint.

[25] *Understanding the Confidential Containers Attestation Flow.* [Online]. Available: https://www.redhat.com/en/blog/understanding-confidential-containers-attestation-flow.

[26] H. Tschofenig, Y. Sheffer, P. Howard, I. Mihalcea, and Y. Deshpande, "Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," Internet Engineering Task Force, Internet Draft draft-fossati-tls-attestation-03, Mar. 13, 2023, 27 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-fossati-tls-attestation-03.

[27] C. Shepherd, R. N. Akram, and K. Markantonakis, "Establishing Mutually Trusted Channels for Remote Sensing Devices with Trusted Execution Environments," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, Reggio Calabria Italy: ACM, Aug. 29, 2017, pp. 1–10, ISBN: 978-1-4503-5257-4. DOI: 10.1145/3098954.3098971. [Online]. Available: https://dl.acm.org/doi/10.1145/3098954.3098971.

[28] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., red. by G. Goos, J. Hartmanis, D. Barstow, *et al.*, vol. 293, Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378, ISBN: 978-3-540-18796-7 978-3-540-48184-3. DOI: 10.1007/3-540-48184-2_32. [Online]. Available: http://link.springer.com/10.1007/3-540-48184-2_32.

[29] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System,"

[30] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient Sparse Merkle Trees," in *Secure IT Systems*, B. B. Brumley and J. Röning, Eds., vol. 10014, Cham: Springer International Publishing, 2016, pp. 199–215, ISBN: 978-3-319-47559-2 978-3-319-47560-8. DOI: 10.1007/978-3-319-47560-8_13. [Online]. Available: http://link.springer.com/10.1007/978-3-319-47560-8_13.

[31] E. Rescorla, "Diffie-Hellman Key Agreement Method," 2631, Jun. 1999, 13 pp. DOI: 10.17487/RFC2631. [Online]. Available: https://www.rfc-editor.org/info/rfc2631.

[32] J. Black, "Authenticated encryption," in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg, Ed., Boston, MA: Springer US, 2005, pp. 11–21, ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_15. [Online]. Available: https://doi.org/10.1007/0-387-23483-7_15.

[33] Y. Nir and A. Langley, *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, Jun. 2018. DOI: 10.17487/RFC8439. [Online]. Available: https://www.rfc-editor.org/info/rfc8439.

[34] J. A. Salowey, D. McGrew, and A. Choudhury, *AES Galois Counter Mode (GCM) Cipher Suites for TLS*, RFC 5288, Aug. 2008. DOI: 10.17487/RFC5288. [Online]. Available: https://www.rfc-editor.org/info/rfc5288.

[35] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties," in *2012 IEEE 25th Computer Security Foundations Symposium*, Jun. 2012, pp. 78–94. DOI: 10.1109/CSF.2012.25.

[36] David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt, *Tamarin Prover*, https://tamarin-prover.github.io/. [Online]. Available: https://tamarin-prover.github.io/.

[37] C. J. F. Cremers, "The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds., vol. 5123, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 414–418, ISBN: 978-3-540-70543-7 978-3-540-70545-1. DOI: 10.1007/978-3-540-70545-1_38. [Online]. Available: http://link.springer.com/10.1007/978-3-540-70545-1_38.

[38] J. Menetrey, M. Pasin, P. Felber, and V. Schiavoni, "WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, Bologna, Italy: IEEE, Jul. 2022, pp. 1177–1189, ISBN: 978-1-66547-177-0. DOI: `10.1109/ICDCS54860.2022.00116`. [Online]. Available: `https://ieeexplore.ieee.org/document/9912246/`.

[39] *WebAssembly.* [Online]. Available: `https://webassembly.org/`.

[40] Google, *Google Scholar.* [Online]. Available: `https://scholar.google.com/`.

[41] W. Li, H. Li, H. Chen, and Y. Xia, "AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, Florence Italy: ACM, May 18, 2015, pp. 75–88, ISBN: 978-1-4503-3494-5. DOI: `10.1145/2742647.2742676`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2742647.2742676`.

[42] T. Abera, N. Asokan, L. Davi, *et al.*, *C-FLAT: Control-FLow ATtestation for Embedded Systems Software*, Aug. 17, 2016. arXiv: `1605.07763 [cs]`. [Online]. Available: `http://arxiv.org/abs/1605.07763`, preprint.

[43] G. Scopelliti, S. Pouyanrad, J. Noorman, *et al.*, "End-to-End Security for Distributed Event-Driven Enclave Applications on Heterogeneous TEEs," *ACM Transactions on Privacy and Security*, Apr. 13, 2023, ISSN: 2471-2566. DOI: `10.1145/3592607`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3592607`.

[44] J. H. Ostergaard, E. Dushku, and N. Dragoni, "ERAMO: Effective Remote Attestation through Memory Offloading," in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, Rhodes, Greece: IEEE, Jul. 26, 2021, pp. 73–80, ISBN: 978-1-66540-285-9. DOI: `10.1109/CSR51186.2021.9527978`. [Online]. Available: `https://ieeexplore.ieee.org/document/9527978/`.

[45] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, Shanghai China: ACM, Oct. 14, 2017, pp. 287–305, ISBN: 978-1-4503-5085-3. DOI: `10.1145/3132747.3132782`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3132747.3132782`.

[46] S. Hristozov, J. Heyszl, S. Wagner, and G. Sigl, "Practical Runtime Attestation for Tiny IoT Devices," in *Proceedings 2018 Workshop on Decentralized IoT Security and Standards*, San Diego, CA: Internet Society, 2018, ISBN: 978-1-891562-51-8. DOI: `10.14722/diss.2018.23011`. [Online]. Available: `https://www.ndss-symposium.org/wp-content/uploads/2018/07/diss2018_11_Hristozov_paper.pdf`.

[47] M. Kylänpää and A. Rantala, "Remote Attestation for Embedded Systems," in *Security of Industrial Control Systems and Cyber Physical Systems*, A. Bécue, N. Cuppens-Boulahia, F. Cuppens, S. Katsikas, and C. Lambrinoudakis, Eds., vol. 9588, Cham: Springer International Publishing, 2016, pp. 79–92, ISBN: 978-3-319-40384-7 978-3-319-40385-4. DOI: `10.1007/978-3-319-40385-4_6`. [Online]. Available: `http://link.springer.com/10.1007/978-3-319-40385-4_6`.

[48] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London United Kingdom: ACM, Nov. 6, 2019, pp. 1723–1740, ISBN: 978-1-4503-6747-9. DOI: `10.1145/3319535.3363205`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3319535.3363205`.

[49] U. Lee and C. Park, "SofTEE: Software-Based Trusted Execution Environment for User Applications," *IEEE Access*,

vol. 8, pp. 121 874–121 888, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3006703. [Online]. Available: https://ieeexplore.ieee.org/document/9131703/.

[50] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, Berkeley, CA, USA: IEEE, 2004, pp. 272–282, ISBN: 978-0-7695-2136-7. DOI: 10.1109/SECPRI.2004.1301329. [Online]. Available: http://ieeexplore.ieee.org/document/1301329/.

[51] M. M. Quaresma, "TrustZone based Attestation in Secure Runtime Verification for Embedded Systems," 2020. [Online]. Available: https://mquaresma.github.io/assets/dissertation.pdf.

[52] I. D. O. Nunes, K. Eldefrawy, and N. Rattanavipanon, "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19, Santa Clara, CA, USA: USENIX Association, 2019, pp. 1429–1446, ISBN: 978-1-939133-06-9.

[53] J. H. Silverman, "An Introduction to the Theory of Elliptic Curves,"

[54] *The Noise Protocol Framework*. [Online]. Available: http://www.noiseprotocol.org/noise.html#dh-functions-cipher-functions-and-hash-functions.

[55] B. Dowling, P. Rösler, and J. Schwenk, "Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework," in *Public-Key Cryptography – PKC 2020*, A. Kiayias, M. Kohlweiss, P. Wallden, and V. Zikas, Eds., vol. 12110, Cham: Springer International Publishing, 2020, pp. 341–373, ISBN: 978-3-030-45373-2 978-3-030-45374-9. DOI: 10.1007/978-3-030-45374-9_12. [Online]. Available: https://link.springer.com/10.1007/978-3-030-45374-9_12.

[56] Meta, *WhatsApp Encryption Overview: Technical White Paper*. [Online]. Available: https://www.academia.edu/50744993/Whatsapp_encryption_overview_technical_white_paper.

[57] J. A. Donenfeld, *WireGuard: Fast, modern, secure VPN tunnel*. [Online]. Available: https://www.wireguard.com/.

[58] *Lightning Network In-Progress Specifications*, Lightning Network, Jun. 25, 2023. [Online]. Available: https://github.com/lightning/bolts.

[59] *Nijeboer, F.J. (Frank, Student M-CS) / PATAT Proof · GitLab*, GitLab, Sep. 7, 2023. [Online]. Available: https://gitlab.utwente.nl/s2011972/patat-proof.

[60] *Rust Teaclave TrustZone Sdk*, GitHub. [Online]. Available: https://github.com/apache/incubator-teaclave-trustzone-sdk.

[61] QEMU, *Qemu*. [Online]. Available: https://www.qemu.org/.

[62] *Snow - Rust*. [Online]. Available: https://docs.rs/snow/latest/snow/.

[63] *Nijeboer, F.J. (Frank, Student M-CS) / PATAT Protocol · GitHub*, GitHub. [Online]. Available: https://github.com/NijeboerFrank/patat-protocol.

[64] *Nijeboer, F.J. (Frank, Student M-CS) / PATAT Server · GitHub*, GitHub. [Online]. Available: https://github.com/NijeboerFrank/patat-server.