# TEE$^2$ - Combining Trusted Hardware to Enhance the Security of TEEs

**TEE$^2$ - Kombination vertrauenswürdiger Hardware zur Erhöhung der Sicherheit von TEEs**

Master-Thesis von Fritz Alder

December 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Security Engineering

# TEE² - Combining Trusted Hardware to Enhance the Security of TEEs
TEE² - Kombination vertrauenswürdiger Hardware zur Erhöhung der Sicherheit von TEEs

Vorgelegte Master-Thesis von Fritz Alder

1. Gutachten: Prof. Dr. Stefan Katzenbeisser
2. Gutachten: Prof. N. Asokan
Betreuer: Dr. Andrew Paverd

Tag der Einreichung:

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Fritz Alder, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

| | |
|---|---|
| Datum / Date: | Unterschrift / Signature: |
| ______________________ | ______________________________ |

## Abstract

Trusted Execution Environments (TEEs) have become common in end-user hardware over the recent years and are starting to see deployment in cloud computing environments. Utilizing TEEs for cloud computing has a wide range of benefits ranging from confidential computations to integrity protection of workloads. However, side-channel attacks against TEEs are a persistent challenge that can only partly be solved by secure software design of trusted applications. Especially recent attacks against TEEs had an impact on their trustworthiness for end users. We identify two fundamental types of TEE adversaries: Weak adversaries that compromise the confidentiality but not the integrity of a TEE, and strong adversaries that are able to completely undermine the confidentiality and integrity guarantees of an attacked TEE. While the first adversary is able to read out secret information from trusted applications running inside the TEE, the second adversary is even able to fully impersonate a TEE as he is able to fake functionality such as remote attestation. In this thesis, we explore how multiple TEEs, potentially from different vendors, can be combined to create a system that stays secure as long as one of the participating TEEs remains uncompromised. First, we define a model of an Ideal TEE that acts as an ideal baseline for a TEE that has no flaws and can not be compromised. Then we introduce our model of a Combined TEE which consists of multiple real-world TEEs that may have flaws in their design or implementation. We discuss a wide range of one party and two party protocols for our Combined TEE and compare them to the optimal version of the Ideal TEE. Each of the discussed protocols is evaluated in terms of its security properties against both weak and strong adversaries. Furthermore, we formally verify the security properties of a subset of these protocols using the Tamarin prover and provide a prototype implementation with Intel SGX. Our prototype implementation shows a linear overhead for adding TEEs to the Combined TEE and shows a small constant increase in overhead when moving from a weak to a strong adversary model.

## Acknowledgments

First of all, I thank Prof. Dr. Stefan Katzenbeisser and Prof. N. Asokan for making this thesis possible and allowing me to work on it from Finland. While my situation as a german student working remotely on his thesis is not usual, Prof. Katzenbeisser not only agreed to supervise the thesis but also took his time to actively contribute to it during regular calls. Additionally, Prof. Asokan gave me full freedom to work on my thesis and other research topics at the same time and sparked my interest to pursue a research oriented career. Thank you.
Furthermore, I thank Dr. Andrew Paverd for his invaluable insight and advice during my thesis and my overall time at Aalto. I thoroughly enjoyed working with him and am incredibly grateful that he endured early morning calls and remote discussions while being away from Finland.
I also thank all members of the Secure Systems Group at Aalto University for hosting me and making my time in Finland highly enjoyable. This explicitly includes (in alphabetical order) Arseny Kurnikov, Hans Liljestrand, Dr. Lachlan Gunn, and Thomas Nyman for feedback and interesting discussions regarding my thesis and platform security research in general. Next, I thank Koen Tange, Max Reuter, Dr. Niina Idänheimo, Ricardo Vieitez Parra, Dr. Samuel Marchal, Sebastian Szyller, and Vera Kortoci for making the workplace a little bit like home.
During the course of my thesis, I had the pleasure to discuss my work with multiple people for whose feedback I am highly thankful: Dr. Michael Steiner, Prof. Dr. Christopher Brzuska, Ilhan Gurel, Dr. Jian Liu, and Dr. Jan-Erik Ekberg.
Finally, I thank my family and Sonja Skiba for their never ending support, even under the stress of own theses and personal challenges.

# Contents

## List of Figures

## List of Tables

## List of Listings

## Acronyms

| | |
|---|---|
| 2PC | Two-Party Computation |
| AES | Advanced Encryption Standard |
| AES-GCM | Advanced Encryption Standard in Galois/Counter Mode |
| CAN | Controller Area Network |
| DoS | Denial of Service |
| ECDH | Elliptic Curve Diffie-Hellman |
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| EPC | Enclave Page Cache |
| EPID | Enhanced Privacy ID |
| GCM | Galois/Counter Mode |
| IC | Integrated Circuit |
| MPC | Multi-Party Computation |
| REE | Rich Execution Environment |
| SDK | Software Development Kit |
| SEV | Secure Encrypted Virtualization |
| SGX | Software Guard Extensions |
| TA | Trusted Application |
| TCB | Trusted Computing Base |
| TEE | Trusted Execution Environment |
| TTP | Trusted Third Party |

## Nomenclature

| | |
|---|---|
| $\mathcal{A}$ | Symbol for User A if there is more than one user |
| $\mathcal{B}$ | Symbol for User B if there is more than one user |
| $\mathcal{C}$ | Symbol for the Combined TEE |
| $\mathcal{I}$ | Symbol for the Ideal TEE |
| $\mathcal{M}$ | Malicious adversary as defined in Section 3.4 |
| $\mathcal{M}_W$ | Confidentiality but not integrity compromising adversary as defined in Section 3.4 |
| $\mathcal{M}_S$ | Integrity compromising malicious adversary as defined in Section 3.4 |
| $\mathcal{T}$ | Symbol for a real-world TEE that is part of $\mathcal{C}$ |
| $\mathcal{T}_M$ | Symbol for the compromised TEE that is controlled by $\mathcal{M}$ |
| $\mathcal{T}_S$ | Symbol for the secure, not compromised, TEE |
| $\mathcal{U}$ | Symbol for the user |
| | |
| $\mathbb{M}_{\mathcal{U}\to\mathcal{I}}$ | Message from user $\mathcal{U}$ to Ideal TEE instance $\mathcal{I}$ |
| $\mathbb{M}_{\mathcal{I}\to\mathcal{U}}$ | Message from Ideal TEE instance $\mathcal{I}$ to user $\mathcal{U}$ |
| $\mathbb{R}$ | Random number generation of $\mathcal{I}$ |
| $\mathbb{G}$ | Key generation of $\mathcal{I}$ |
| $\mathbb{D}$ | Decryption of a ciphertext of $\mathcal{I}$ |
| $\mathbb{S}$ | Signature creation of $\mathcal{I}$ |
| $\mathbb{SF}$ | Store and forward scheme between two users with $\mathcal{I}$ |
| $\mathbb{OT}_n^m$ | Oblivious transfer between two users with $\mathcal{I}$ |
| | |
| $\mathfrak{M}_{\mathcal{U}\to\mathcal{T}}$ | Message from user $\mathcal{U}$ to TEE $\mathcal{T}$ (element of $\mathcal{C}$) |
| $\mathfrak{M}^A_{\mathcal{U}\to\mathcal{T}}$ | Authenticated message from user $\mathcal{U}$ to TEE $\mathcal{T}$ (element of $\mathcal{C}$) |
| $\mathfrak{M}_{\mathcal{T}\to\mathcal{U}}$ | Message from TEE $\mathcal{T}$ (element of $\mathcal{C}$) to user $\mathcal{U}$ |
| $\mathfrak{M}^A_{\mathcal{T}\to\mathcal{U}}$ | Authenticated message from TEE $\mathcal{T}$ (element of $\mathcal{C}$) to user $\mathcal{U}$ |
| $\mathfrak{R}$ | Authenticated random number generation of $\mathcal{C}$ for a weak adversary |
| $\mathfrak{R}_{\mathcal{M}_S}$ | Authenticated random number generation of $\mathcal{C}$ for a strong adversary |
| $\mathfrak{G}$ | Authenticated key generation of $\mathcal{C}$ for a weak adversary |
| $\mathfrak{G}_{\mathcal{M}_S}$ | Authenticated key generation of $\mathcal{C}$ for a strong adversary |
| $\mathfrak{D}$ | Decryption of a ciphertext of $\mathcal{C}$ |
| $\mathfrak{S}$ | Signature creation of $\mathcal{C}$ |
| $\mathfrak{SF}$ | Store and forward scheme between two users with $\mathcal{C}$ |
| $\mathfrak{OT}_n^m$ | Oblivious transfer between two users with $\mathcal{C}$ |

# 1 Introduction

Trusted Execution Environments (TEEs) are isolated, integrity protected execution environments that are integrated into computing hardware for the purposes of performing security critical functions. This isolated environment can be used to ensure the integrity of code running within the TEE, and the confidentiality and integrity of data in the TEE. Remote parties can receive strong assurance of these TEE properties using remote attestation. TEEs have become widely deployed in several classes of computing devices such as smartphones and PC platforms. As such, TEEs can be applied in a wide range of scenarios, ranging from ensuring confidentiality in cloud computing to protecting data on personal devices such as smartphones. One TEE software vendor, Trustonic, recently reported surpassing the threshold of one billion devices with their TEE technology[1]. Intel deploys their own TEE named Intel Software Guard Extensions (SGX) on all Intel end-user processors since the Skylake generation (2015)[2]. Android utilizes ARM TrustZone-based TEEs and uses them for their Keymaster Trusted Application (TA) that protects their users' cryptographic keys[3]. While most TEEs are still only available on consumer hardware, enterprise demand to leverage TEEs in cloud computing resulted in Microsoft Azure introducing a platform called Confidential Computing[4] and IBM Cloud introducing Data-In-Use protection[5]. Both cloud providers use Intel SGX to provide TEE-based computations to their customers.

While the demand for trustworthy computations is high, this trust is difficult to establish between end-users and TAs[6]. Intel SGX for example has been under scrutiny since its announcement for design decisions such as its remote attestation mechanism [70] and researchers stated their doubts regarding the trust model surrounding SGX [18]. So far, researchers have demonstrated attacks against TAs running on nearly every widely deployed TEE that either required immediate action by the TA developer or even updates from the TEE vendor. This list includes AMD Secure Encrypted Virtualization (SEV) [56], ARM TrustZone [78], and Intel SGX [73]. Due to the inherent nature of a TEE, any uncovered vulnerability is damaging to its long-term trustworthiness.

In this thesis, we explore how multiple TEE instances can be combined in order to provide a secure and trustworthy service even under the assumption that a possible adversary can compromise any TEE. We realize security under such strong adversary by amalgamating multiple TEE instances into a Combined TEE that is secure as long as any one of the involved TEEs is uncompromised.

## 1.1 Problem overview

A wide array of research has demonstrated a range of vulnerabilities in most of the common TEEs. These vulnerabilities can roughly be classified into two categories. The first class of vulnerabilities compromises the confidentiality of the attacked TEE but it does not allow the adversary to impact its integrity. In other words, an adversary outside the TEE can read data that should normally only be accessible to the TEE, but cannot modify the TEE's behaviour. The second class of vulnerabilities compromises both the confidentiality and integrity of the TEE, allowing the attacker to change the behaviour of the TEE. While both classes are damaging to the user's trust into the TEE, the second class is especially damaging as it allows adversaries to impersonate a TEE completely. Considering a system model of a client communicating with a TEE through an untrusted host, the second class of vulnerabilities effectively eliminates the user's trust into the remote attestation mechanism provided by the TEE. One result of this is that neither the user nor any third party can trust output of such a TEE as they can not be sure if the TEE has been compromised by a strong adversary.

[1] https://www.trustonic.com/news/company/adoption-trustonic-security-platforms-passes-1-billion-device-milestone/
[2] https://ark.intel.com
[3] https://source.android.com/security/keystore
[4] https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/
[5] https://www.ibm.com/blogs/bluemix/2018/05/data-use-protection-ibm-cloud-using-intel-sgx/
[6] The term *trust* has many different meanings. In this thesis, we consistently use it as a synonym for trustworthy.

## 1.2 Solution overview

Similar to the concept of cryptographic and robust combiners introduced by Harnik et al. [33], we propose to combine multiple TEEs into one system. By combining TEEs from multiple different vendors or even multiple different implementations of the same protocol on the same TEE, we effectively construct a hardware combiner of TEEs that ensures security as long as any one of the participating TEEs is uncompromised. It is important to note that regular robust combiners base their security on the notion of redundancy, e.g. by encrypting any plaintext in succession with each involved cipher (also called the *Cascade* combiner). Assuming two of such ciphers, the security of the plaintext relies on either cipher being secure as the adversary needs to be able to decrypt both ciphers to regain the plaintext. This is fundamentally different to the hardware combiner approach that we are following when combining TEEs. We assume the strongest possible adversary model in this thesis whereby a compromise of a TEE means all secrets are leaked to an adversary. It is not possible to apply robust combiners to the problem setting in this case as one has to presume that the party performing the encryption may be compromised and may divulge secrets regardless of the security of the cipher. This creates the need for a system model where no single party is entrusted with the whole secret. Otherwise, if any of the TEEs is compromised and can leak its secrets to an adversary, all information that is known by this TEE is also known by the adversary. Using the same example of nesting an encryption of a plaintext, the adversary can regain the plaintext if he compromises the TEE that is supposed to perform the first encryption step. As such, no TEE can ever be trusted with an important part of the secret.
To prepare our solution, we first introduce the notion of an Ideal TEE that is not susceptible to any attacks and can as such be used as a baseline for protocols to realize functionalities with a TEE. We then present our design of a Combined TEE that consists out of multiple TEEs amalgamated into one system. All Combined TEE protocols are carefully designed to provide the same security as their analog Ideal TEE protocol as long as at least one TEE involved in the Combined TEE system remains uncompromised. This includes that the Combined TEE protocols never reveal the complete secret to any single TEE. Critically, we do not assume to know *which* TEE will remain uncompromised. This makes the problem significantly more challenging, but means that our solution can be applied to a larger class of real-world scenarios, for example, multi-party protocols where no single TEE is trusted by all participants. In addition to carefully defining our adversary scenario, we present multiple Combined TEE protocols, provide a prototype implementation, formally verify our solution, and finally perform performance evaluations of the prototype implementation.

## 1.3 Structure of the thesis

The remainder of this thesis is structured as follows. Chapter 2 gives the necessary background that is required for the topics discussed in this thesis. In Chapter 3 we introduce the notion of an Ideal TEE and discuss its system and adversary model. Based on the ideal model we introduce our design of a Combined TEE in Chapter 4. We follow this design by discussing multiple utility, one-party, and two-party protocols in Chapter 5. Each of these protocols is first discussed for the Ideal TEE and then realized with the Combined TEE. In Chapter 6 we present our prototype implementation of a Combined TEE based on Intel SGX. We present a formal model of a subset of the presented protocols in Chapter 7 and verify the core security properties using the Tamarin prover. We evaluate the performance of the given implementation in Chapter 8. Appendix A shows additional Ideal TEE protocols and Appendix B depicts the full source code of our Tamarin models that are presented in Chapter 7. Finally, we discuss related work in Chapter 9 and conclude the thesis with additional discussion in Chapter 10.
Table 1.1 gives an overview over all protocols discussed in this thesis, first for the Ideal TEE and then for the Combined TEE. It lists the name, the given symbol, the chapter in which each protocol is discussed, its figure, and also the pointer to the formal model of each protocol, if applicable. For the Combined TEE protocols, the table also points to the chapter where the prototype implementation is described.

**Table 1.1.:** Overview of discussed protocols and categorization of their status in this thesis

| Protocol | Symbol | Discussed | Figure | Formal model |
|---|---|---|---|---|
| **Ideal TEE protocols** | | | | B.1 |
| Key exchange | | 5.1.1 | 5.1 | 7.1.1 |
| Messages | $\mathbb{M}$ | 5.1.2 | A.1 & A.2 | 7.1 |
| Random number generation | $\mathbb{R}$ | 5.2.1 | A.3 | 7.1.2 |
| ElGamal operations | $\mathbb{G}$ & $\mathbb{D}$ | 5.2.2 | A.4 & A.5 | - |
| Signing | $\mathbb{S}$ | 5.2.3 | A.6 | - |
| Store and forward | $\mathbb{SF}$ | 5.3.1 | A.6 | - |
| Oblivious Transfer | $\mathbb{OT}^m_n$ | 5.3.2 | A.6 | - |

| Protocol | Symbol | Discussed | Figure | Formal model | Implemented |
|---|---|---|---|---|---|
| **Combined TEE protocols** | | | | B.2 & B.3 | |
| Key exchange | | 5.1.1 | 5.2 | 7.2.1 | 6.2 |
| Messages | $\mathfrak{M}$ | 5.1.2 | | 7.2.1 | 6.2 |
| - Authenticated version | $\mathfrak{M}^A$ | 5.1.2 | 5.3 & 5.4 | - | 6.2 |
| Authenticated random number generation | | | | | |
| - Weak adversary | $\mathfrak{R}$ | 5.2.1 | 5.5 | 7.2.2 | 6.2 |
| - Strong adversary | $\mathfrak{R}_{\mathcal{M}_S}$ | 5.2.1 | 5.6 | - | 6.2 |
| Authenticated key generation (ElGamal) | | | | | |
| - Weak adversary | $\mathfrak{G}$ | 5.2.2 | 5.7 | - | - |
| - Strong adversary | $\mathfrak{G}_{\mathcal{M}_S}$ | 5.2.2 | 5.8 | - | - |
| ElGamal decryption | $\mathfrak{D}$ | 5.2.2 | 5.9 | - | - |
| Signing | $\mathfrak{S}$ | 5.2.3 | 5.10 | - | 6.2 |
| Store and forward | $\mathfrak{SF}$ | 5.3.1 | 5.11 | - | - |
| Oblivious Transfer | $\mathfrak{OT}^m_n$ | 5.3.2 | 5.12 & 5.13 | - | - |

# 2 Background

We split this section into three parts. First, we give the necessary cryptographic background for this thesis. Next, we explain TEEs and finally complete this chapter by discussing real-world TEEs.

## 2.1 Cryptographic background

To understand the cryptographic elements that stand at the core of this thesis, we explain in the following the essentials of a Diffie-Hellman key exchange, elliptic curve ElGamal encryption, and the Goldwasser-Micali encryption.

### 2.1.1 Diffie-Hellman key exchange

The Diffie-Hellman key exchange often serves as the basis of encrypted communication between two parties [69]. Taking a finite abelian group $G$ of prime order $q$ as base, the two communication partners agree on a generator $g \in G$. Using the two parties Alice and Bob, Alice chooses a random $a \in G$ and calculates $A = g^a$ while Bob chooses a random $b \in G$ and calculates $B = g^b$. $A$ and $B$ are the two public keys of Alice and Bob and can be exchanged. Once Alice receives Bob's public key, she can calculate $K = B^a = g^{ba} = g^{ab}$ while Bob can calculate $K = A^b = g^{ab}$. Based on this shared key, Alice and Bob can employ a key derivation function to calculate a common session key to secure their communication with a symmetric encryption scheme. It is also possible to base the Diffie-Hellman key exchange on elliptic curves instead of on a group with a prime order [69, 44].

### 2.1.2 Elliptic curve ElGamal encryption

Based on the construction of Koblitz [44], we utilize an elliptic curve variation of the ElGamal encryption scheme [21]. In the following, we use a similar notation to the one used by Koblitz [44]. For any cryptographic operation with elliptic curve ElGamal, one party first chooses common parameters that are publicly known and serve as a base for all involved parties. The first of these parameters are a large prime $p$ and an integer $n \in \mathbb{N}$ with which we compute $q = p^n$. Next, an elliptic curve $E$ in $GF(q)$ and a point $G \in E$ that lies on this curve are chosen.
To generate a key pair, the receiver $\mathcal{B}$ first chooses his private key $X$ randomly and publishes the Point $X \cdot G$ as his public key. In order to send a message to $\mathcal{B}$, user $\mathcal{A}$ first has to map the message $m$ to a point on the curve. To map $m$ to such a Point, we assume an injective, easily invertible function $f$ that is publicly known. As such, $\mathcal{A}$ and $\mathcal{B}$ are assumed to have access to a function $f : \{0,1\}^l \rightarrow GF(q)$ with $l \in \mathbb{N}$ that performs this mapping. The construction of $f$ is out of scope of this background section, but there exist several of such functions, for example some near optimal constructions as shown in [24].
After mapping the message $m$ to its point on the curve, $\mathcal{A}$ chooses a random integer $k \in \mathbb{N}$ and sends the tuple $(k \cdot G, f(m) + k \cdot (a \cdot G))$ to $\mathcal{B}$. $\mathcal{B}$ can recover the message by multiplying the first parameter with his secret $X$ and subtracting it from the second parameter:

$$f(m) + k \cdot (a \cdot G) - k \cdot X \cdot G = f(m)$$

Afterwards, $\mathcal{B}$ calculates $f^{-1}(f(m)) = m$ to retrieve the message.
In the following, we use the following notation:

- All parties are assumed to have knowledge of a tuple of domain parameters $\lambda$ that consist of $\lambda = (p, q, n, E, G)$.

- The private key is denoted as $X$ and the public key as $Y = X \cdot G$.
- Ciphertexts are denoted as tuples $C = (C_1, C_2)$ with $C_1 = k \cdot G$ and $C_2 = f(m) + k \cdot (a \cdot G)$.
- We omit the use of function $f$ whenever it is clear that we are discussing message $m$ and assume that the use of function $f$ to create a mapping from a bit string to a point on $E$ is implied whenever $m$ is mentioned.

### 2.1.3 Goldwasser-Micali encryption

The Goldwasser-Micali encryption was presented by Goldwasser and Micali in 1984 [66]. It is a probablistic public-key encryption and is based on the difficulty of the quadratic resudiosity problem. To generate the keys, the receiver chooses two large primes $p$ and $q$ and calculates $N = p \cdot q$. Next he chooses two values $y_p \in \mathbb{F}_p^*$ and $y_q \in \mathbb{F}_q^*$ such that the Legendre symbols satisfy $\left(\frac{y_p}{p}\right) = \left(\frac{y_q}{q}\right) = -1$. With these two values, he can retrieve $y$ from $y_p$ and $y_q$ via the Chinese Remainder Theorem [69]. The private key is now $(p, q)$ while the public key is $(y, N)$. To encrypt a message $m$ consisting of the bits $m_0, ..., m_n$, the user chooses one random number $x_i \ mod \ N$ per bit and calculates the bitwise ciphertexts $c_i = y^{m_i} \cdot x_i^2 \ mod \ N$. To decrypt a bit of the message, the receiver only needs to find out if $c_i$ is a quadratic residuo modulo $N$. If $c_i$ is a quadratic residuo, this means that $m_i$ was zero, otherwise it was one.
Crucial for the use in this thesis is the homomorphic property of the Goldwasser-Micali encryption. Given two bits $m_0$ and $m_1$ with their respective ciphertexts $c_0$ and $c_1$, the multiplication of the ciphertexts $c_0 \cdot c_1 \ mod \ N$ is equal to the encryption of the exclusive or (XOR) of the two plaintext bits $m_0 \oplus m_1$. As such, it is possible to use the Goldwasser-Micali encryption as a homomorphic encryption respective to XOR by multiplying two ciphertexts.

## 2.2 Trusted Execution Environments

A TEE provides an isolated execution environment to its users that is typically shielded from the normal operation on the same machine and allows its users to perform secure operations. This shielding prevents attacks from other, untrusted areas on the machine such as the normal operating system, also called the Rich Execution Environment (REE). Within the isolated TEE environment, specific regions of code and data can be allocated and programs, called TAs, can be loaded and executed. On the surface, this is similar to the protection that virtual machines aim to provide to its users with the difference that a TEE typically enforces this protection on the hardware level. This means that there is often a separate processor or a specific TEE mode on the processor that can be enabled to switch from the REE to the isolated TEE mode. In this mode, the REE, i.e. the normal operating system, is not in control anymore and can not read the memory that is controlled by the TEE. Based on prior definitions of Vasudevan et al. [75] and Global Platform [28] [29], the four core security properties of a TEE are as follows:

- Isolated execution
- Code integrity
- Sealed Storage
- TEE attestation

### 2.2.1 Isolated execution

*Isolated execution* guarantees that the TA is executed within an isolated environment. The security guarantees of this isolated environment can be summarized with three features: Memory protection, code isolation, and protected flow of execution.

Memory protection guarantees the confidentiality of the TA's data by protecting its memory from any access of the REE. This can be realized by encrypting all data that is written to TEE memory and decrypting it again when the data is accessed. If the data is encrypted, it is necessary to use an authenticated encryption scheme to prevent an attacker from tampering with encrypted data stored in memory. Another option is to predefine certain regions of memory that are not accessible from the REE.
Code isolation prevents an attacker from gaining control over the execution flow of the code that the TA runs. This means that a TA can only be entered from the REE at precisely specified code addresses. Any other jumps into the code are not allowed and are prevented by the TEE. This prevents attacks such as return oriented programming that target the control flow of a TA from the REE.
Lastly, protected flow of execution guarantees that even if the execution is interrupted by the REE, when the execution continues, it continues at the correct and intended code locations without the REE being able to tamper or change the state of the TA between interrupts. However, it might be possible for the REE to prevent a program from terminating by interrupting it and preventing it to continue. It can be impossible for the TEE to ensure that its TAs terminate.

### 2.2.2 Code integrity

*Code integrity* ensures that the code that the user intends to run in the TEE is being executed without modifications. If such a guarantee could not be given to a callee of a TA, then an adversary with control over the REE can swap or extend the code that is loaded into the TEE with code that he controls. The callee would expect his trusted code to be run while the adversary is actually able to control the execution and the processing of all inputs into the TA. To protect the code integrity, some TEEs employ digital signatures of TAs. After compilation of a trusted program, the developer digitally signs a hash of the TA code. After the TEE loaded the TA code, it performs the same hashing operation and verifies the TA's signature before execution. If the digital signature is not valid or if the loaded code is not equal to the signed hash, the TEE refuses the execution.

### 2.2.3 Sealed storage

Another security property of the TEE is *sealed storage*. If the TEE has access to storage on the host machine, the TEE can use it to store encrytpted data. The encryption keys for this process are usually derived from a hardware secret that is provisioned during manufacturing. Access to the hardware key is only possible under specific circumstances, such as the processor being in the secure TEE mode. This process of encrypting data with a key that is only accessible within a strictly defined configuration or set of environmental variables is called *sealing*. A TEE can provide sealing operations to TAs and as such give them access to persistent, secure storage. As any memory of the trusted environment becomes inaccessible once the host machine shuts down, persistent storage is the only way for a trusted program to retain data across reboots. The TEE guarantees that the data that is sealed is secret and can not be read or inferred by any untrusted party, especially the adversary. Typically this is done by using an authenticated, symmetric encryption scheme such as Advanced Encryption Standard in Galois/Counter Mode (AES-GCM). Additionally, the TEE can prevent TAs from accessing sealed data of other TAs by deriving a unique set of sealing keys for each program running inside the TEE. This gives all TAs access to sealed storage but prevents potentially malicious interactions between two programs running inside the same TEE.
It is important to note that sealed storage does not necessarily provide freshness of the data and does not guarantee its availability. As the TEE is relying on the REE to provide the sealed data, there is no guarantee that the REE will provide the most recent version of the data, or any data at all. While programs that make use of sealed storage can implement a custom system based on the features of the underlying TEE, e.g. by using monotonic counters or utilizing time stamps, the challenge of data availability is not solvable by a trusted program. As such, TAs have to carefully design their use of sealed storage. Also note, that sealed storage can not be used by clients to encrypt arbitrary data for the TEE as input as only the TEE has access to the sealing keys if all requirements are met (e.g. the processor is in the TEE mode and the specific code that is bound to this sealing key is loaded).

### 2.2.4 TEE attestation

The last property, *TEE attestation*, enables the TEE to convince a remote party about its internal state and about the TA that is being executed. One essential challenge in trusted computing is to establish trust between a trusted component and remote parties that do not have direct access to the hardware. Remote attestation is a process that allows remote parties to verify the internal state of a TEE and of specific TAs. Typically, the code running within the TEE is hashed and then digitally signed with a secure key held by the TEE. This secure key is signed by the hardware manufacturer during provisioning of the TEE keys and can be verified by remote parties either directly or with the help of the hardware manufacturer or TEE vendor. To protect the confidentiality of attestation keys, they can be controlled by a very small Trusted Computing Base (TCB) which only handles the measurement and attestation of TAs. Remote attestations can also additionally allow the TA to add a small amount of user data to an attestation message which can be used as a trustworthy channel to the remote party. Such trustworthy channels can be used to perform a key exchange and establish a session key between the remote party and a TA which can then be used to communicate without further remote attestations being necessary.

## 2.3 Real-world Trusted Execution Environments

Several research projects have aimed to design and develop implementations of Trusted Execution Environments, some of which are Haven [9], Graphene [71], Sancus [60], and more recently the RISC-V based Sanctum [19] and its open-source based successor Keystone[1]. Additionally, there exist several real-world implementations of TEEs that are deployed in end-user hardware such as Intel SGX [18], AMD SEV [42], and also ARM TrustZone [6] based TEEs like Trustonic Kinibi[2], Op-TEE[3], or Android Trusty TEE[4]. While the research in this thesis aims to be applicable to any of these TEEs, we focus on one specific implementation of a TEE, namely Intel SGX. In this chapter, we first give an introduction to Intel SGX and then present recent attacks that have been demonstrated against real-world TEEs.

### 2.3.1 Intel Software Guard Extensions (SGX)

Intel Software Guard Extensions (SGX) is a set of CPU instructions available on all customer grade processors since the Skylake generation first sold in 2015[5]. At its core, SGX is a TEE that allows users to load arbitrary code and data into trusted regions called *enclaves* which provide isolation against the REE, in this case the normal operating system [52]. For this, Intel SGX-enabled processors provide a set of CPU instructions that handle the interaction with the TEE and also allow the untrusted OS to retain control over the execution of programs on its system. To load an enclave, the untrusted OS can use the CPU instructions ECREATE, EADD, and EINIT to create an enclave, add code pages to the Enclave Page Cache (EPC), and finalize the loading process and initialize the enclave for execution [37]. The last operation in the creation provess, EINIT, performs a check whether the loaded enclave is equal to the program code that was digitally signed by its developer during development. These equality checks are performed by verifying the value of the MRENCLAVE value [3] which is a hash of all code pages that are loaded into the enclave. SGX only proceeds with a successful enclave initialization if the MRENCLAVE value matches the expected value that is signed by the developer.
After initialization of an enclave, programs in the user space of the untrusted OS can use the CPU instructions EENTER, EEXIT, and ERESUME to enter the execution of the enclave, exit it, and resume execution after an interrupt occured. Entering an enclave via the EENTER command is only allowed through predefined call gates named ECALLs. These ECALLs are defined at compile time and enforced by SGX by verifying the desired address of each EENTER instruction.

[1] https://keystone-enclave.org/
[2] https://www.trustonic.com/solutions/
[3] https://www.op-tee.org/
[4] https://source.android.com/security/trusty
[5] https://ark.intel.com

This allows enclaves to prevent undesired code execution outside of precisely defined function boundaries [39]. Inputs into such an ECALL are boundary checked by the SGX state machine as part of the process of entering the enclave to prevent common run-time attacks such as buffer overflows and all data is copied into the trusted environment by the SGX drivers. Within the execution of an enclave, the executed code can not perform any interaction with code that is not already loaded into the EPC. This essentially prevents the use of dynamic libraries as these would be loaded or accessed after the enclave has been loaded into the EPC. Instead, enclaves can only rely on static libraries that are loaded together with the remainder of the enclave code before execution. In order to still allow the enclave to perform I/O operations with the untrusted OS, Intel SGX provides an additional feature called OCALLs. OCALLs are similar to ECALLs as they are specified during compile time by the developer. In contrast to ECALLs however, OCALLs allow the enclave to exit the secure execution mode and receive input from an outside function. As such, OCALLs are the inverse of an ECALL by giving the trusted code access to a specific function in the untrusted part of the application. Once the untrusted part returns from the execution of an OCALL, any returned result is boundary checked and copied into the trusted environment where the enclave can resume execution [40].

In addition to ECALLs and OCALLs, Intel SGX enables enclaves to perform *sealing* operations. Sealing is integrated into the Intel SGX architecture and allows an enclave to encrypt arbitrary data to a key that is bound to the enclave's MRENCLAVE value. This means that no other enclave is able to retrieve the sealing key necessary to decrypt any data sealed by another enclave. SGX realizes this by deriving the sealing key from the hardware secret that is embedded into the CPU during manufacturing. This hardware secret is combined with the MRENCLAVE value of the current enclave in order to retrieve the sealing key of that enclave [3].

Another built in feature of SGX is local and remote attestation. Local attestation allows an enclave to create a REPORT structure that contains 512 byte of user data and that can be used by another enclave to verify the integrity of its peer. The REPORT structure can be targeted at another MRENCLAVE value and SGX enforces that only the intended enclave can verify the report. Two communicating enclaves can exchange these REPORT structures in order to perform a local attestation and assure themselves of the identity of their peer while performing a key exchange with the user data in the REPORT structure. This process of local attestation can also be used by an enclave to attest itself to one of the core architectural enclaves of SGX, the *quoting enclave*. The quoting enclave holds a hardware embedded secret that is signed by Intel during manufacturing and that can be used to sign a REPORT structure [41]. Such a signed report can then be verified by remote parties using the Intel Attestation Service. This can be used to convince the remote party about the MRENCLAVE value of the originating enclave, effectively attesting the enclave state to the remote party. Similar to local attestation, the user data in a remote attestation report can be used to perform an attested key exchange between a remote party and an Intel SGX enclave. By using a system called Enhanced Privacy ID (EPID), Intel SGX ensures that a remote attestation can not be traced back to one specific machine [13]. This prevents fingerprinting of devices that perform multiple remote attestations over the course of their lifetime.

### 2.3.2 Attacks against real-world TEEs

Researchers have demonstrated many attacks against the implementations or the overall designs of real-world TEE products, such as for ARM TrustZone [78], Intel SGX [12, 74, 47, 15], and AMD SEV [56, 36]. Most of the attacks are side-channel attacks like cache side-channel attacks [32], timing side-channel attacks [27], and other variations [77]. While some TEEs classify side-channel attacks as out of scope of their security model, this does not allow TEEs to be used in environments in which they might be subject to side-channel attacks (e.g. a cloud provider that is not fully trusted). Shinde et al. [68] described pidgeonhole attacks against Haven [9] where the untrusted OS prepares data pages in a manner that lets it monitor the page faults that are triggered by the TA. They demonstrated that such page fault side-channels can leak up to 100% of the secret bits in OpenSSL and Libcrypt. Similarly, Xu et al. [77] presented controlled channel attacks that track page faults of both code and data pages and that allow the untrusted OS to compromise the execution flow of the attacked program. Defenses to both attacks for Intel SGX enclaves were described by Shih [67] who presented T-SGX, a compiler-level defense against side-channel attacks. Chen et al. [16] presented Déjá Vu which achieves a similar protection for SGX enclaves through the use of a reference clock based on Intel Transactional Synchronization Extensions. Later, Brasser et al. [12] demonstrated that cache attacks against SGX are realistic and were able to extract RSA keys from protected enclaves. Lee et al. [47] presented branch-shadowing side-channel attacks that leak the executed branches of the targeted TA. If the chosen branches are dependent on a secret, the branch-shadowing attack is able to deduce some or all bits of the secret. Hosseinzadeh et al. [38] presented an improved defense to this attack that also accounts for stronger adversary capabilities based on work by Van Bulck et al. [72]. The attacks called Spectre [45] and Meltdown [48] additionally introduced speculative execution side-channels that work by tricking the victim process into speculatively executing code that would not normally occur. The attack then allows the adversary to deduce confidential information via a side-channel from this speculative execution. A first applicable attack of Spectre on Intel SGX was performed by Chen et al. [15] who demonstrated the possibility of leaking data and temporarily altering the control flow of the targeted enclave. Van Bulck et al. [73, 76] then developed Foreshadow which allows an attacker to fully compromise an Intel SGX enclave. The Foreshadow attack was the first attack on Intel SGX that not only compromised the confidentiality but also the integrity of the underlying TEE. This is crucial as all previous attacks were only able to leak secret information out of a TA running inside a TEE while Foreshadow was able to leak information out of architectural enclaves of Intel SGX which have a very small TCB and are provisioned during manufacturing. The difference is that while other side-channel attacks might leak confidential information of a TA, they do not leak secrets that are protected by the core architecture of the TEE. Such core architectural secrets include the keys for remote attestation and the machine's hardware secrets that are used to derive sealing keys. If an adversary gains control over e.g. the remote attestation keys, he is able to completely impersonate an Intel SGX TEE without the necessity of actually executing the code inside a TEE. As such, side-channel attacks in general and especially strong side-channel attacks like Foreshadow limit the use of TEEs to environments in which these attacks are unlikely to be exploited.

# 3 System and Adversary Model

First, we introduce the features of an Ideal TEE as a baseline representation of a TEE which we use below to compare other TEE solutions to the ideal version of a TEE. Then, we discuss the system model a TEE operates in and state the adversary goals and capabilities. We conclude this chapter by defining the security requirements that any real-world system has to meet in order to achieve the same security guarantees as an Ideal TEE.

## 3.1 Ideal TEE

In Section 2.2 we already described the functionality of a Trusted Execution Environment (TEE) by combining prior definitions of Vasudevan et al. [75] and Global Platform [28] [29]. There, we described a TEE through its four core security properties:

- Isolated execution (Section 2.2.1)
- Code integrity (Section 2.2.2)
- Sealed storage (Section 2.2.3)
- TEE attestation (Section 2.2.4)

We use these core properties to introduce the concept of an *Ideal* TEE that correctly and securely implements all features of a TEE. In this sense, an Ideal TEE has all four mentioned properties and implements them without flaws. In the context of the Ideal TEE, we define the four TEE security properties as follows:

**Ideal TEE Security Property 1** (Isolated execution)**.** *If the execution completes, then the resulting state of the system will be equivalent to the case in which the program executed in the absence of any adversary. Additionally, any memory used by the system remains confidential and can not be read or accessed by the adversary at any point in time.*

**Ideal TEE Security Property 2** (Code integrity)**.** *If the execution completes (which is not guaranteed), then the executed program was identical to the unmodified program that the caller specified at call time.*

**Ideal TEE Security Property 3** (Sealed storage)**.** *Data placed in persistent storage by the Ideal TEE cannot be read by the adversary (confidentiality) and any modifications can be detected by the TEE (integrity). There is, however, no guarantee that the data is available to the Ideal TEE at all times.*

**Ideal TEE Security Property 4** (Ideal TEE attestation)**.** *A remote party can verify the integrity of outputs of an Ideal TEE and is able to associate them to one specific TEE instance.*

It is important to note that the Ideal TEE does not provide any availability guarantees. Specifically, there is no guarantee that the execution will complete, nor that the sealed storage will be available, nor that the remote attestation protocol will successfully complete.

## 3.2 System model

The typical setting for a TEE is depicted in Figure 3.1. A client wants to communicate with a TEE over a secure channel in order to use it for his purpose. At the same time, the setting places the TEE on a host platform that is neither trusted by the TEE nor by the client that communicates with it. In fact, the adversary might have full access to the host of the TEE and as such it needs to encrypt all communication with the client to hide it from the host.

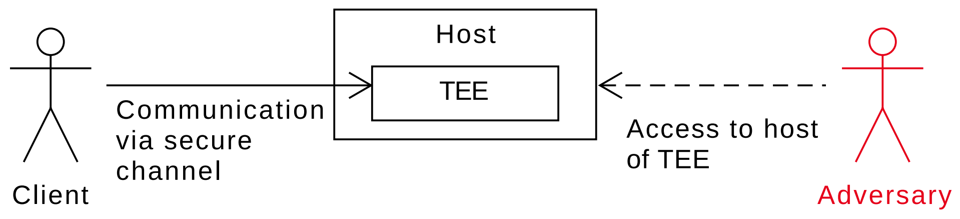


**Figure 3.1.:** TEE system model: A client uses and communicates with a TEE through an untrusted network and operating system. The adversary has full control over the network and the untrusted operating system, but is assumed to be interested in maintaining the service and a good reputation.

In this scenario, the client initially performs an attested key exchange with the TEE and can then send encrypted inputs to let it perform certain operations. After processing, the TEE responds to the client with some encrypted output. Since the client can not communicate directly with the TEE, he sends the actual messages over the network to the untrusted host who is requested to forward those messages for the client. While the untrusted host can easily drop messages and deny the TEE service, we assume that he has an interest in maintaining the service to the client, such as economic interests or to hide the presence of the adversary from the client. Note, that the system model used in this thesis is based on utilizing TEEs in a cloud computing environment. Use of TEEs in e.g. mobile phones might have different requirements which do not fit this system model such as device binding of cryptographic keys.

## 3.3 Adversary goals

According to the system model, the client uses the TEE as a sort of service that is provided by a service provider. Such a TEE oriented service would be hosted by a service provider who is remunerated by clients that in turn get access to the TEE. As such, possible adversaries are an external attacker, the service provider himself, or in a multi-tenant setting, another client of the same or another TEE. Since all of these potential adversaries could collaborate, we combine them in one strong adversary model. The result is an adversary that tries to undermine any of the previously defined TEE (or specifically Ideal TEE) properties while avoiding to disrupt the availability of the service.

**Adversary goals.** *The adversary aims to undermine any of the four properties of a TEE. In doing so, the adversary is not interested in diminishing the availability of the TEE-based service.*

We briefly give examples for the motivation of some possible attackers and how they might want to subvert the TEE properties:

- *Service provider:* The goals of a malicious service provider might be twofold. On the one hand, he might be interested in maximizing his profits by either reducing the amount of work the TEE performs, or by replacing it with a non-genuine TEE that impersonates a genuine one. On the other hand, the service provider might be interested in stealing the user's data and use it for his own purpose. While the second goal comes close to the notion of a malicious-but-cautious attacker [64], the first goal allows the service provider to perform any action as long as he stays undetected and maintains the service to the user. Remaining undetected is the principal concern for the service provider in order to retain his reputation and business.

- *Other Ideal TEE users:* There exist two possible users that can be seen as adversaries in the described setting: A co-located user and an active communication partner of the attacked user:
  - *Co-located user:* The first malicious user is using the same TEE but besides this has no relation to the targeted user. The goals of such a malicious co-located user is to use his access to the same TEE to gain information

or perform tasks that should be limited to another user. If, for example, a co-located user is able to use cryptographic keys of the victim, he does not need to know the victim's keys to practically have broken the confidentiality guarantee of the TEE.

  – *(Protocol) Peer:* The second malicious user is a peer of the victim in a two- or multi-party protocol and aims to cheat or gain advantage in this protocol by undermining the TEE security properties. As it is possible to use protocols in which the TEE takes on the role of a Trusted Third Party (TTP), peers can gain a crucial advantage if they compromise the TTP.

- *External attacker:* An external attacker has no direct involvement with other stakeholders of the system, and as such is not concerned about his reputation or being detected after successfully completing an attack. As such, he has the most flexible goals. For the purposes of the system model we assume an external attacker aims to subvert any of the Ideal TEE properties defined in Section 3.1.

Note, that all messages that are sent from the client to the TEE are passed through the untrusted host. As such, the host has full control over the messages sent through the network and he can tamper with, delay, or completely drop those messages. This would allow the adversary to completely run a Denial of Service (DoS) attack on the service of the TEE as he is assumed to have control over the network and the host. Because this can not be avoided within the mentioned setting, we assume that the adversary is interested in maintaining the service of the TEE to the client and is not interested in performing a DoS attack. This is realistic if one thinks about the system model in a setting similar to cloud computing operations such as IaaS, PaaS, or SaaS [54]. This even holds true for the external attacker. While he might not be concerned about the availability of the TEE, he is actively working against the service provider who aims to maintain the service to his users. Since it is not possible to maintain the availability of such a service with an attacker present that can drop messages, we conclude that even for the external attacker, disrupting the availability is not in scope as he would quickly raise suspicion by the service provider and lose his advantage point for attacks.

## 3.4 Adversary capabilities with respect to real-world TEEs

In the scope of this work, we assume an active malicious adversary that is computationally bounded but can eavesdrop all interaction between the TEEs and the client. As such, we base our adversary on the Dolev-Yao model and assume that he has full access and control over all inputs and outputs of the TEE, as well as over the networking of the setup. In addition to this standard adversary model, we also allow the attacker certain capabilities against TEEs. Based on the attacks on real-world TEEs as described in Section 2.3.2, we split the adversary capabilities into two subclasses: Strong attackers that can compromise the confidentiality and integrity of the TEE, and weak attackers that can compromise the confidentiality but not integrity of the TEE.

### Strong attacker compromises TEE integrity and confidentiality

The first attacker can fully compromise a TEE by reading all data that it handles and which effectively enables the attacker to impersonate the TEE at his will. Such capabilities can be the result of a full breach with side-channel attacks or speculative execution attacks like the Foreshadow [73] attack against Intel SGX. Since the attacker can read out all TEE internal secrets, he is even able to fake core functionality such as TEE attestation. This is a very strong adversary model and essentially reduces the TEE to a normal party that can not be trusted in any way as it is completely under the control of the attacker.

### Weak attacker compromises TEE confidentiality but not integrity

In the second model, the adversary can compromise the confidentiality but not the integrity of the TEEs. After the compromise, the adversary can read out all secrets relating to the current operation of the TEE, but can not compromise core architectural secrets such as TEE attestation keys. One example is a side-channel attack which reads out the encryption keys used by a TA running inside a TEE. This side-channel attack only compromises the specific TA within a TEE but does not compromise the underlying architecture of a TEE. This has been demonstrated in various cache side-channel attacks against various TEEs, such as [78], [12], or [42]. The result of such a successful attack is that all secrets of the targeted program were to be leaked to the adversary, but as he did not compromise the underlying TEE architecture, he would not be able to impersonate the TEE to a remote party.

Since the first notion of the attacker's capabilities (compromised integrity) leads to a stronger attacker model, we mostly use the strong adversary in the following sections and assume it to be the main adversary model in this thesis. However, some protocols can either not be realized at all if the TEE integrity is compromised, or can be realized with considerably lower complexity if the weaker adversary is assumed. Thus, we always discuss the situation in the presence of the weaker attacker first and then explain what steps have to be taken to also account for the stronger adversary.

## 3.5 Security requirements

Our aim is to design a real-world system that provides the same security properties as the Ideal TEE. The following definition specifies the requirements on such real-world system in order to achieve the same functionality as the Ideal TEE.

**Security requirements.** *A real-world system is said to achieve the functionality of an Ideal TEE, if and only if, given the adversary capabilities defined in Section 3.4, any attack that is possible against the real-world system is also possible against the Ideal TEE.*

In other words, an external observer who can only observe the adversary's progress towards achieving the adversary goals (as defined in Section 3.3) cannot distinguish between the real-world system and an Ideal TEE. This places the attacker against a real-world system on the same level as the attacker against the Ideal TEE and makes them comparable in the scope of the underlying system model.
Note, that the mentioned security requirement does not state that any program needs to be secure within the context of the real-world system. As a matter of fact, this is not even the case for the Ideal TEE. There, the security properties given by the Ideal TEE only guarantee that the underlying architecture is secure. However, attacks against the programs running inside the TEEs are not in scope from an architectural perspective, as the adversary can still perform run-time attacks or utilize timing side-channels to achieve his goals. The only statement that can be made from an architectural perspective is that it is possible to design a program that runs within the Ideal TEE and to which the Ideal TEE adds certain security properties that can not be undermined by the adversary. Taking this into account, it is not required that the real-world system prevent all types of attacks such as run-time attacks. It is only necessary that any attack that is possible against a program running in the real-world system is also possible against the same program running in the Ideal TEE. This excludes any program specific vulnerabilities and focuses on the core-architectural question of the system's security.

# 4 Design

We present our design of a Combined TEE that aims to achieve the same security guarantees as an Ideal TEE by combining several real-world TEEs to one system. This serves as a preparation for the protocols that we discuss in Section 5 where we dedicate a chapter to a wide range of functionalities and how these functions can be achieved with the Combined TEE.

The *Combined* TEE is a system of two or more real-world TEEs that aims to achieve the same security properties as an Ideal TEE. At its core, a Combined TEE can be seen as similar to the concept of robust combiners introduced by Harnik et al. [33] as both approaches aim to achieve a secure system from potentially insecure parts. Considering the *Cascade* combiner discussed by Herzberg et al. [35, 34], a robust combiner uses multiple cryptographic operations in succession where the output of the first operation serves as input into the second operation. If at any given point in time one of the involved cryptographic operations becomes insecure, the robust combiner ensures the security as it cascaded the input through multiple different operations. We use a similar mechanism by combining multiple TEEs to provide security even if one of them becomes compromised. It is crucial to note that in contrast to most robust combiners and especially to the cascade combiner, a Combined TEE can never reveal parts of the secret to any of its participating TEEs. For a Combined TEE, which is made up of real-world TEEs, we assume that its TEEs may have flaws which result in any of them being compromised. This stands in contrast to the Ideal TEE where we assume that the system has no flaws. Given this adversary model, an attacker only needs to compromise the first TEE in a cascade scheme to gain crucial information about the secret. The goal of the Combined TEE however is to still guarantee the security of the overall system under the premise that parts of it are compromised. Since the side-channel attacks mentioned in Section 2.3.2 are mostly specific to a single implementation of a TEE, it is reasonable to assume that the attacker is not able to simultaneously compromise multiple TEEs if the client utilizes TEEs from different vendors. We use the same adversary model and goals for the Combined TEE that we described above (see Section 3.3 and 3.4). The Combined TEE is designed to achieve the same security properties as the Ideal TEE if any one of the real-world TEEs remains uncompromised.

For clarity, we discuss the Combined TEE in a configuration that only has two TEEs (in the following named TEE$^2$). However, all of the discussed protocols can be extended to an arbitrary amount of participating TEEs (in the following named TEE$^N$) and we explain such extensions in Section 5.4. First, we describe the system model of the Combined TEE and follow this by the security guarantees that need to be achieved by the Combined TEE.

## 4.1 System model

Figure 4.1 gives an overview of the system model for the context of our Combined TEE. Similar to the general TEE system model described above, a client has access to a system of TEEs and wants to communicate with it in order to utilize it for his computations. In contrast to the general TEE model however, this system consists of two separate untrusted hosts that run two separate real-world TEEs. The client can communicate with each untrusted host independently and the hosts can independently forward messages to and from their TEE to the client. Additionally, the adversary is assumed to have full control over both untrusted hosts and can compromise the two TEEs at will. Essentially, we assume that the adversary is unbounded and can perform any compromise. Based on this scenario, we will then later show how security can still be achieved as long as at least one TEE remains uncompromised.

If needed, the TEEs can communicate with each other by passing messages through the client. This simplifies the system model as there is no need for an additional physical channel between the TEEs. It also simplifies the security evaluation of the protocols as we already need to assume that one of the TEEs is leaking its secrets to the client (since the adversary may be one of the clients) and as such giving the user control over the communication between the TEEs is a stronger assumption than a direct communication channel.

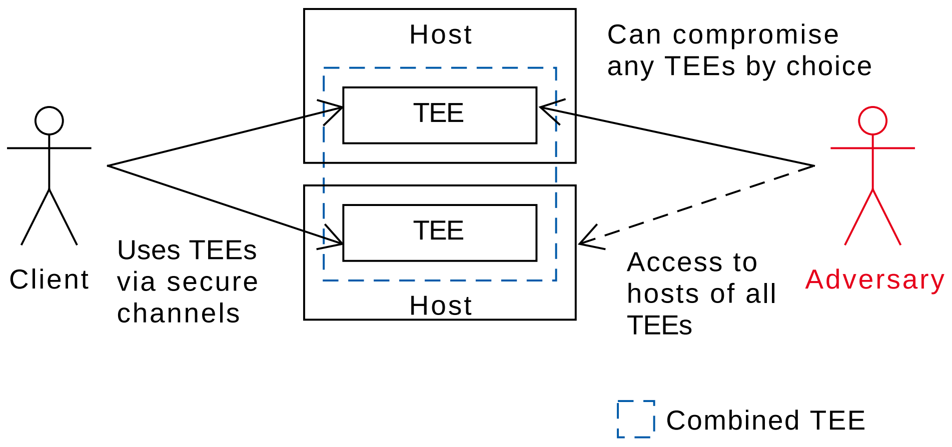

Host
Can compromise any TEEs by choice
TEE
TEE
Client
Uses TEEs via secure channels
Host
Access to hosts of all TEEs
Adversary
Combined TEE

**Figure 4.1.:** Combined TEE system model: Two separate TEEs are used by a client through an untrusted network and untrusted operating systems. Similar to Figure 3.1, the adversary has full control over the network and the untrusted operating systems, but is assumed to be interested in maintaining the service and a good reputation. However additionally to the previous Figure, the adversary can also compromise the two TEEs at his will. The Combined TEE is designed to still provide its security properties in case one of the TEEs remains uncompromised.

## 4.2 Security properties of the Combined TEE

The security that is provided by the Combined TEE depends upon the number of TEEs that the adversary has compromised. As long as one element of the Combined TEE remains uncompromised, the system can guarantee the same security as the Ideal TEE described above. This, however, requires that there exists one TEE involved in the Combined TEE that is never compromised by the adversary throughout the complete run of a protocol. If several protocols are linked together and executed in succession, they need to be seen as one protocol as the adversary can trivially combine knowledge from multiple protocols (e.g. a session key) to break the overall security of the system. While the attacker can choose to compromise any of the TEEs at any time he chooses, the client remains unaware of this choice. This strengthens the security model of the Combined TEE by lowering the assumptions the user is able make about the system. In the context of TEE$^N$, the attacker can compromise up to $N-1$ TEEs, while again at least one TEE needs to stay uncompromised for the Combined TEE to retain is security properties. In the context of TEE$^N$, the user is also neither aware of which TEE remains uncompromised, nor how many TEEs remain uncompromised. Therefore, our fundamental security property is as follows:

*The Combined TEE is secure as long as at least one of its involved TEEs remains uncompromised throughout a protocol run. Any other TEE participating in the Combined TEE can be compromised without the Combined TEE losing its security properties.*

In a real-world setting, it is reasonable to assume a Combined TEE setup where a user considers at least one of the participating TEEs as uncompromised while all other TEEs might not be trusted. We discuss three scenarios with such a disjoint trust setting.

#### Users trust different vendors

Assume the case of two communicating parties who want to execute two-party protocols through the Combined TEE. As long as each user trusts at least one of the involved TEEs, she can trust the whole Combined TEE system. Note however, that in order for both users to trust the Combined TEE, they do not necessarily need to trust the same TEE. Consider a scenario where both users only trust a specific TEE vendor and have control over their own TEE. Both users want to include their own TEE in the Combined TEE but do not trust the TEE vendor of their communication partner. Since no single TEE is trusted by both parties, they normally resolve to traditional cryptographic protocols that take this lack of trust into account. However, as both parties assume their own TEE to be uncompromised, they can utilize the Combined TEE which can ensure the security properties.

**Side-channel attacks compromise different TEEs**

Most of the side-channel attacks discussed in Chapter 2.3.2 only affect one specfic TEE. Some attacks have implications on multiple TEEs but are effectively only compromising the TEE of one specific vendor. If one constructs a Combined TEE out of multiple different TEEs of different type, it is reasonable to assume that any newly discovered side-channel attack is not able to compromise all TEEs at the same time. While one of the involved TEEs might then be compromised by this side-channel attack, there remains at least one TEE that is uncompromised. This allows the user to keep using the Combined TEE until the TEE vendor is able to protect against the side-channel attack and after updating the affected TEE to a secure version, the user can regenerate all keys to rely on the security of all TEEs again. It is reasonable to assume that an attacker is not able to find multiple new side-channel attacks against all involved TEEs at the same time if the Combined TEE is chosen carefully and diverse. Especially if the user can reestablish her security after resecuring her TEEs, it becomes highly difficult to affect all TEEs at the same time as the attacker can not reuse learned knowledge of an older TEE compromise.

**Users trust different implementations**

In addition to two users trusting different TEE vendors, it is also possible that multiple users trust different protocol implementations on the same TEE. Consider again the case of two users that want to utilize two-party protocols. While they both trust the same TEE vendor, they do not trust the other user's implementation of the desired protocol. Such a scenario is realistic for complex implementations where no user can easily verify that the other party's implementation is not leaking secrets or is influenced by the user's input. A simple solution is for both users to have their own implementation that they trust while both implementations adhere to the overall protocol. Since the users trust the underlying TEE, they can remotely attest the code that is running in the other user's TEE and can place their trust into their own implementation on their TEE. As the Combined TEE is secure as long as one of the underlying protocol implementations is secure, the users can both deploy their own implementation and trust the resulting combined system.

# 5 Protocols

With the established design of the Combined TEE, we now describe a variety of utility protocols, one-party protocols, and two-party protocols. For each of the discussed functions, we first explain a protocol that solves the function with the Ideal TEE to give a security baseline and an intuitive approach to the challenges that each function brings. Based on these Ideal TEE protocols we then describe the same functionality within the context of a Combined TEE and discuss how to perform the same task with a weak or a strong attacker present. The main goal of the Combined TEE protocols is to provide the same security guarantees as the corresponding Ideal TEE protocol. Thoroughly introducing and discussing the protocols of the Ideal and Combined TEE in this chapter allows us to compare the approaches below and reason about the security properties that they provide. We base such reasoning on the security requirements as well as the weak and strong adversary we defined in Section 3.5. In this section, we point out limitations of each function that exist for both the Ideal and Combined TEE to explicitly define the context in which each protocol has to provide security guarantees.

The first step in any communication with a TEE is to establish a secure channel between the user and the TEE. As such, we first explain how to perform a key exchange with the Ideal TEE and the Combined TEE and follow this by discussing how to communicate with these TEEs using such an encrypted channel. While the key exchange is nearly identical for the Ideal and Combined TEE, the resulting communication that is based on such key exchange already varies depending on the adversary model in use. After these utility functions, we explain several one party functions that are of interest if used as part of a larger system. The first of these functions is random number generation which is fairly straightforward with the Ideal TEE but raises some challenges for the Combined TEE. The second set of one party functions are key generation and decryption for the elliptic curve ElGamal cryptographic scheme, as introduced in Section 2.1.2. The last function discussed for a single participant is a signature generation performed by the TEE. Finally, we discuss two party protocols that have two participants utilizing the TEE at the same time, namely policy-based store-and-forward and oblivious transfer. We conclude the chapter by explaining how these protocols can be extended to the case of TEE$^N$.

**Notes on Ideal TEE protocols**

If the user has access to an Ideal TEE, he can use it in a similar manner to a TTP where, after establishing a session key, the TTP can provide certain services that are otherwise more difficult to achieve without access to a trusted party. Some examples of such simple services are the authenticated random number generation or cryptographic operations that can be proven to be originating from the TTP. As the adversary has no access to any confidential data held within the Ideal TEE, most of its protocols have a very low complexity.

**Notes on Combined TEE protocols**

In contrast to the Ideal TEE, in the context of a Combined TEE we can not assume that any specific TEE is uncompromised. Instead, every TEE has to be treated equally and only by combining all parts of the Combined TEE, security can be guaranteed and trust established. The protocol design in this context is strongly influenced by the fact that it is not known which of the TEEs can be trusted. For sake of simplicity, we discuss all protocols in the base setting of two involved TEEs (TEE$^2$). However, it is possible to extend all discussed protocols to the general case of TEE$^N$ and we dedicate the last part of this section to explain how such extension would be done for each protocol.

**Nomenclature**

In the following, we use the nomenclature as defined in Table 5.1. We refer to the Combined TEE that consists of two real-world TEEs as $\mathcal{C}$ and refer to each of the two real-world TEEs as $\mathcal{T}$. In cases where we need to distinguish between the two $\mathcal{T}$ instances, i.e. between the instance that is compromised and the one that is not, we refer to the compromised TEE as $\mathcal{T}_M$ and the other one as $\mathcal{T}_S$. Note, that this nomenclature does not result in a loss of generality, as we do not specify beforehand which of the two TEEs is compromised and which one is not. While this distinction is clear in the case of TEE$^2$, it can be generalized for the case of TEE$^N$ if $\mathcal{T}_M$ is seen as a set and $\mathcal{T}_S$ is seen as a single, uncompromised TEE. The Ideal TEE has the symbol $\mathcal{I}$, the malicious adversary $\mathcal{M}$, and the client as a user of the system $\mathcal{U}$. If the protocol has more than one user, we instead use the symbols $\mathcal{A}$ and $\mathcal{B}$ for user A and user B respectively. When discussing the actions of the adversary, we use $\mathcal{M}$ to denote a general adversary as introduced in Section 3.4, and use the specific notation $\mathcal{M}_S$ to denote the strong attacker that can compromise the confidentiality and integrity of the TEEs, and $\mathcal{M}_W$ to denote the weak attacker that can compromise the confidentiality but not the integrity of the TEEs. This simplifies the distinction between adversary models if there are consequences for the discussed protocols.

**Table 5.1.:** Notation used in the following chapters

| | |
|---|---|
| General malicious adversary | $\mathcal{M}$ |
| Integrity compromising adversary | $\mathcal{M}_S$ |
| Confidentiality compromising adversary | $\mathcal{M}_W$ |
| Client / User of the TEE | $\mathcal{U}$ |
| User A | $\mathcal{A}$ |
| User B | $\mathcal{B}$ |
| Ideal TEE instance | $\mathcal{I}$ |
| Combined TEE instance | $\mathcal{C}$ |
| Real-world TEE (part of $\mathcal{C}$) | $\mathcal{T}$ |
| Compromised actual TEE | $\mathcal{T}_M$ |
| Non-compromised actual TEE (secure TEE) | $\mathcal{T}_S$ |

**Protocol functions**

Table 5.2 lists functions that we use in the protocols of the following sections. TEEs can use the function $quote_{\mathcal{T}}(x)$ to create a quote that contains a cryptographic hash of the value $x$. Such a quote is used during remote attestation and can be used by the remote party to verify the integrity of the response that contains $x$ and link this response to the specific TEE $\mathcal{T}$. $Sig_A\{x\}$ denotes a signature of $x$ with the secret key of A while $Enc_A\{x\}$ denotes a public key encryption with the public key of A. For symmetric encryption, $\{x\}_K$ denotes an encryption of $x$ with the symmetric key $K$. $H(x)$ is used to indicate a cryptographic hash of x and $x \overset{\$}{\leftarrow} \mathcal{D}$ denotes that $x$ is randomly sampled from a domain $\mathcal{D}$ (e.g. such domain could be $\{0,1\}^n$ to sample $n$ random bits).

## 5.1 Utility protocols

Before any functionality can be realized with the TEE, a secure and authenticated channel needs to be established and a secure way of communication through this channel is to be defined. Thus, we first describe the utility protocols of authenticated key exchange and various versions of messaging between a user and a TEE and use these utility protocols in all future protocols.

### 5.1.1 Key exchange

The first step in any communication with a TEE as described in our system models is to establish a key exchange that can be linked to one specific TEE instance $\mathcal{I}$ or $\mathcal{C}$. The key exchange allows the user $\mathcal{U}$ to set up an ephemeral shared key

**Table 5.2.:** Functions used in the following chapters

| | |
|---|---|
| $quote_{\mathcal{T}}(x)$ | Signed statement binding the software identity of the TEE $\mathcal{T}$ to a hash of value $x$ provided by that TEE. A quote can be linked to one specific TEE. |
| $\{x\}_K$ | Symmetric encryption of $x$ with shared key $K$ |
| $Enc_A\{x\}$ | Public-key encryption of $x$ with public key of A |
| $Sig_A\{x\}$ | Signature of a hash of $x$ with the secret key of A |
| $H(x)$ | Cryptographic hash of $x$ |
| $x \overset{\$}{\leftarrow} \mathcal{D}$ | Random sampling of $x$ from Domain $\mathcal{D}$ |
| $x \overset{\$}{\leftarrow} \{0,1\}^n$ | Random sampling of $x$ with $n$ bits. |

which can be used to derive a session key to later encrypt messages. Encrypting the messages hides the content of $\mathcal{U}$'s communication with the TEE from the adversary and allow the TEE to link all messages that it receives to one specific user as the key was derived from an ephemeral key that is linked to $\mathcal{U}$'s public signing key. In this thesis, we employ an authenticated Diffie-Hellman key exchange as introduced in Section 2.1.1 and leverage a public signing key pair on the user's side and publicly verifiable quotes on the TEEs' side for the authentication.

**Key exchange with Ideal TEE (Figure 5.1)**

At the start of the protocol, the user is assumed to have access to a signing key pair and a fresh nonce that will be used to protect against replay attacks. The protocol is then initiated by the user $\mathcal{U}$ who sends a *request_key_exchange* message to the Ideal TEE $\mathcal{I}$, including the user's public key and nonce. After receiving a key exchange request message, $\mathcal{I}$ chooses a fresh random value $a$ and computes $g^a$, where $g$ is a public generator, prepares a quote over $g^a$ and the received public key and nonce, and sends this quote together with $g^a$ back to $\mathcal{U}$. The quote protects $\mathcal{U}$ from man-in-the-middle attacks where the adversary impersonates the Ideal TEE by changing the contents of the message before passing it to $\mathcal{U}$, by exchanging the actual $g^a$ for a key he controls. As the quote can be verified by $\mathcal{U}$, she can verify the integrity of the message content and trace the origin of the message back to $\mathcal{I}$. As the quote also includes the public signing key and the nonce of $\mathcal{U}$, she can be sure that $\mathcal{I}$ is responding to her original initial message and the attacker has not been replaying or modifying messages in between. After checking the correctness of the received quote, $\mathcal{U}$ generates her own $b$ parameter and can calculate the shared key $K$. $\mathcal{U}$ concludes the protocol by sending $g^b$ to $\mathcal{I}$ together with a signature over $g^b$ and $g^a$. After verifying that the signature is valid and signed by the same signing key that was specified in the original *request_key_exchange* message, $\mathcal{I}$ can also calculate the shared key $K$ and complete the protocol. The signature in the last message from $\mathcal{U}$ to $\mathcal{I}$ serves three purposes. First, it verifies the integrity of $g^a$ that is sent together with the signature. Second, it allows $\mathcal{I}$ to verify that the last message and the initial message were sent by the same party. This is crucial to ensure on the side of the $\mathcal{I}$ to prevent an attacker from performing an attack in the middle of the protocol. Lastly, the signature includes $g^b$ as a nonce of $\mathcal{I}$ to prevent replay attacks.

Note, that $\mathcal{I}$ only identifies $\mathcal{U}$ through her public signing key $pk_{\mathcal{U}}$. In doing this, $\mathcal{I}$ has no guarantee that it communicates with the actual user $\mathcal{U}$ and instead can only link messages to the same sender (assuming the corresponding private key has not been shared). This means that the public key is used as a proxy for actual authentication here where the Ideal TEE does not actually authenticate a user but merely a public key that is representing one specific entity. The crucial part about this concept is that any user who communicates with an Ideal TEE needs to verify that her public key is included in the quote produced by $\mathcal{I}$. If $\mathcal{U}$ detects that the signing key that was returned by $\mathcal{I}$ is not identical to her own public signing key, she has to abort the protocol as there has been a middle man that tampered with the message that was passed to $\mathcal{I}$. Effectively, using the public signing key as a proxy for authentication makes the system more flexible as the Ideal TEE

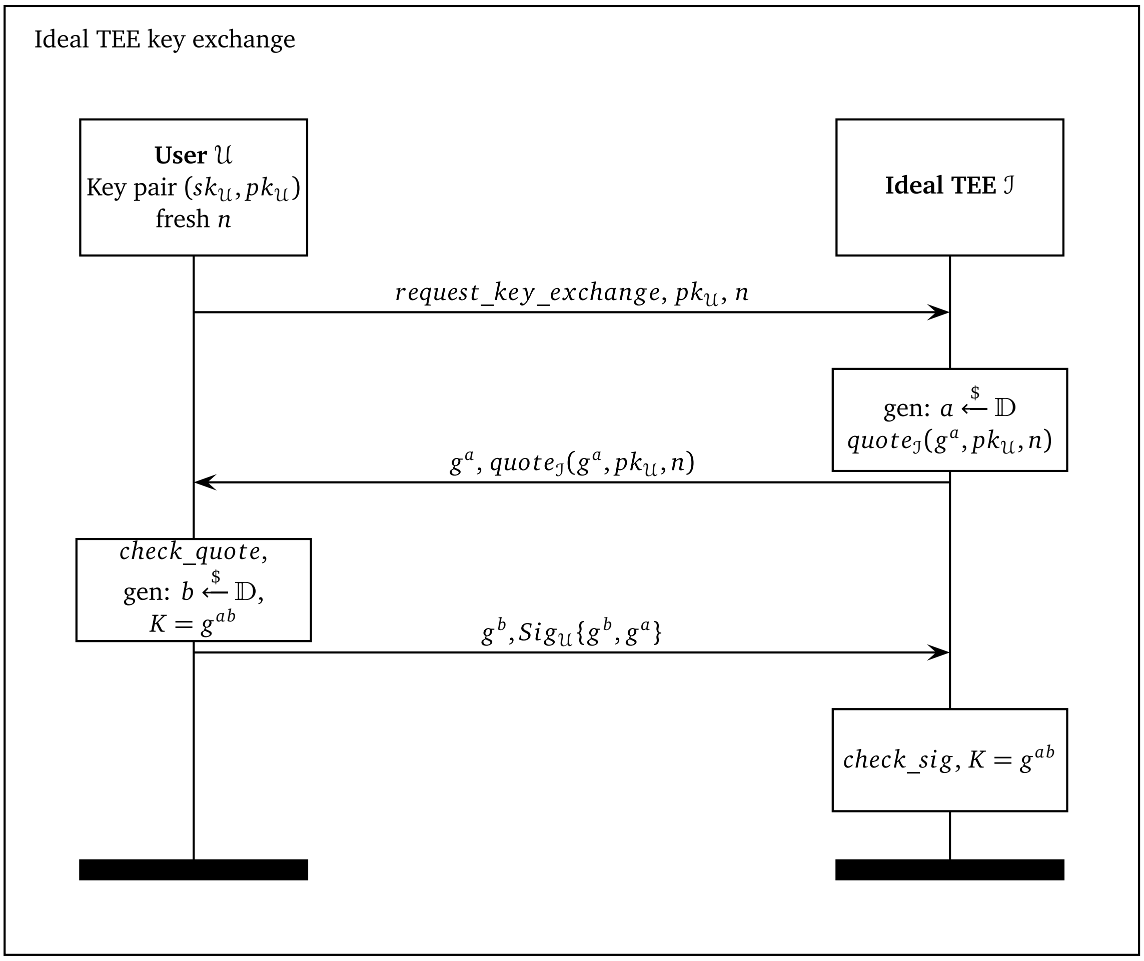

Ideal TEE key exchange
**User** $\mathcal{U}$
Key pair $(sk_{\mathcal{U}}, pk_{\mathcal{U}})$
fresh $n$
**Ideal TEE** $\mathcal{I}$
$request\_key\_exchange, pk_{\mathcal{U}}, n$
gen: $a \xleftarrow{\$} \mathbb{D}$
$quote_{\mathcal{I}}(g^a, pk_{\mathcal{U}}, n)$
$g^a, quote_{\mathcal{I}}(g^a, pk_{\mathcal{U}}, n)$
$check\_quote,$
gen: $b \xleftarrow{\$} \mathbb{D},$
$K = g^{ab}$
$g^b, Sig_{\mathcal{U}}\{g^b, g^a\}$
$check\_sig, K = g^{ab}$

**Figure 5.1.:** Diffie-Hellman key exchange between a user $\mathcal{U}$ and an instance of the Ideal TEE $\mathcal{I}$. $g^a$ is used as nonce of $\mathcal{I}$ while $n$ is used as a nonce of $\mathcal{U}$. The quote ensures $\mathcal{U}$ that she is communicating with a valid Ideal TEE while the signature is ensuring $\mathcal{I}$ that its communication partner does not change during the run of the protocol.

does not need to know all users that communicate with the system beforehand but it also places more responsibility on the users who now have to check if the response they receive from the Ideal TEE contains the expected parameters.

#### Key exchange with Combined TEE (Figure 5.2)

Figure 5.2 shows a key exchange between a user $\mathcal{U}$ and a Combined TEE $\mathcal{C}$. This key exchange is nearly identical to the Ideal TEE key exchange presented above. The main distinction is that instead of establishing one session key with a single TEE, $\mathcal{U}$ instead establishes a vector of two session keys that represent the secure communication with each part of $\mathcal{C}$. From the perspective of $\mathcal{U}$, the shared key is not a single secret anymore that is used to communicate with the TEE but instead consists of several secrets that are used to communicate with every element of the TEE $\mathcal{T}$ separately. As such, $\mathcal{U}$ initiates the protocol by sending two *request_key_exchange* messages, one to $\mathcal{T}_1$ and one to $\mathcal{T}_2$.

Again, these initial messages contain the user's public signing key and a nonce for each TEE to ensure freshness. Both TEEs then generate their Diffie-Hellman private key, create a quote over the public key and the parameters received in the initial message, and return the quote together with their own public key back to $\mathcal{U}$. After receiving responses from both TEEs, $\mathcal{U}$ first verifies both quotes. Additionally, in $\mathcal{C}$, $\mathcal{U}$ also needs to verify that both quotes come from different TEEs. This is essential to the security of the Combined TEE as the malicious adversary $\mathcal{M}$ could impersonate a Combined TEE with one compromised TEE if he can convince $\mathcal{U}$ to communicate twice with the same TEE. After performing the quote checks, $\mathcal{U}$ generates her own private Diffie-Hellman keys for $\mathcal{T}_1$ and $\mathcal{T}_2$, calculates her key vector $K = [k_1, k_2]$ and sends $g^{b_i}$ together with a signature over $g^{a_i}$ and $g^{b_i}$ back to the TEEs. After checking the signature, the TEEs can also

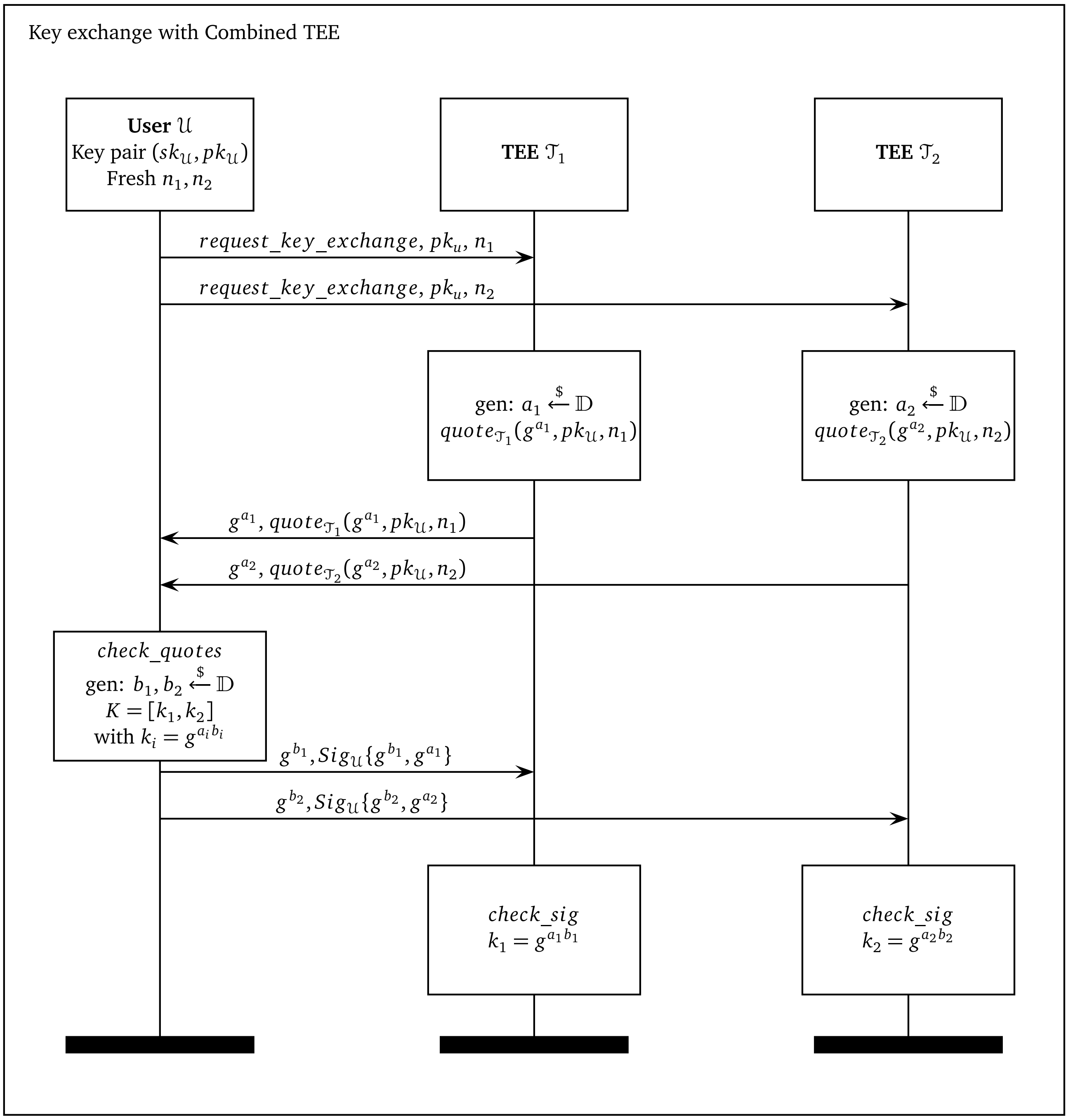

Key exchange with Combined TEE
User $\mathcal{U}$
Key pair $(sk_{\mathcal{U}}, pk_{\mathcal{U}})$
Fresh $n_1, n_2$
TEE $\mathcal{T}_1$
TEE $\mathcal{T}_2$
$request\_key\_exchange, pk_u, n_1$
$request\_key\_exchange, pk_u, n_2$
gen: $a_1 \xleftarrow{\$} \mathbb{D}$
$quote_{\mathcal{T}_1}(g^{a_1}, pk_{\mathcal{U}}, n_1)$
gen: $a_2 \xleftarrow{\$} \mathbb{D}$
$quote_{\mathcal{T}_2}(g^{a_2}, pk_{\mathcal{U}}, n_2)$
$g^{a_1}, quote_{\mathcal{T}_1}(g^{a_1}, pk_{\mathcal{U}}, n_1)$
$g^{a_2}, quote_{\mathcal{T}_2}(g^{a_2}, pk_{\mathcal{U}}, n_2)$
$check\_quotes$
gen: $b_1, b_2 \xleftarrow{\$} \mathbb{D}$
$K = [k_1, k_2]$
with $k_i = g^{a_i b_i}$
$g^{b_1}, Sig_{\mathcal{U}}\{g^{b_1}, g^{a_1}\}$
$g^{b_2}, Sig_{\mathcal{U}}\{g^{b_2}, g^{a_2}\}$
$check\_sig$
$k_1 = g^{a_1 b_1}$
$check\_sig$
$k_2 = g^{a_2 b_2}$

**Figure 5.2.:** Key exchange between a user $\mathcal{U}$ and the Combined TEE with its two elements $\mathcal{T}_1$ and $\mathcal{T}_2$. $g^{a_i}$ is used as nonce of $\mathcal{T}_i$ while $n_i$ is used as a nonce of $\mathcal{U}$. This key exchange is the same key exchange as used in the Ideal TEE, executed for each element of the Combined TEE. Checking the quotes implies that $\mathcal{U}$ verifies that she is talking to two different TEEs. User $\mathcal{U}$ ends up with $K$ which is a vector of keys that represents the shared key with the Combined TEE.

calculate the session keys with $\mathcal{U}$. Identical to the key exchange of the Ideal TEE, the protocol is replay protected through the use of a nonce on $\mathcal{U}$'s side and through returning the TEEs public key back to the TEEs.

This protocol provides the same authentication properties as its counterpart from the Ideal TEE where the TEEs only receive a public signing key as an identity and can only guarantee that the session key is bound to the initial owner of this session key. In contrast to the key exchange in the Ideal TEE however, in the Combined TEE the user either does not have the guarantee that she established a key with two TEEs (in the presence of an integrity compromising adversary $\mathcal{M}_S$), or she does not have the guarantee that all of those established keys are secret (in the presence of a confidentiality compromising adversary $\mathcal{M}_W$). We discuss how to deal with both of these adversaries during normal communication in the next section.

### 5.1.2 Messaging

After establishing a session key with a TEE, $\mathcal{U}$ can use this shared secret to encrypt messages. This ensures the confidentiality of the messages and protects the content of the messages from tampering by $\mathcal{M}$. To protect against replay attacks, it is assumed that $\mathcal{U}$ and the TEE both have access to a sequence number $n$ that is used in all messages and is checked and incremented upon receiving a message. While communication and implementation errors could create a state in which these sequence numbers mismatch, we assume that this is out of scope as the surrounding implementation and system would be responsible for properly handling the sequence numbers. While encrypted messages are trivial for the case of the Ideal TEE, they become more complex for the case of the Combined TEE. First, $\mathcal{U}$ wants to communicate with the complete system $\mathcal{C}$ but has to distinguish each Combined TEE element $\mathcal{T}_i$ when sending actual messages to the Combined TEE elements. Secondly, depending on the adversary model, precautions might have to be taken to prevent the malicious party $\mathcal{M}$ from faking messages in the message flow if he has control over one element of the Combined TEE (denoted as $\mathcal{T}_M$).
We start by describing the process of communicating with the Ideal TEE through encrypted messages and then discuss the same communication in the presence of an attacker $\mathcal{M}_S$ that can compromise the integrity of TEEs and in the presence of an attacker $\mathcal{M}_W$ that can compromise the confidentiality but not the integrity of TEEs.

**$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}$ and $\mathbb{M}_{\mathcal{I}\to\mathcal{U}}$: Messaging with Ideal TEE (Figures A.1 and A.2)**
After establishing a shared key with the Ideal TEE $\mathcal{I}$, the user can communicate with it by sending encrypted messages. For the Ideal TEE, the communication via shared keys does not differ from the standard notion of exchanging messages with a shared key. Figure A.1 shows the process of sending messages from $\mathcal{U}$ to $\mathcal{I}$ and A.2 shows the process of sending messages from $\mathcal{I}$ to $\mathcal{U}$. Both directions work equally and can be described as follows:

1. Combine message $m$ with sequence number $n$ and encrypt them using an authenticated encryption scheme and key $k$

2. Send encrypted message to communication partner

3. Receiver verifies the authenticated encryption and the sequence number

In the following, we assume for all protocols that the participating parties have performed a key exchange with the Ideal TEE and use such replay protected messages whenever sending messages to $\mathcal{I}$. To make this clear, we use $\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(m)$ to denote messages from $\mathcal{U}$ to $\mathcal{I}$, and $\mathbb{M}_{\mathcal{I}\to\mathcal{U}}(m)$ to denote messages from $\mathcal{I}$ to $\mathcal{U}$ respectively. However we do not specifically mention the specific function if it is clear in what direction the message is being sent.

**$\mathfrak{M}^A_{\mathcal{U}\to\mathcal{T}}$ and $\mathfrak{M}^A_{\mathcal{T}\to\mathcal{U}}$: Messaging with Combined TEE in the presence of $\mathcal{M}_W$ (Figures 5.3 and 5.4)**
In the case of $\mathcal{M}_W$ as an adversary, the situation of sending simple communication messages is different to the base case of an Ideal TEE. While it is still the case that only one of the keys that $\mathcal{U}$ maintains in the key set $K$ is uncompromised, $\mathcal{U}$ now also has the guarantee that each of the TEEs $\mathcal{U}$ communicates with maintain their integrity. This has direct consequences for the simplicity of some protocols as the user has not to account for misbehaving TEEs anymore. Specifically, $\mathcal{M}_W$ can learn the key $k_i$ established between $\mathcal{U}$ and $\mathcal{T}$ in the key establishment phase and use this to forge correctly encrypted messages from $\mathcal{U}$ to the TEE or vice-versa. Therefore, $\mathcal{U}$ now has to take steps to ensure that messages she sends to the TEEs are verified by the TEE in order to prevent $\mathcal{M}_W$ from impersonating $\mathcal{U}$. The TEE also needs to take additional steps to ensure that messages it sends to $\mathcal{U}$ can be verified by her in order to prevent $\mathcal{M}_W$ from impersonating $\mathcal{T}$. Both directions can be ensured via authenticated messages and are depicted in Figure 5.3 and 5.4.

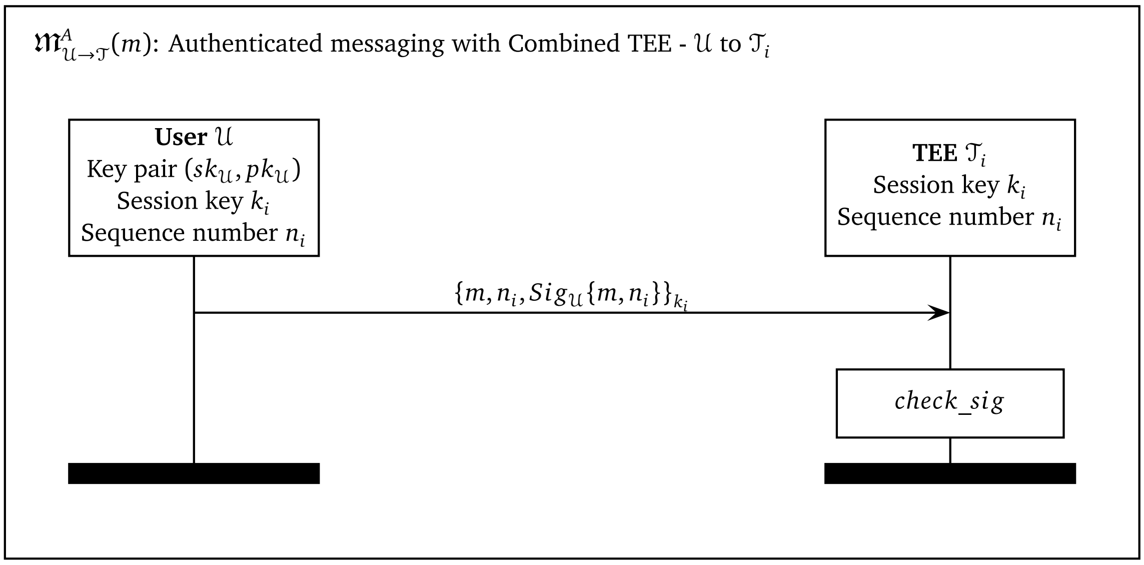

$\mathfrak{m}^A_{\mathcal{U}\to\mathcal{T}}(m)$: Authenticated messaging with Combined TEE - $\mathcal{U}$ to $\mathcal{T}_i$
**User** $\mathcal{U}$
Key pair $(sk_{\mathcal{U}}, pk_{\mathcal{U}})$
Session key $k_i$
Sequence number $n_i$
**TEE** $\mathcal{T}_i$
Session key $k_i$
Sequence number $n_i$
$\{m, n_i, Sig_{\mathcal{U}}\{m, n_i\}\}_{k_i}$
$check\_sig$

**Figure 5.3.:** Authenticated messaging between user $\mathcal{U}$ and TEE $\mathcal{T}_i$. In the following denoted as $\mathfrak{m}^A_{\mathcal{U}\to\mathcal{T}}(m)$. If $\mathcal{M}$ controls the session key $k_i$, the signature creatung using $sk_{\mathcal{U}}$ ensures that $\mathcal{M}$ can not forge any messages. The sequence number ensures freshness in the messages.

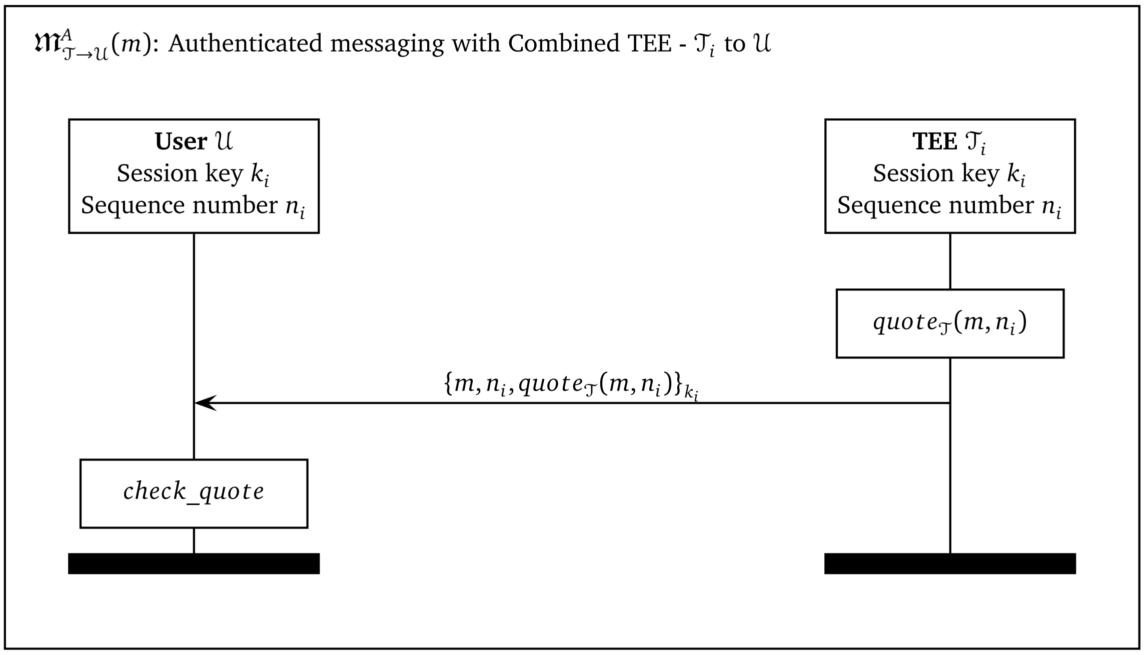

$\mathfrak{m}^A_{\mathcal{T}\to\mathcal{U}}(m)$: Authenticated messaging with Combined TEE - $\mathcal{T}_i$ to $\mathcal{U}$
**User** $\mathcal{U}$
Session key $k_i$
Sequence number $n_i$
**TEE** $\mathcal{T}_i$
Session key $k_i$
Sequence number $n_i$
$quote_{\mathcal{T}}(m, n_i)$
$\{m, n_i, quote_{\mathcal{T}}(m, n_i)\}_{k_i}$
$check\_quote$

**Figure 5.4.:** Authenticated messaging between TEE $\mathcal{T}_i$ and user $\mathcal{U}$. In the following denoted as $\mathfrak{m}^A_{\mathcal{T}\to\mathcal{U}}(m)$. If $\mathcal{M}_W$ controls the session key $k_i$, the quote guarantees that the adversary can not alter the message in transit before it reaches $\mathcal{U}$.

To send authenticated messages from $\mathcal{U}$ to $\mathcal{T}$, in the following denoted as $\mathfrak{m}^A_{\mathcal{U}\to\mathcal{T}}$, $\mathcal{U}$ can simply use the same signing key that she already used during key exchange to authenticate herself during establishing of a session key. As depicted in Figure 5.3, in addition to the message and the sequence number, $\mathcal{U}$ also includes a signature in the encrypted message. This signature is computed over the hash of the message and the sequence number and can be verified using the public key of $\mathcal{U}$. After receiving such a message, $\mathcal{T}$ can verify the content of the decrypted message to originate from $\mathcal{U}$ instead of $\mathcal{M}_W$. As the integrity of $\mathcal{T}$ is considered to not be impaired, $\mathcal{T}$ would cease any functionality once it detects an incorrect signature.

For authenticated messages from the TEE to the user, in the following denoted as $\mathfrak{m}^A_{\mathcal{T}\to\mathcal{U}}$, the TEE can attach a quote to all encrypted messages similarly to the message sent from the user described previously. As $\mathcal{M}_W$ is not assumed to have any control over architectural capabilities such as quotes, it is possible for $\mathcal{T}$ to create a quote over its own content without the adversary being able to tamper with such a quote. Figure 5.4 depicts the process of such a message. In addition to the message and sequence number, the TEE also encrypts the quote over the same parameters. After receiving such an authenticated message from $\mathcal{T}$, $\mathcal{U}$ checks the correctness and the origin of the quote and only proceeds if the message passes all checks. Since the user is expected to be interested in securing her own communication, we can also expect that she would abort a protocol if she detects that $\mathcal{M}_W$ manipulated a message.
For the sake of simplicity, we also use $\mathfrak{m}^A_{\mathcal{T}\to\mathcal{U}}$ in the following sections to show that $\mathcal{T}$ is sending a quote over the content of the message to $\mathcal{U}$. This helps to maintain a simple notation for generating and verifying quotes and as such can also be used if $\mathcal{M}_S$ is used as an adversary model.

**$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}$ and $\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}$: Messaging with Combined TEE in the presence of $\mathcal{M}_S$**

If an adversary $\mathcal{M}_S$ has compromised the integrity of all but one of the involved TEEs, $\mathcal{U}$ only has the guarantee that one of the established keys in the key set $K$ is secure while all other keys have to be assumed to be compromised. However, as $\mathcal{U}$ has no knowledge about which key is secure, she has to assume that any key could be compromised during normal communication, and resolve this trust issue through other means such as using secret sharing among elements of the Combined TEE. This means, that $\mathcal{U}$ can not perform any steps for simple messages to increase her security and sends messages to each individual $\mathcal{T}$ as if she was communicating with the Ideal TEE $\mathcal{I}$. Thus, the protocol for sending messages in the case of $\mathcal{M}_S$ is the same as the base protocols that are depicted for the Ideal TEE in Figures A.1 and A.2. We denote sending simple encrypted messages in the Combined TEE from $\mathcal{U}$ to $\mathcal{T}$ with $\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(m)$ and in the other direction with $\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(m)$. While these functions work equally to the functions $\mathbb{M}_{\mathcal{U}\to\mathcal{I}}$ and $\mathbb{M}_{\mathcal{I}\to\mathcal{U}}$ defined for the Ideal TEE, it is interesting to distinguish the functions if they are operated in the context of the Combined TEE as there are security implications from these functions within the adversary model of the Combined TEE. We point out such differences in the discussion of each specific protocol.
It is important to note that the authenticated messages $\mathfrak{m}^A_{\mathcal{U}\to\mathcal{T}}$ and $\mathfrak{m}^A_{\mathcal{T}\to\mathcal{U}}$ could also be used in the presence of $\mathcal{M}_S$. However, in doing this $\mathcal{U}$ would not gain any benefits as she could not trust any response sent by $\mathcal{T}$ and could not use any information retained in such response implicitly. In contrast, the surrounding protocol would need to ensure that any responses sent by the TEEs to $\mathcal{T}$ would result in the desired security guarantees, e.g. by $\mathcal{U}$ combining the responses and treating them equally. As $\mathcal{T}_S$ is considered to be uncompromised, its response would be part of the combined response and would provide the desired security guarantees while an authenticated message to $\mathcal{T}_M$ does not impact the security of the result.

## 5.2 One party protocols

Based on a secure and authenticated channel with the TEE, the user can start to query the TEE to perform specific tasks. A core capability of any TEE-based system is that it can produce authenticated (i.e. attested) outputs that can be verified by remote parties. As such we describe authenticated random number generation, authenticated key generation and ciphertext decryption, and signing with a TEE. While some of these protocols account for a third party that verifies the results of a function, there is no online involvement required by these third parties. Instead, a single user can run these protocols without any involvement of a third party and can then hand the outputs of the protocols to third parties for verification.

### 5.2.1 Authenticated random number generation

The first one-party protocol we discuss is the process of generating an authenticated random number with a TEE. Such authenticated number can be verified by a third party to originate from inside a TEE. Authenticated random numbers can be the basis for other protocols and can be an important starting point for cryptographic operations. As the TEE guarantees its security properties to hold, it is reasonable to assume that a random number generated inside the TEE is more suitable for cryptographic operations than a random number generated by the user on her own device. While the user already requires some random number generator to even establish communication with the TEE, there is still a benefit in using the better random number generator of the TEE for future protocols. In using an authenticated random number, the remote party can be convinced about the source of the randomness if it does not trust the user to properly generate randomness. By leveraging the TEE attestation property to create a quote over the random number, the user could convince the remote party that she did not influence the generation of the random string and that it can be trusted if one trusts the TEE that generated the number. We expect the following security guarantees from a secure and authenticated random number:

1. The result must be a randomly generated string of length $n$ where $n$ is specified by the user. The randomness must not be influenced by any other party than the TEE itself.
2. The adversary must be unaware of the final random string.
3. A third party must later be able to verify that the random string was generated by a TEE.

**$\mathbb{R}(n)$: Authenticated random number generation with Ideal TEE (Figure A.3)**

To generate a random number with $\mathcal{I}$, $\mathcal{U}$ can simply request that the Ideal TEE generate and return it to $\mathcal{U}$. This is secure as $\mathcal{I}$ can be used as a TTP here. Figure A.3 shows such a protocol where $\mathcal{U}$ sends a *request_random* message to $\mathcal{I}$ that specifies the number of bits to be generated. $\mathcal{I}$ then generates a random bit string of the requested length and returns it to $\mathcal{U}$. As $\mathcal{U}$ needs to present the output to a third party, $\mathcal{I}$ also creates a quote over the specified length and the output. $\mathcal{U}$ can send this quote together with the random number to third parties to convince them of the origin of this generated random number.

**$\mathfrak{R}(n)$: Authenticated random number generation in the presence of $\mathcal{M}_W$**

For a weak adversary $\mathcal{M}_W$, we can use a protocol similar to Mavroudis et al. [51] with the addition that our protocol also generates quotes that result in an authenticated random number. Figure 5.5 shows protocol $\mathfrak{R}(n)$ where $\mathcal{U}$ requests two random numbers of length $n$ from $\mathcal{T}_1$ and $\mathcal{T}_2$. Both TEEs then generate a random number of requested length and return them as authenticated messages that also contain a quote over the corresponding random number and the length. After combining the random numbers locally with an XOR operation, $\mathcal{U}$ can present the quotes together with $r_1$ and $r_2$ to remote parties. Any remote party that receives these random numbers first needs to check the validity of both quotes (and that they originate from two different TEEs) and then recompute the combined random number from its parts that are verified through the quotes. As $\mathcal{M}_W$ has no control over how the TEE generates the random number, he can not influence it. While it holds true that $\mathcal{M}_W$ knows one part of the resulting random number, he does not know the other part and as such can not recompute even parts of the resulting XOR operation. Lastly, even if $\mathcal{U}$ colludes with $\mathcal{M}_W$, they can not influence the randomness of the result. $\mathcal{U}$ can mix the received numbers from multiple runs of the protocol, but as the integrity of the TEEs is not compromised, she can never influence their randomness and as such any resulting random number will still be influenced by at least one uncompromised TEE.
$\mathfrak{R}$ is secure because in order to obtain a specific desired random number, $\mathcal{M}_W$ would need to query $\mathcal{T}_M$ until it returns the desired result. This is computationally infeasible for sufficiently large values of $n$.

**$\mathfrak{R}_{\mathcal{M}_S}(n)$: Authenticated random number generation in the presence of $\mathcal{M}_S$**

For the strong adversary $\mathcal{M}_S$, the Protocol $\mathfrak{R}$ is not sufficient. While it still holds true that $\mathcal{M}_S$ can not learn the combined random number $r_c$, he can collude with $\mathcal{U}$ in order to modify the combined generated random number. As in $\mathfrak{R}$ the two random numbers are not dependent on each other, $\mathcal{U}$ can query the uncompromised TEE $\mathcal{T}_S$ for a random number first and then calculate the second random number based on the desired output. $\mathcal{U}$ can then request $\mathcal{M}_S$ to create a quote

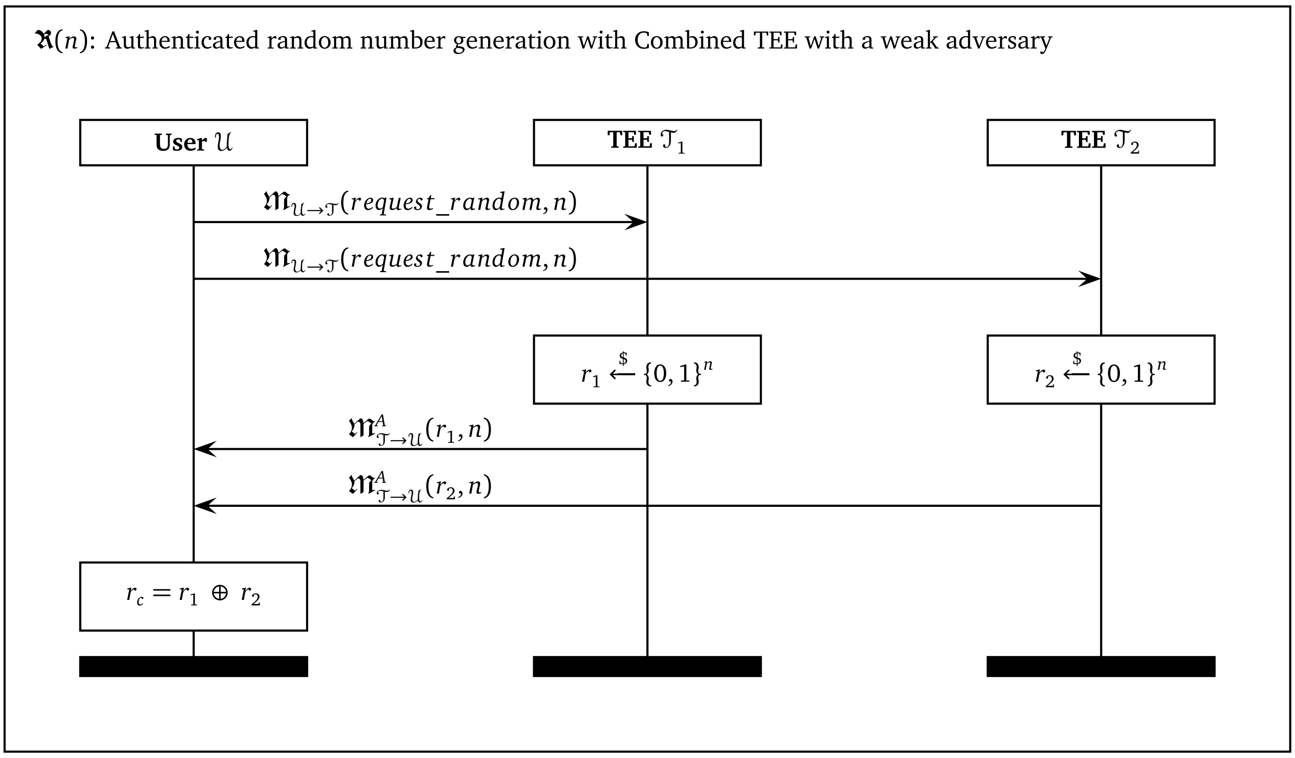

$\mathfrak{R}(n)$: Authenticated random number generation with Combined TEE with a weak adversary
User $\mathcal{U}$
TEE $\mathcal{T}_1$
TEE $\mathcal{T}_2$
$\mathfrak{M}_{\mathcal{U}\rightarrow\mathcal{T}}(request\_random, n)$
$\mathfrak{M}_{\mathcal{U}\rightarrow\mathcal{T}}(request\_random, n)$
$r_1 \xleftarrow{\$} \{0,1\}^n$
$r_2 \xleftarrow{\$} \{0,1\}^n$
$\mathfrak{M}^{A}_{\mathcal{T}\rightarrow\mathcal{U}}(r_1, n)$
$\mathfrak{M}^{A}_{\mathcal{T}\rightarrow\mathcal{U}}(r_2, n)$
$r_c = r_1 \oplus r_2$

**Figure 5.5.:** Authenticated random number generation between user $\mathcal{U}$ and the Combined TEE $\mathcal{C}$ consisting of the actual TEEs $\mathcal{T}_1$ and $\mathcal{T}_2$. $\mathcal{U}$ obtains a random number of length $n$ while neither $\mathcal{T}_1$ nor $\mathcal{T}_2$ have any knowledge of the combined output as it the result of an XOR operation over both outputs. $\mathcal{U}$ needs to abort the protocol if any of the $r_i$ does not have length $n$.

over this generated value with the identity of the TEE $\mathcal{T}_M$ he has compromised. The result would be a combined number that was fully chosen by $\mathcal{U}$ and does not contain any randomness.

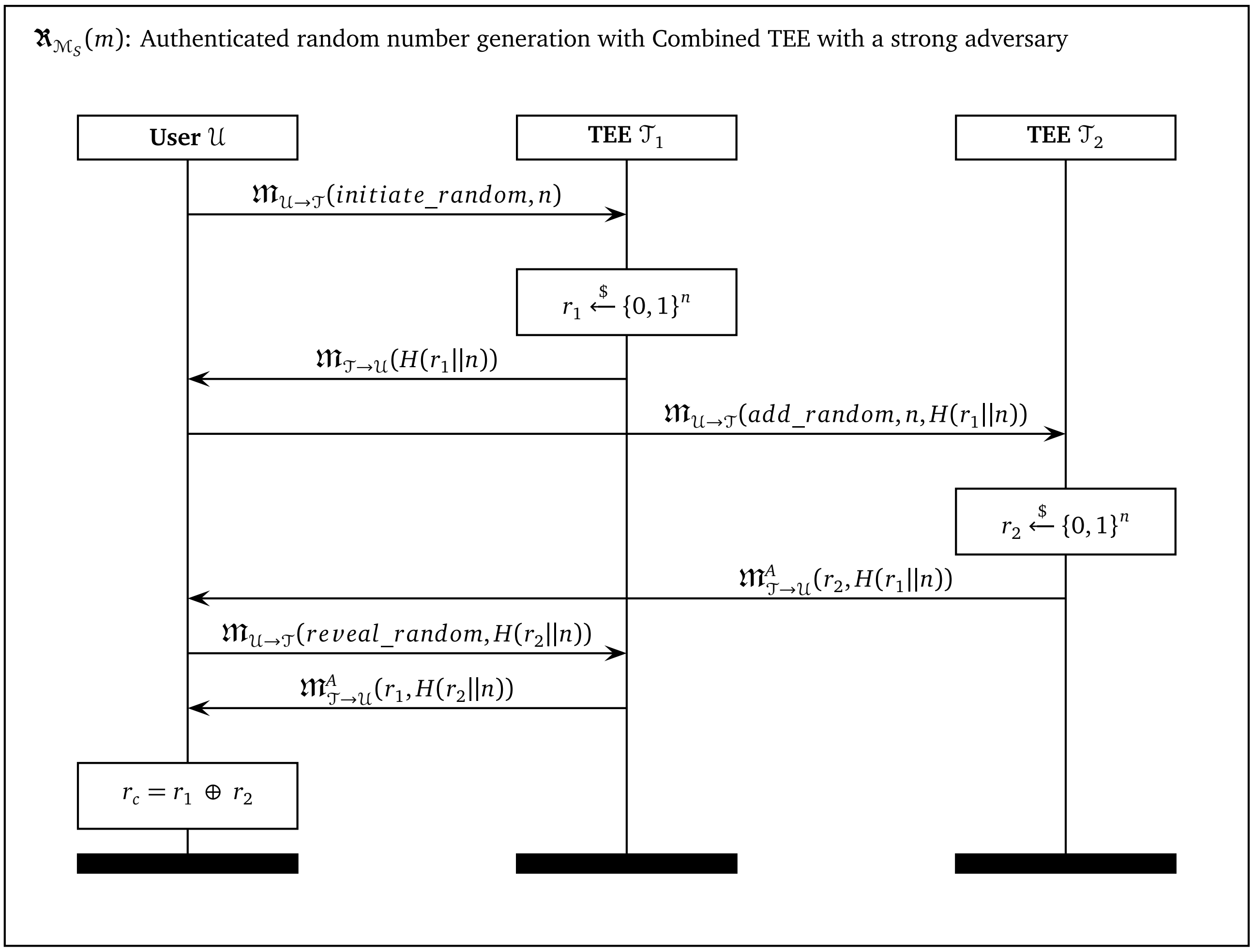
Thus, in the presence of $\mathcal{M}_S$, we choose a similar approach as Mavroudis et al. [51] chose for their key generation protocol. The protocol roughly consists of three phases: The commitment phase, a round where the second TEE submits its value based on the commitment, and lastly the reveal phase where the committed number is revealed based on the already submitted value of the second TEE. As Figure 5.6 shows, $\mathfrak{R}_{\mathcal{M}_S}$ starts similar to $\mathfrak{R}$ when $\mathcal{U}$ sends an initial message to $\mathcal{T}_1$ with the requested amount of bits to start the protocol. Next, $\mathcal{T}_1$ generates a random number but in contrast to the protocol above does not directly send it back to $\mathcal{U}$. Instead, $\mathcal{T}_1$ only returns a hash over the random number concatenated with the amount of bits to the user. This creates a commitment from $\mathcal{T}_1$ that prevents it from changing the random number after it receives the random value of $\mathcal{T}_2$. In the second phase of the protocol, $\mathcal{U}$ sends the commitment to $\mathcal{T}_2$ together with the desired amount of bits in a *add_random* message to request $\mathcal{T}_2$ to add a random number based on the current commitment. Upon receiving such a request, $\mathcal{T}_2$ generates a random number $r_2$ and returns an authenticated message of it and the received commitment. This authenticated message contains a quote over the content and can as such be verified by third parties. Once $\mathcal{U}$ receives the random number that is bound to the first commitment, $\mathcal{U}$ creates a hash of $r_2$ and $n$, similar to the commitment that is returned by $\mathcal{T}_1$ in the first phase, and sends it to $\mathcal{T}_1$ to request that it reveals its random number based on $r_2$. This third and last phase is then completed by $\mathcal{T}_1$ sending and authenticated message over $r_1$ and the hash of $r_2$. It is crucial that $\mathcal{U}$ only send the hash of $r_2$ to $\mathcal{T}_1$ to prevent an attacker that has compromised $\mathcal{T}_1$ from obtaining both parts of $r_c$. A third party can authenticate the combined random number $r_c$ by checking both quotes and verifying that they are generated by two different TEEs. Afterwards, the third party has to check the contents of both quotes, i.e. verify that the hash contained in each message corresponds to the random number contained in the other message and also ensure that both sizes of $n$ match to the given number and the requirement specified by $\mathcal{U}$.

With this protocol, $\mathcal{M}_S$ is not able to choose $r_c$ anymore if he colludes with $\mathcal{U}$. Specifically, if one TEE stays uncompromised, $\mathcal{M}_S$ has two options: He compromises $\mathcal{T}_1$ or he compromises $\mathcal{T}_2$.

1. $\mathcal{T}_1$ is $\mathcal{T}_M$: If $\mathcal{T}_1$ is compromised, $\mathcal{M}_S$ has to commit to a random number before $\mathcal{T}_S$ generates $r_2$. As $\mathcal{T}_2$ then binds its value to the commitment, $\mathcal{M}_S$ can not change his choice anymore without finding a hash collision which is computationally infeasible. As such, $\mathcal{M}_S$ can not influence the random number at all in this scenario as it completely depends on the randomness generated by $\mathcal{T}_2$.

2. $\mathcal{T}_2$ is $\mathcal{T}_M$: If $\mathcal{T}_2$ is compromised on the other hand, $\mathcal{T}_1$ already commited to $r_1$ before it has to generate its own number $r_2$. However, $\mathcal{T}_2$ only learns the hash of $r_1$ which is, again, infeasible to reverse. As such, $\mathcal{T}_2$ can commit to any number without being able to guess or influence the final result of $r_c$.

We conclude that it is not possible for $\mathcal{M}_S$ to either influence or deduce the final value of $r_c$ if does not have control over all participating TEEs, even if he colludes with $\mathcal{U}$.



$\aleph_{\mathcal{M}_S}(m)$: Authenticated random number generation with Combined TEE with a strong adversary
User $\mathcal{U}$
TEE $\mathcal{T}_1$
TEE $\mathcal{T}_2$
$\mathfrak{M}_{\mathcal{U}\to\mathcal{T}}(initiate\_random, n)$
$r_1 \xleftarrow{\$} \{0,1\}^n$
$\mathfrak{M}_{\mathcal{T}\to\mathcal{U}}(H(r_1||n))$
$\mathfrak{M}_{\mathcal{U}\to\mathcal{T}}(add\_random, n, H(r_1||n))$
$r_2 \xleftarrow{\$} \{0,1\}^n$
$\mathfrak{M}^A_{\mathcal{T}\to\mathcal{U}}(r_2, H(r_1||n))$
$\mathfrak{M}_{\mathcal{U}\to\mathcal{T}}(reveal\_random, H(r_2||n))$
$\mathfrak{M}^A_{\mathcal{T}\to\mathcal{U}}(r_1, H(r_2||n))$
$r_c = r_1 \oplus r_2$

**Figure 5.6.:** Authenticated random number generation in the presence of the strong adversary $\mathcal{M}_S$. User $\mathcal{U}$ uses the Combined TEE $\mathcal{C}$ consisting of the actual TEEs $\mathcal{T}_1$ and $\mathcal{T}_2$. $\mathcal{U}$ obtains a random number of fixed length $n$ while neither $\mathcal{T}_1$ nor $\mathcal{T}_2$ have any knowledge of the combined result. After the successful execution of the protocol, $\mathcal{U}$ can present all quotes to prove that $r_c$ is the result of a proper execution of the protocol. $\mathcal{U}$ and any remote verifier must omit iterations as invalid where any random number does not have length $n$ or where both responses come from the same TEE.

**Limitations of authenticated random numbers**

It is difficult to reason about remotely generated randomness in general. If, for example, the random number is supposed to be a substitute for a coin flip where a decision would be made upon the value of the generated number, the remote party has no easy way to verify that the presented output is in fact the result of the first iteration of the coin flip. It is certainly possible for $\mathcal{U}$ to rerun the protocol often enough times to obtain a random number that she is satisfied with and that she can then present to a third party. If $\mathcal{U}$ would have a benefit in choosing a specific random number, this would undermine the security guarantee that the underlying protocol would try to achieve with such an authenticated random number. This does not take away the general usefulness of the mentioned protocol however. There might exist scenarios where a remote third party is interested in being convinced that a random number was actually generated inside an Ideal TEE without outside influences. Such a random number could then be used as a common seed to construct further pseudo random numbers. It is certainly also possible to create a random number generation based on a one way function that will always produce the same output based on a challenge by a third party. That way, the user could not try to cheat by rerunning the protocol until she is satisfied with the result. However, these constructions strongly depend on the surrounding scenario and on the security guarantees that the surrounding protocol aims to achieve. We consider any extensions on $\mathbb{R}$ as out of scope, especially since it is possible to use this protocol as a base to design solutions to specific tasks around it.

**Notes on non-authenticated random number generation**

If $\mathcal{U}$ does not need an authenticated random number but simply wants to query randomness from her TEE, she can use protocol $\mathfrak{R}$ regardless of her adversary. For simplicity, she can then also use a simplification of the protocol where the last messages from the TEEs to $\mathcal{U}$ are not authenticated messages as she will not need the quote. For a non-authenticated random number, this protocol is also secure in the presence of $\mathcal{M}_S$ as $\mathcal{U}$ still combines the input of both TEEs to a final output. As long as $\mathcal{M}_S$ did not compromise all participating TEEs, he is not able to control or know all parts of the final output and will be unaware of its content.

### 5.2.2 ElGamal cryptographic operations

The second set of one-party protocols allows the user $\mathcal{U}$ to utilize ElGamal cryptographic operations with a TEE. As such, $\mathcal{U}$ can request the TEE to generate a public key that can be verified by third parties to originate from inside a TEE. Any encrypted message must then be forwarded from $\mathcal{U}$ to the TEE in order to receive the plaintext of the message, as $\mathcal{U}$ does not have access to the private key. This gives a certain guarantee on the key secrecy to third parties as they can verify that the private key has never left the realm of the TEE and has especially not been leaked to any other party than the TEE itself. However, it is certainly possible for $\mathcal{U}$ to pass on the plaintext of messages to other users. Since this is an orthogonal trust issue and not in scope of an encryption scheme, we consider this out of scope. The resulting security guarantees provided by an authenticated ElGamal key generated by the TEE are similar to the same guarantees defined for the random number above:

1. The private ElGamal key is solely controlled by the TEE and is neither known nor has been influenced by the adversary.

2. The public key can be verified by third parties and linked back to one specific TEE instance.

3. During decryption of a ciphertext, neither the user nor the adversary learn the private key and the adversary does not learn the plaintext of the decrypted message.

When using the ElGamal encryption, we use an elliptic curve version with a tuple of domain parameters $\lambda = (p, q, n, E, G)$ as described in Section 2.1 and based on the construction of Koblitz [44].

**$\mathbb{G}$ and $\mathbb{D}$: ElGamal cryptographic operations with Ideal TEE (Figures A.4 and A.5)**

ElGamal cryptographic operations with the Ideal TEE are essentially equal to the base version of the elliptic curve version of the ElGamal cryptographic system described in Section 2.1.2. Key generation ($\mathbb{G}$) is performed by $\mathcal{U}$ requesting a key generation from $\mathcal{I}$. Upon this request, $\mathcal{I}$ generates a random private key $X$, calculates the corresponding public key $Y = X \cdot G$, and returns that public key to $\mathcal{U}$ together with a quote that proofs the origin of the key. That way, third parties can link a public key back to the Ideal TEE and verify that the key was generated inside $\mathcal{I}$. This is similar to the $\mathbb{R}$ protocol above.

For any ciphertext $C = (C_1, C_2)$ as discussed in Section 2.1.2, $\mathcal{U}$ can use the protocol $\mathbb{D}$ for decryption. First, $\mathcal{U}$ sends $C_1$ to $\mathcal{I}$ to get the decryption share $D$. After $\mathcal{I}$ returned $D = X \cdot C_1$, $\mathcal{U}$ can retrieve the message $m = C_2 - D$.

**$\mathfrak{G}$: Authenticated ElGamal key generation in the presence of $\mathcal{M}_W$ (Figure 5.7)**

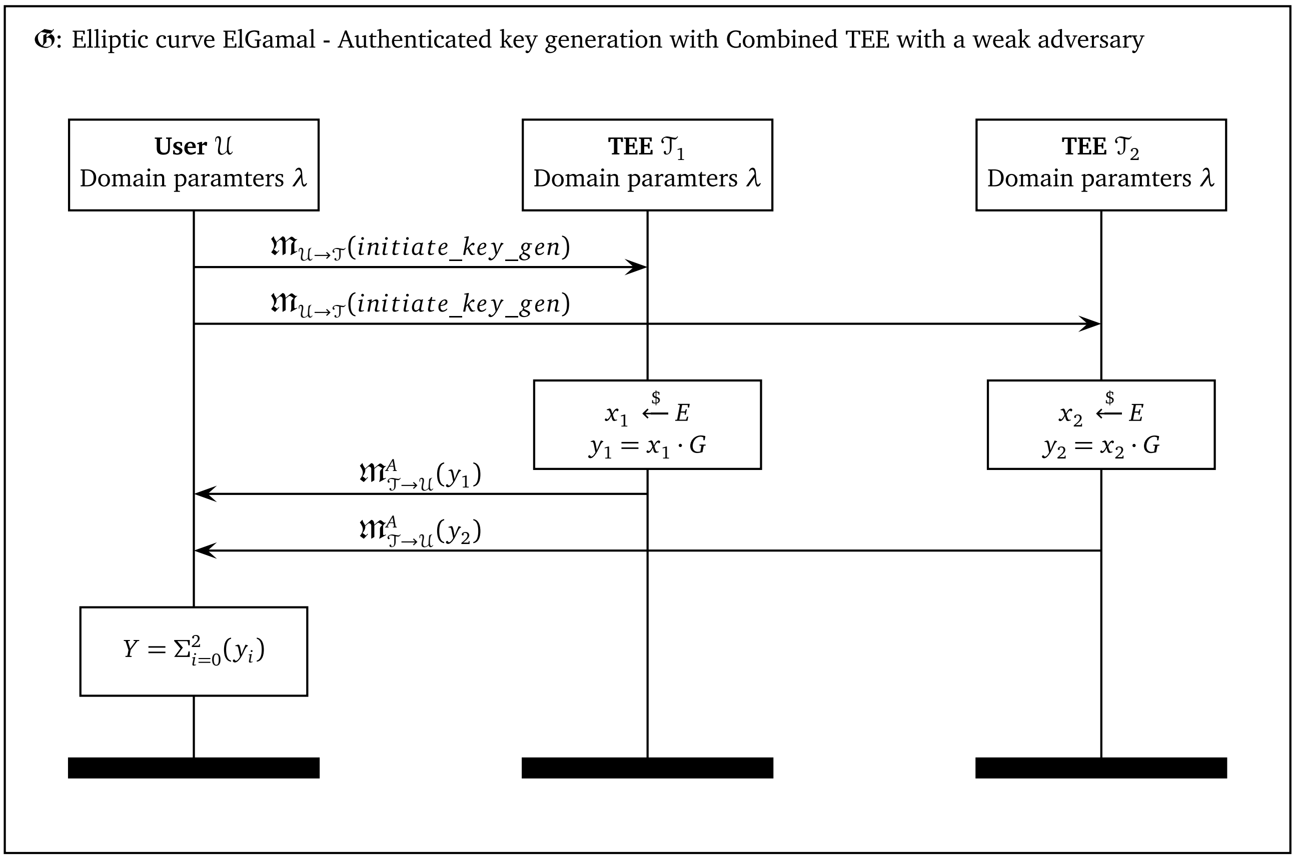
In conrast to the Ideal TEE, the Combined TEE needs to spread out the trust of the private key held in the system and ensure that no element of the Combined TEE has access to the full private key at any point in time. This means that it is not possible to employ the same protocol as is used by the Ideal TEE against an adversary $\mathcal{M}_W$ that can compromise the TEE confidentiality. In the presence of $\mathcal{M}_W$, we instead employ a threshold ElGamal scheme similar to the construction of Mavroudis et al. [51] or Brandt [11]. This ensures that $\mathcal{M}_W$ cannot deduce the private key or the plaintext of decrypted messages if at least one TEE is uncompromised. Figure 5.7 shows the key generation process for the threshold encryption elliptic curve ElGamal key. Similar to protocol $\mathfrak{R}$, the user $\mathcal{U}$ initiates the protocol by sending two *initiate_key_gen* messages to $\mathcal{T}_1$ and $\mathcal{T}_2$. Those both randomly generate their private key as a point on the elliptic curve $E$ as defined in the domain parameters. The public key $y_i$ corresponding to the private key $x_i$ is then calculated by each TEE as $y_i = x_i \cdot G$ and returned to $\mathcal{U}$ in an authenticated message that contains a quote over this public key share. Due to the homomorphic properties of ElGamal, the final public key $Y$ can be calculated with $Y = \Sigma_{i=0}^{2}(y_i)$. Remote parties can verify a public key by checking the quotes and recalculating $Y$ locally. Again, both the user and any remote party need to verify that the quotes originate from two different TEEs as otherwise $\mathcal{M}_W$ could trivially perform this protocol with only one TEE $\mathcal{T}_M$.

In contrast to Mavroudis et al., no commitment phase is needed in this protocol as the TEEs do not change their key share once they received other shares and $\mathcal{M}_W$ can not arbitrarily choose a point on the curve that would result in a public key that he has control over. In our protocol it is also not necessary to let the two TEEs calculate the aggregated key on their own for the same reason, they will always honestly report their share to $\mathcal{U}$. This considerably lowers the complexity of the protocol as there is only one round of communications needed.

The security of $\mathfrak{G}$ stems from $\mathcal{M}_W$ not being able to actively choose the private key that $\mathcal{T}_M$ generates. Instead, $\mathcal{M}_W$ can only query $\mathcal{T}_M$ as often as he wants to and until he decides to pick one of the generated public key shares. If $\mathcal{U}$ colludes with $\mathcal{M}_W$, they are together able to simply pick a suitable response from $\mathcal{T}_S$ and match it with a desired response from $\mathcal{T}_M$. However, doing this until they end up with a key that they control implies querying $\mathcal{T}_M$ arbitrarily often. As such the security of $\mathfrak{G}$ depends on the complexity of randomly generating a specific desired key which is computationally infeasible.

**$\mathfrak{G}_{\mathcal{M}_S}$: Authenticated ElGamal key generation in the presence of $\mathcal{M}_S$ (Figure 5.8)**

The restriction that prevents a weak attacker from choosing a combined key during the key generation is based on the attacker being limited to querying the TEE. A strong attacker $\mathcal{M}_S$ however is not limited to querying the TEE but can instead simply choose what the TEE does or impersonate it directly. Due to the homomorphic properties of ElGamal that we use to combine the public key from multiple shares, the adversary can calculate the difference between the public key share of $\mathcal{T}_S$ and a public key that is controlled by $\mathcal{M}_S$ and $\mathcal{U}$ and use this difference as a public key share as public key of $\mathcal{T}_M$. The result is a public key that consists of two separate shares but of which $\mathcal{U}$ also knows the corresponding private key and does not need to make use of the TEEs anymore. This breaks the mentioned security guarantees as the user now has no incentive anymore to use the TEEs and the remote party has no guarantee anymore that $\mathcal{U}$ will be the only party to ever have access to the key.



$\mathfrak{G}$: Elliptic curve ElGamal - Authenticated key generation with Combined TEE with a weak adversary
**User** $\mathcal{U}$
Domain paramters $\lambda$
**TEE** $\mathcal{T}_1$
Domain paramters $\lambda$
**TEE** $\mathcal{T}_2$
Domain paramters $\lambda$
$\mathfrak{M}_{\mathcal{U}\to\mathcal{T}}(initiate\_key\_gen)$
$\mathfrak{M}_{\mathcal{U}\to\mathcal{T}}(initiate\_key\_gen)$
$x_1 \overset{\$}{\leftarrow} E$
$y_1 = x_1 \cdot G$
$x_2 \overset{\$}{\leftarrow} E$
$y_2 = x_2 \cdot G$
$\mathfrak{M}^A_{\mathcal{T}\to\mathcal{U}}(y_1)$
$\mathfrak{M}^A_{\mathcal{T}\to\mathcal{U}}(y_2)$
$Y = \Sigma^2_{i=0}(y_i)$

**Figure 5.7.:** Authenticated key generation between user $\mathcal{U}$ and the Combined TEE in the presence of a weak adversary $\mathcal{M}_W$. The TEEs each generate a private key $x_i$ and its corresponding public key $y_i = x_i \cdot G$ and send $y_i$ back to $\mathcal{U}$ as an authenticated message. $\mathcal{U}$ then combines each public key share and obtains the aggregated public key $Y$. Third parties can verify the quotes and use the $y_i$ shares to recalculate $Y$ themselves. Again, both $\mathcal{U}$ and remote parties need to verify that the quotes originate from different TEEs.

Against the stronger adversary $\mathcal{M}_S$, we additionally employ a commitment round similar to protocol $\mathfrak{K}_{\mathcal{M}_S}$. As such, after sending a *initiate_keygen* message to $\mathcal{T}_1$, $\mathcal{U}$ receives back a commitment to the public key share $y_1$ and the domain parameters $\lambda$. $\mathcal{U}$ then sends this commitment to $\mathcal{T}_2$ with the request *add_key* that requests that $\mathcal{T}_2$ adds its own key to this commitment. After receiving $y_2$ bound by a quote to the first commitment, $\mathcal{U}$ requests $\mathcal{T}_1$ with a *reveal_key* message to reveal its public key based on the commitment of $\mathcal{T}_2$. To complete the protocol, $\mathcal{U}$ combines the two key shares into the combined public key $Y = \Sigma^2_{i=0}(y_i)$. Any third party that verifies the public key again needs to check each quote, verify that they originate from two different TEEs, and recalculate the public key based on its shares. It is also important to note that both the user $\mathcal{U}$ and any remote party need to abort the protocol whenever they encounter a public key share $y_i$ that is not on the elliptic curve $E$.

The security argument is the same as for protocol $\mathfrak{K}_{\mathcal{M}_S}$ as $\mathcal{M}_S$ can collude with $\mathcal{U}$ to choose a private key of $\mathcal{T}_M$ as he wishes, but due to the chain of commitments he is not able to change the resulting key without reversing the hash of the commitment. Thus, the adversary can only resolve to rerunning the protocol until he is satisfied with the result which requires an arbitrary amount of protocol runs while each of the runs contain at least one correctly and randomly generated private key by $\mathcal{T}_S$.

### $\mathfrak{D}$: ElGamal decryption with Combined TEE (Figure 5.9)

In contrast to the key generation protocols that require modifications based on the adversary model, ElGamal decryption with the Combined TEE works as in the threshold ElGamal scheme also discussed by Mavroudis et al. [51] or Brandt [11]. To decrypt a ciphertext $C$ that is a tuple $(C_1, C_2)$, $\mathcal{U}$ sends $C_1$ to each TEE with a *request_decryption* message which return

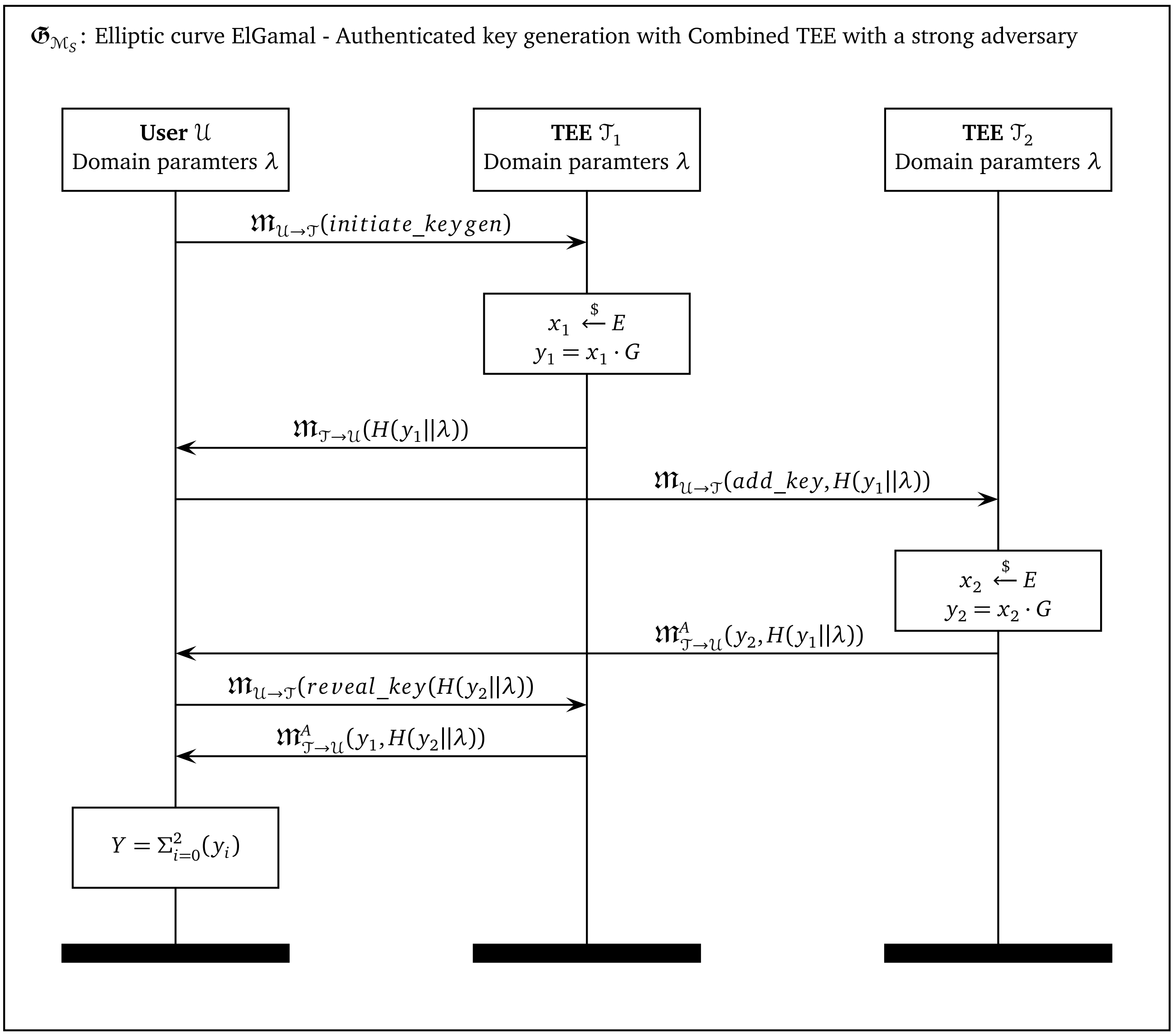

$\mathfrak{G}_{\mathcal{M}_S}$: Elliptic curve ElGamal - Authenticated key generation with Combined TEE with a strong adversary
User $\mathcal{U}$
Domain paramters $\lambda$
TEE $\mathcal{T}_1$
Domain paramters $\lambda$
TEE $\mathcal{T}_2$
Domain paramters $\lambda$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(initiate\_keygen)$
$x_1 \overset{\$}{\leftarrow} E$
$y_1 = x_1 \cdot G$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(H(y_1||\lambda))$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(add\_key, H(y_1||\lambda))$
$x_2 \overset{\$}{\leftarrow} E$
$y_2 = x_2 \cdot G$
$\mathfrak{m}^A_{\mathcal{T}\to\mathcal{U}}(y_2, H(y_1||\lambda))$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(reveal\_key(H(y_2||\lambda))$
$\mathfrak{m}^A_{\mathcal{T}\to\mathcal{U}}(y_1, H(y_2||\lambda))$
$Y = \Sigma_{i=0}^{2}(y_i)$

**Figure 5.8.:** Authenticated key generation between user $\mathcal{U}$ and the Combined TEE in the presence of a strong adversary $\mathcal{M}_S$. The TEEs each generate a private key $x_i$ and its corresponding public key $y_i = x_i \cdot G$ and bind the public key to the commitment of the other TEE. $\mathcal{U}$ then combines each public key share and obtains the aggregated public key $Y$. In contrast to the normal key generation, the authenticated key generation requires a commitment round so that a verifier can attest that all participating TEEs are correctly participating in the resulting key and no TEE can misbehave during the key generation process.

their decryption share $d_i = -x_i \cdot C_1$. $\mathcal{U}$ can then combine the decryption shares and add them to $C_2$ to retrieve the original plaintext message $m$.

Since the TEEs only return their decryption share, any adversary $\mathcal{M}$ does not learn the plaintext of the messages if he did not compromise all participating TEEs. It is important to note that $\mathcal{M}_S$ can endanger message integrity by manipulating the decryption share of $\mathcal{T}_M$ which would change the result of the resulting plaintext message. However, this can be resolved with orthogonal mechanisms like MAC-then-encrypt and is considered to be beyond the scope of this protocol.

### Notes on non-authenticated ElGamal operations

If a $\mathcal{U}$ does not need an authenticated version of ElGamal but instead only wishes to diversify her keys onto multiple TEEs, she does not need to rely on $\mathfrak{G}_{\mathcal{M}_S}$ even for $\mathcal{M}_S$. This is because she does not need to convince a third party that she did not collude with $\mathcal{M}_S$ and only requires the key to have sufficient strength and randomness. Since $\mathcal{M}_S$ can not influence

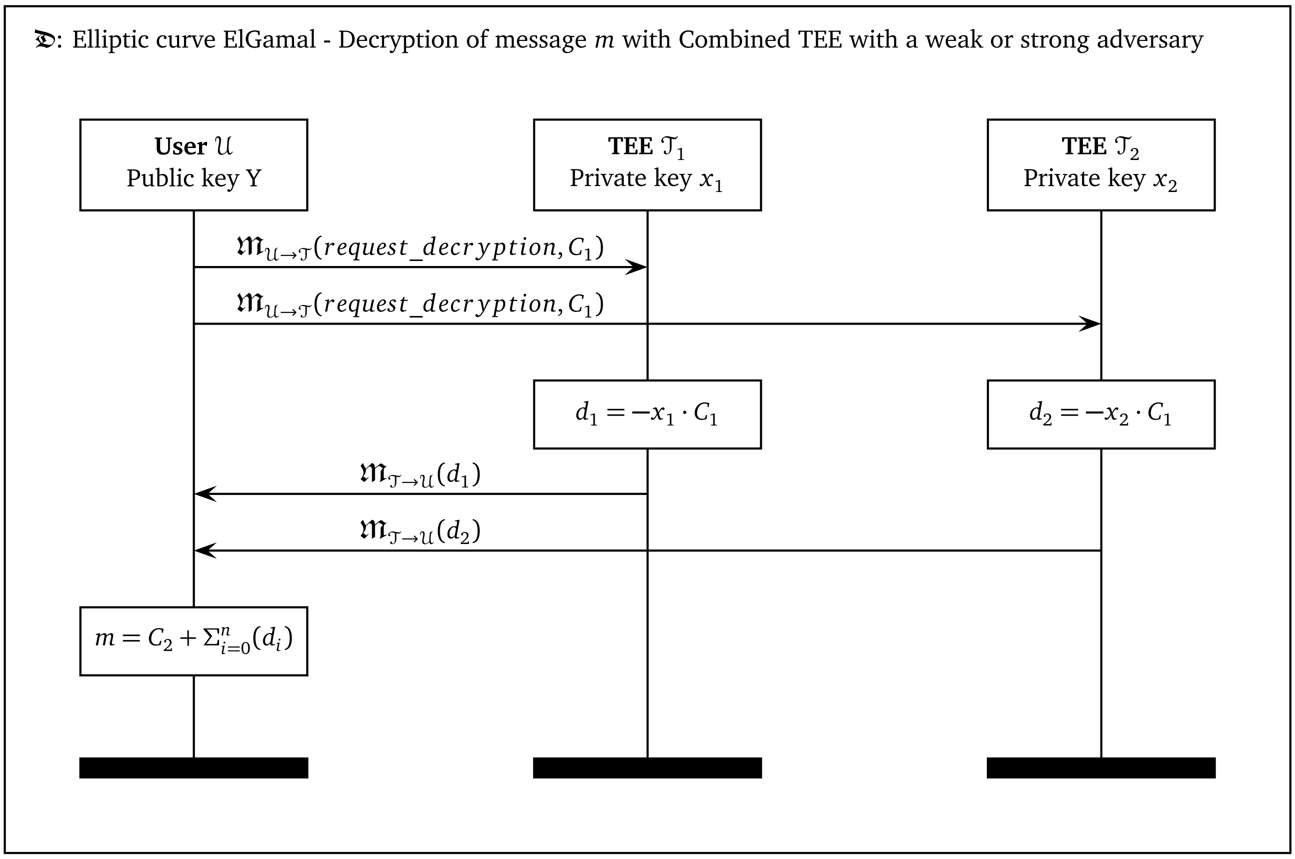

$\mathfrak{D}$: Elliptic curve ElGamal - Decryption of message $m$ with Combined TEE with a weak or strong adversary
User $\mathcal{U}$
Public key Y
TEE $\mathcal{T}_1$
Private key $x_1$
TEE $\mathcal{T}_2$
Private key $x_2$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(request\_decryption, C_1)$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(request\_decryption, C_1)$
$d_1 = -x_1 \cdot C_1$
$d_2 = -x_2 \cdot C_1$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(d_1)$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(d_2)$
$m = C_2 + \Sigma_{i=0}^{n}(d_i)$

**Figure 5.9.:** Decryption of message $m$ for user $\mathcal{U}$ and the Combined TEE. The Ciphertext $C$ is a tuple of $(C_1, C_2)$ with $C_1 = r \cdot G$ and $C_2 = m + r \cdot Y$ where $r$ is random. The decryption works by sending $C_1$ to the TEEs which return $C_1 \cdot x$. The client then calculates the sum of these shares and adds it to $C_2$ to obtain the original message.

the security of both key shares as long as one TEE remains uncompromised, the protocol $\mathfrak{G}$ is sufficient to guarantee the key security for $\mathcal{U}$ in a stand-alone protocol.

### 5.2.3 Signing

The last one-party protocol we discuss is the process of signing a message $m$ with a TEE. This allows a user to create signatures where the signing key is attestably held inside a TEE. Here, we assume that the signed message can be public and does not need to be hidden. If the user needs to retain the confidentiality of the signed content, she can for example calculate a blinded cryptographic hash over the message and use that as an input into the signing protocol. Desired security properties of the signing protocol are as follows:

- The private signing key must only be known by the TEE and the adversary must not be able to create valid signatures.
- The public signature verification key must be attestable by remote parties and linkable to one specific TEE.

**$\mathbb{S}$: Signature generation with the Ideal TEE (Figure A.6)**
As the Ideal TEE can be used as a TTP, user $\mathcal{U}$ can send a message $m$ to the Ideal TEE $\mathcal{I}$ to which it returns a signature $\mathbb{S}(m)$. Assuming the Ideal TEE previously published its public signature verification key together with a quote binding this key to its own identity, remote parties can verify that the signature is valid and generated by the Ideal TEE.
This Ideal TEE protocol can be used as a proxy for TEE attestation where messages that originate from the user $\mathcal{U}$ or the Ideal TEE $\mathcal{I}$ only have to be attested once (for the signing key) and can from then on be verified based on a signature

from this key. While such a proxy would still require one attestation round, it prevents future attestations and reduces the overall complexity of the surrounding system. It is also possible to extend this protocol by binding the use of the signing key to some policy [46]. Such a policy would be published together with the initial quote that proves the authenticity of the signing key and could limit the use of the signing key or enforce certain environment variables to hold for the Ideal TEE to sign any message. However, for the Ideal TEE, these extended protocols can trivially be based on the same principles as the function $\mathbb{S}$ and are not discussed further.

**$\mathfrak{S}$: Signing with the Combined TEE (Figure 5.10)**

Figure 5.10 depicts the protocol $\mathfrak{S}$ to sign messages with the Combined TEE using any standard signature algorithm. The combined public key of the signature scheme is seen as a vector $PK = [pk_1, pk_2]$ where $pk_i$ is a public key generated by TEE $\mathcal{T}_i$ and is assumed to be published in conjunction with a quote from $\mathcal{T}_i$ attesting its origin. To convince remote parties of the origin pf $PK$, $\mathcal{U}$ has to publish it together with both quotes from $\mathcal{T}_1$ and $\mathcal{T}_2$. Any remote party can then verify both signatures in the public key vector $PK$. At the start of $\mathfrak{S}$, user $\mathcal{U}$ sends a *request_signature* message that contains $m$. Next, both TEEs send back their signature share of $m$ based on their own public key $pk_i$. $\mathcal{U}$ can now combine both signatures $s_1$ and $s_2$ into the signature vector $S = [s_1, s_2]$. When verifying the signature vector $S$, it is important to verify both signatures for their correctness. $S$ is only to be seen as valid if both signature shares are valid.
Alternatively, it would also be possible to modify the protocol to utilize threshold signatures such as distributed Schnorr signatures [51] which would decrease the complexity of the protocol as only one public key and one signature would need to be stored and verified by $\mathcal{U}$. If at least one TEE is uncompromised, the adversary will only ever be able to falsely generated signatures with $\mathcal{T}_M$ while $\mathcal{T}_S$ will refuse to generate a signature for the adversary. This is independent of whether $\mathcal{M}$ is a weak or strong adversary, as signatures from both $\mathcal{T}_1$ and $\mathcal{T}_2$ are required for the set of signatures to be valid. As such, the protocol $\mathfrak{S}$ provides all necessary security properties with arbitrary signing algorithms. The benefit of this is that by using standard signature algorithms, implementing the protocol does not require any major changes on the side of remote parties besides the fact that they have to receive and verify two signatures instead of one.

## 5.3 Two party protocols

Building on the one party protocols as a base, we can build more complex protocols between two users of the TEE. In contrast to the one party protocols, these protocols require active online involvement of both parties that goes beyond a simple verification of attestations or signatures. The first such protocol is a policy-based store-and-forward scheme where a user wishes to forward a secret to another user under the restriction that the receiver can only retrieve the secret when a certain policy has been matched. The second protocol is an $m$ of $n$ oblivious transfer where a user wants to share $m$ elements of a set of $n$ elements with another user without the receiver learning the other $n-m$ elements or the originating user learning which $m$ elements were requested.

### 5.3.1 Policy based store-and-forward

A policy based store-and-forward scheme involves a user $\mathcal{A}$ who wishes to share a secret $s$ with user $\mathcal{B}$ under the restriction that $\mathcal{B}$ can only retrieve $s$ when a certain policy has been matched. For sharing, $\mathcal{A}$ uses the TEE as a TTP to store the secret and only forward it to $\mathcal{B}$ when $\mathcal{B}$ can present an input that matches the specified policy. Such policy could be time based (e.g. only access the secret on or after a specific date), condition based (e.g. only access if $\mathcal{B}$ can present an authorization code), or any other restriction that can be verified by the TEE. In any case, the policy is bound to the target of $s$ and can as such only be matched by $\mathcal{B}$. However, the policy can also be targeted towards $\mathcal{A}$ herself or even to a set of users. If $\mathcal{U}$ wishes to share a secret with herself, she simply has to take both roles in the protocol and specify herself in the policy. For clarity and without loss of generality, we discuss store-and-forward in the context of $\mathcal{A}$ sharing a secret with $\mathcal{B}$. In the following, we discuss the policy $\mathcal{P}$ under the assumption that it consists of two parts: A public signing key and a rule $\mathcal{R}$. The public signing key serves as an element of authorization to only allow specific users

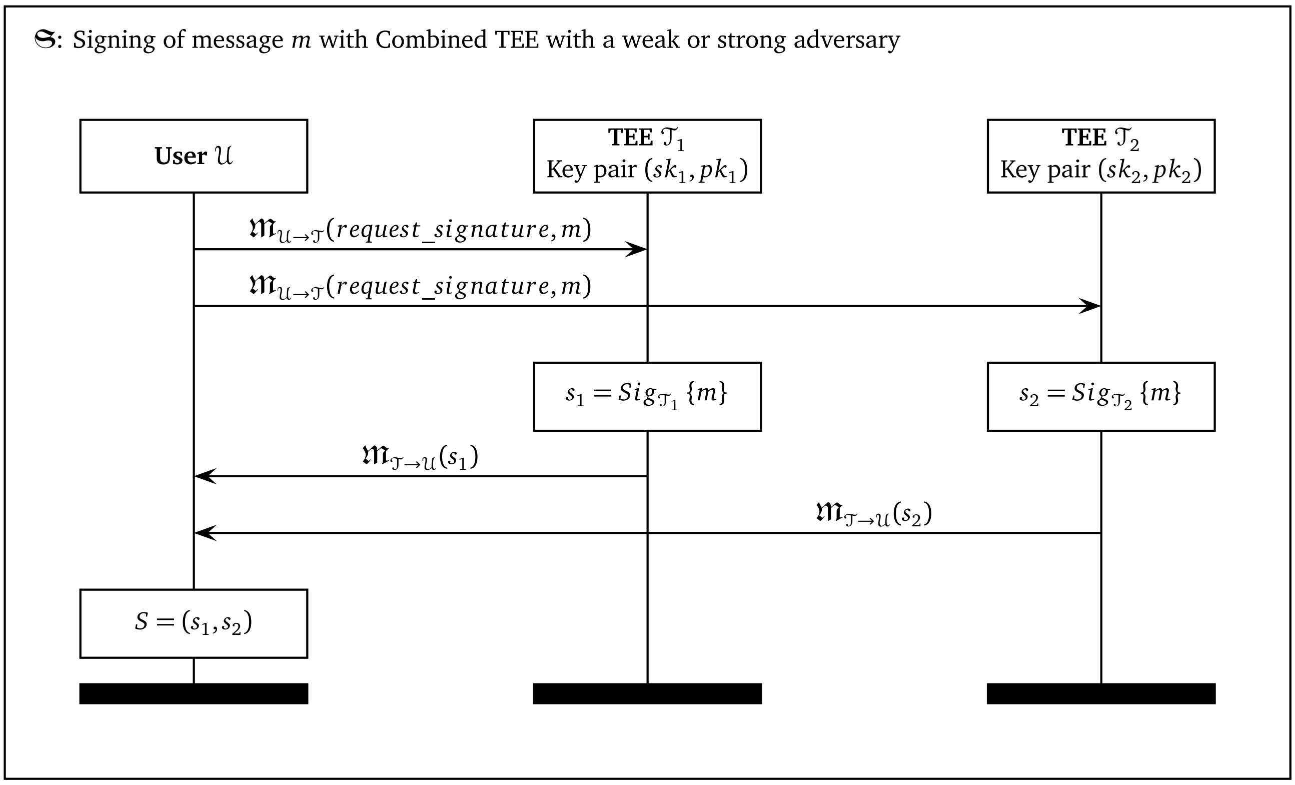

$\mathfrak{S}$: Signing of message $m$ with Combined TEE with a weak or strong adversary
User $\mathcal{U}$
TEE $\mathcal{T}_1$ Key pair $(sk_1, pk_1)$
TEE $\mathcal{T}_2$ Key pair $(sk_2, pk_2)$
$\mathfrak{m}_{\mathcal{U}\rightarrow\mathcal{T}}(request\_signature, m)$
$\mathfrak{m}_{\mathcal{U}\rightarrow\mathcal{T}}(request\_signature, m)$
$s_1 = Sig_{\mathcal{T}_1}\{m\}$
$s_2 = Sig_{\mathcal{T}_2}\{m\}$
$\mathfrak{m}_{\mathcal{T}\rightarrow\mathcal{U}}(s_1)$
$\mathfrak{m}_{\mathcal{T}\rightarrow\mathcal{U}}(s_2)$
$S = (s_1, s_2)$

**Figure 5.10.:** Signing of a message $m$ between user $\mathcal{U}$ and the Combined TEE $\mathcal{C}$ consisting of the actual TEEs $\mathcal{T}_1$ and $\mathcal{T}_2$. $\mathcal{U}$ sends a message $m$ to be signed by both TEEs and obtains a combined Signature $S$ that consists of two separate signatures, one of each of the TEEs. $\mathcal{U}$ has access to both public keys $pk_1$ and $pk_2$ and the combined public key is the tuple $PK = (pk_1, pk_2)$. A signature only verifies as correct if both (all) sub-signatures verify as correct.

to access $s$, while $\mathcal{R}$ can be seen as a function that matches any input $i$ to a boolean value and either grants or denies access to $s$. $s$ should only be shared with $\mathcal{B}$ if he authenticates with the correct public key and can present an input $i$ that matches the rule $\mathcal{R}$.
The desired security guarantees of a policy based store-and-forward protocol are as follows:

1. The secret $s$ must not be leaked to or influenced by $\mathcal{M}$.
2. Only $\mathcal{B}$ must be able to retrieve $s$.
3. $\mathcal{B}$ must not be able to retrieve $s$ until policy $\mathcal{P}$ can be satisfied, possibly requiring input $i$ from $\mathcal{B}$.

**$\mathbb{SF}(s)$: Store-and-forward with Ideal TEE (Figure A.7)**

Realizing a store-and-forward scheme with an Ideal TEE is simple. First, $\mathcal{A}$ sends her secret $s$ together with the ID-specific policy $\mathcal{P}_{\text{ID}}$ to $\mathcal{I}$ in an initial *store* message. In addition to her input, this message also contains a unique identifier ID that is also known by $\mathcal{B}$ and that can later be used by $\mathcal{B}$ to retrieve $s$. Next, $\mathcal{B}$ sends a *forward* message with this ID and input $i$ to $\mathcal{I}$. If $\mathcal{I}$ can correctly match $\mathcal{B}$'s original public signing key and input to the policy, it returns $s$, otherwise it returns an error.
$\mathcal{M}$ can neither retrieve nor influence $r$ in any way and due to $\mathcal{I}$ performing the input check internally, $\mathcal{B}$ can also not cheat by sending an incorrect input.

**$\mathfrak{SF}(s)$: Store-and-forward with Combined TEE (Figure 5.11)**

For the case of real-world TEEs, the secret can never be sent to one TEE in plaintext as $\mathcal{M}$ would easily be able to read that secret if the TEE were compromised. As a solution, we employ secret sharing between the two TEEs $\mathcal{T}_1$ and $\mathcal{T}_2$. To

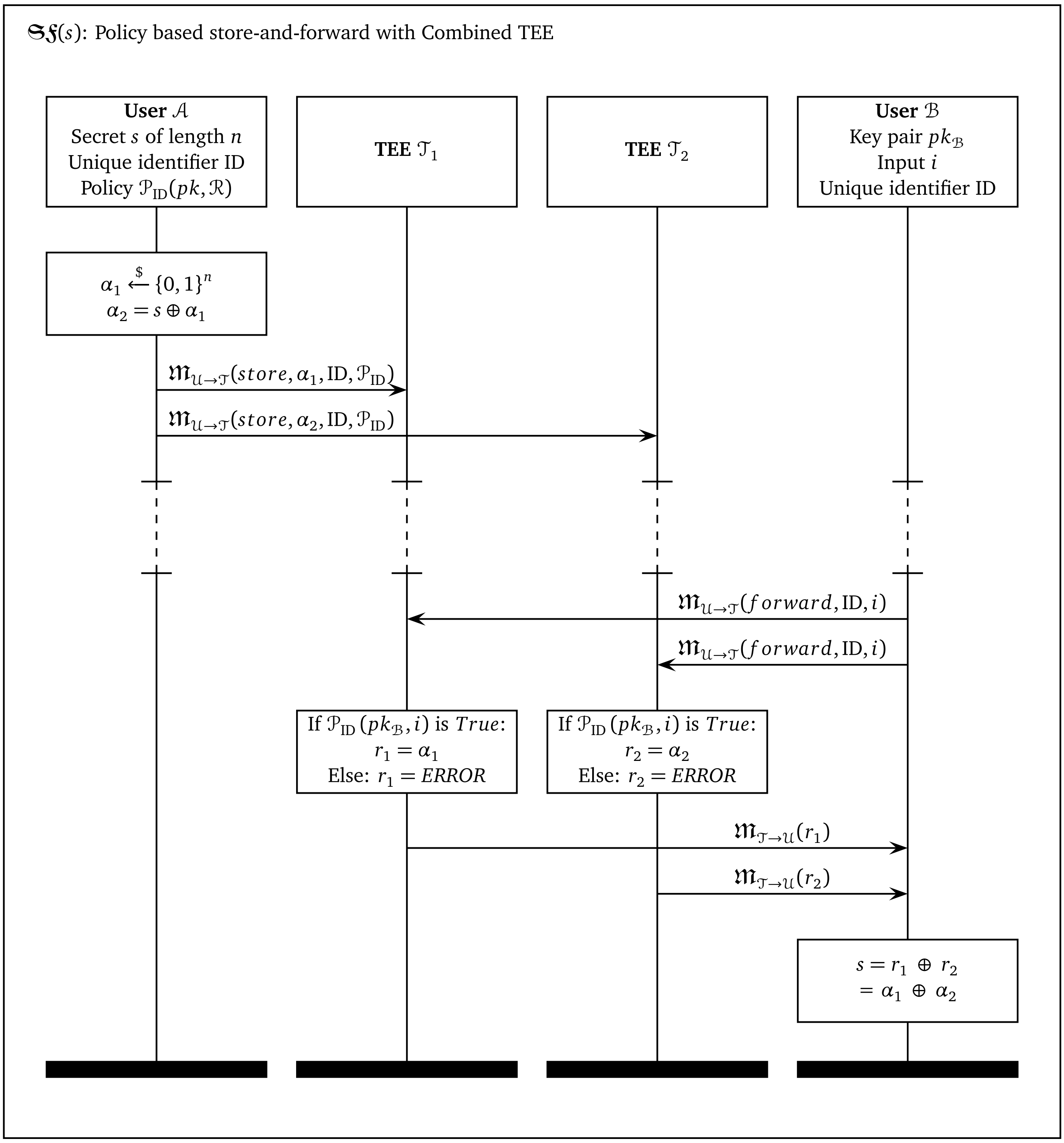

$\mathfrak{SF}(s)$: Policy based store-and-forward with Combined TEE
User $\mathcal{A}$
Secret $s$ of length $n$
Unique identifier ID
Policy $\mathcal{P}_{\mathrm{ID}}(pk, \mathcal{R})$
TEE $\mathcal{T}_1$
TEE $\mathcal{T}_2$
User $\mathcal{B}$
Key pair $pk_{\mathcal{B}}$
Input $i$
Unique identifier ID
$\alpha_1 \xleftarrow{\$} \{0,1\}^n$
$\alpha_2 = s \oplus \alpha_1$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(store, \alpha_1, \mathrm{ID}, \mathcal{P}_{\mathrm{ID}})$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(store, \alpha_2, \mathrm{ID}, \mathcal{P}_{\mathrm{ID}})$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(forward, \mathrm{ID}, i)$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(forward, \mathrm{ID}, i)$
If $\mathcal{P}_{\mathrm{ID}}(pk_{\mathcal{B}}, i)$ is $True$: $r_1 = \alpha_1$ Else: $r_1 = ERROR$
If $\mathcal{P}_{\mathrm{ID}}(pk_{\mathcal{B}}, i)$ is $True$: $r_2 = \alpha_2$ Else: $r_2 = ERROR$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(r_1)$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(r_2)$
$s = r_1 \oplus r_2 = \alpha_1 \oplus \alpha_2$

**Figure 5.11.:** Policy based store-and-forward from user $\mathcal{A}$ to $\mathcal{B}$ with the Combined TEE $\mathcal{C}$ consisting of the actual TEEs $\mathcal{T}_1$ and $\mathcal{T}_2$. In the following denoted as $\mathfrak{SF}(s)$. $\mathcal{A}$ shares a secret with each $\mathcal{T}_1$ and $\mathcal{T}_2$, and binds them to an identifier ID, and policy $\mathcal{P}$ that consists of a public key $pk$ and a rule $\mathcal{R}$. Only users that have the specified public key and that present a valid input $i$ that matches rule $\mathcal{R}$ can retrieve the secret with the identifier ID. $\mathcal{A}$ and $\mathcal{B}$ can be identical for the sake of this protocol if $\mathcal{A}$ wishes to hide a secret until a certain set of conditions are met again (e.g. external signatures, a certified time period has passed, etc).

do this, $\mathcal{A}$ prepares a random string $\alpha_1$ of the same length as $s$ and calculates $\alpha_2 = s \oplus \alpha_1$. Next, $\mathcal{A}$ sends the initial *store* message to $\mathcal{T}_1$ and $\mathcal{T}_2$ with the ID, the ID-specific policy $\mathcal{P}_{\text{ID}}$, and $\alpha_1$ or $\alpha_2$ respectively. After both TEEs received their share, $\mathcal{B}$ can at any future point send a *forward* message to $\mathcal{T}_1$ and $\mathcal{T}_2$ together with the ID and an input $i$ for the rule $\mathcal{R}$. $\mathcal{T}_1$ and $\mathcal{T}_2$ then act similar to $\mathcal{I}$ in the Ideal TEE version of the protocol and verify that the input and the public signing key of $\mathcal{B}$ match $\mathcal{P}_{\text{ID}}$. Only if the policy is matched correctly, the TEEs return $\alpha_1$ and $\alpha_2$ respectively which $\mathcal{B}$ can combine to retrieve $s$.

**Notes on secret sharing with Combined TEE**

There are two important notes to be made on secret sharing with the Combined TEE. The first note is that interestingly, the store-and-forward protocol does not differ if the adversary is $\mathcal{M}_S$ or $\mathcal{M}_W$. This is because $\mathcal{T}_S$ will enforce that its share is not revealed until $i$ matches $\mathcal{P}_{\text{ID}}$ while the secret held by $\mathcal{T}_M$ is leaked to $\mathcal{M}$ no matter if he is $\mathcal{M}_S$ or $\mathcal{M}_W$. However, $\mathcal{M}_S$ could tamper with the secret returned by $\mathcal{T}_M$ which we assume is out of scope as $\mathcal{A}$ can include a cryptographic MAC in $s$ which $\mathcal{B}$ checks upon calculating $s$ based on the two shares.

The second note is that in the presence of $\mathcal{M}_W$, it is possible to use an authenticated message from $\mathcal{B}$ to the TEEs $\mathcal{T}_1$ and $\mathcal{T}_2$. In doing so, the TEEs could ensure that any request they get from $\mathcal{B}$ is authenticated and can not originate from $\mathcal{M}_W$. However, $\mathcal{T}_M$ could leak the secret to $\mathcal{B}$ if $\mathcal{M}_W$ colludes with $\mathcal{B}$. $\mathcal{T}_S$ would not be compromised and would only reveal $s$ upon a message by $\mathcal{B}$, regardless if this is an authenticated message or simply encrypted with the session key that is not known by $\mathcal{M}_W$. We conclude that even if one uses authenticated messages, there is no added security for $\mathcal{M}_W$.

### 5.3.2 Oblivious transfer

The last protocol we discuss is also a protocol for two parties and handles the functionality of oblivious transfer. In the general oblivious transfer setting, a user $\mathcal{A}$ has a vector $S$ that contains $n$ secrets and wants to share $m$ elements of this vector with $\mathcal{B}$. For this, $\mathcal{B}$ chooses $m$ of those elements and should not gain any information about elements that he did not choose. Additionally, $\mathcal{A}$ should not learn what choice $\mathcal{B}$ made. Oblivious transfer is considered to be an essential problem of Two-Party Computation (2PC) and can be used as a foundation for other Multi-Party Computation (MPC) primitives [30, 43]. It has also been widely researched in various variations, e.g. with the involvement of third parties [59, 26], without third parties [63, 58], and in various other settings [57, 10]. In summary, an oblivious transfer protocol needs to give the following security guarantees:

1. $\mathcal{M}$ should not learn any of the shared elements of $S$, nor should he be able to tamper with, nor learn which elements $\mathcal{B}$ chooses.

2. $\mathcal{B}$ should only be able to choose at most $m$ elements and not learn anything about the elements he did not choose.

3. $\mathcal{A}$ only knows $m$ and should not learn which elements $\mathcal{B}$ chose.

$\mathbb{OT}_n^m(S)$: **Oblivious transfer with Ideal TEE (Figure A.8)**

With the Ideal TEE $\mathcal{I}$, an oblivious transfer can easily be realized by using $\mathcal{I}$ as a TTP and handing it the whole set of secrets while letting $\mathcal{I}$ enforce the desired security guarantees. As such, $\mathcal{A}$ sends an initial *transfer* message to $\mathcal{I}$ together with the set of secrets $S$. Next, $\mathcal{B}$ can send a *reveal* message with the set of indices $M$ that he chose from the set. Upon receiving both messages, $\mathcal{I}$ will only once check if the received $M$ has length $m$ and return the desired subset of $S$ to $\mathcal{B}$. $\mathcal{M}$ can neither tamper with nor see the set of secrets and due to $\mathcal{I}$ being an Ideal TEE, neither $\mathcal{A}$ nor $\mathcal{B}$ can cheat.

$\mathfrak{OT}_n^m(S)$: **Oblivious transfer with Combined TEE (Figures 5.12 and 5.13)**

With $\mathcal{M}_W$, the Ideal TEE protocol $\mathbb{OT}_n^m$ is not secure as the adversary could easily compromise the real-world TEE and view the whole set $S$ in plaintext and see $\mathcal{B}$'s input. A naïve approach to oblivious transfer with $\mathcal{C}$ is to use a scheme similar to $\mathfrak{SS}$ above where each TEE receives a part of the secret and only forwards up to $m$ of these shares to $\mathcal{B}$.

However, one core requirement of $\mathfrak{OT}_n^m$ is that $\mathcal{A}$ does not learn which secrets $\mathcal{B}$ chose to retrieve. If $\mathcal{A}$ colludes with $\mathcal{M}$, she can easily detect which of the secrets were retrieved by $\mathcal{B}$ and undermine the security guarantees of the protocol.

As an alternative to secret sharing, we deploy an offline and an online phase to solve oblivious transfer in the presence of $\mathcal{M}_W$ and $\mathcal{M}_S$ simultaneously. The offline phase sets up a key for each element of $S$ that depends on secrets held by $\mathcal{T}_1$ and $\mathcal{T}_2$. This set of keys is then used by $\mathcal{A}$ in the online phase to encrypt the whole set and send it to $\mathcal{B}$ who can retrieve $m$ subkeys from each TEE and decrypt all the shares he desires, up to an amount of $m$. In the following, it is assumed that all secrets in $S$ have length $l$. This is to ensure that $\mathcal{A}$ can not deduce which elements $\mathcal{B}$ picked by detecting the length of the key shares that $\mathcal{B}$ handles. It is furthermore assumed that $\mathcal{A}$ has access to a Goldwasser-Micali key pair as explained in Section 2.1.3 while $\mathcal{B}$ has access to a permutation $\sigma$ with $\sigma : \mathcal{D}^n \to \mathcal{D}^n$.

**Offline phase:** The offline phase starts with $\mathcal{A}$, $\mathcal{T}_1$, and $\mathcal{T}_2$ each generating a random $A$, $Q$, and $R$ respectively, each with $n$ elements of length $l$. This randomness is taken from the same distribution $\mathcal{D}^l$ that the elements of $S$ are in. Additionally, $\mathcal{A}$ queries two random variables $\alpha_1$ and $\alpha_2$ from $\mathcal{D}^l$ as blinding factors for $\mathcal{T}_1$ and $\mathcal{T}_2$. With $A$, $\alpha_1$ and $\alpha_2$, and $pk_{\mathcal{A}}$, $\mathcal{A}$ can prepare the message that she sends to $\mathcal{T}_1$ and $\mathcal{T}_2$. First, she calculates the element-wise XOR of $\alpha_1 \oplus A$ and $\alpha_2 \oplus A$ respectively into a vector of blinded values of $A$. Next, she encrypts these blinded vectors with her own Goldwasser-Micali public key $pk_{\mathcal{A}}$ and sends the vectors to $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. The resulting vector that is sent to $\mathcal{T}_1$ is $[Enc_{\mathcal{A}}(A_1 \oplus \alpha_1), ..., Enc_{\mathcal{A}}(A_n \oplus \alpha_1)]$ while the vector sent to $\mathcal{T}_2$ is constructed similarly based on $\alpha_2$. Each TEE now performs the following operation on the received blinded vector: First, encrypt the vector of $Q$ and $R$ respectively with $\mathcal{A}$'s public key and homomorphically perform the XOR operation with the received vector. With the Goldwasser-Micali crypto system, this means that $\mathcal{T}_1$ encrypts each element of $Q$ to produce the vector $Q_{\mathcal{A}} = [Enc_{\mathcal{A}}(Q_1), ..., Enc_{\mathcal{A}}(Q_n)]$ and performs the homomorphic operation $Enc_{\mathcal{A}}(Q_i) \cdot Enc_{\mathcal{A}}(A_i \oplus \alpha_1) = Enc_{\mathcal{A}}(A_i \oplus \alpha_1 \oplus Q_i)$ for each of the $n$ elements of the vectors. $\mathcal{T}_2$ performs a similar calculation based on $R$ and $\alpha_2$. After both TEEs have sent their calculated vector with the homomorphic operation to $\mathcal{B}$, $\mathcal{B}$ can combine them element-wise and retrieve $Z = [Z_1, ..., Z_n]$ with $Z_i = Enc_{\mathcal{A}}(A_i \oplus \alpha_1 \oplus Q_i) \cdot Enc_{\mathcal{A}}(A_i \oplus \alpha_2 \oplus R_i) = Enc_{\mathcal{A}}A_i \oplus \alpha_1 \oplus Q_i \oplus A_i \oplus \alpha_2 \oplus R_i) = Enc_{\mathcal{A}}(\alpha_1 \oplus \alpha_2 \oplus Q_i \oplus R_i)$. As a last step of the offline phase, $\mathcal{B}$ sends the permutation $\sigma(Z)$ to $\mathcal{A}$. This ensures that $\mathcal{A}$ is not aware of the order of elements in $Z$ and can not map elements in $Z$ to the TEE key shares $Q$ and $R$. Once $\mathcal{A}$ received the permutation of $Z$, she decrypts each element with her secret key $sk_{\mathcal{A}}$ and calculates the vector of keys $K = [K_1, ..., K_n]$ with $K_i = Dec_{\mathcal{A}}(Z_i) \oplus \alpha_1 \oplus \alpha_2 = Dec_{\mathcal{A}}\left(Enc_{\mathcal{A}}(\alpha_1 \oplus \alpha_2 \oplus Q_{\sigma(i)} \oplus R_{\sigma(i)})\right) \oplus \alpha_1 \oplus \alpha_2 = \alpha_1 \oplus \alpha_2 \oplus Q_{\sigma(i)} \oplus R_{\sigma(i)} \oplus \alpha_1 \oplus \alpha_2 = Q_{\sigma(i)} \oplus R_{\sigma(i)}$ where $\sigma(i)$ is the permutation of the $i$-th element of the vector. Intuitively, the aim of the offline phase is to create a set of $n$ keys where each key is split between the two TEEs. $\mathcal{A}$ knows the complete keys but does not know which TEE shares these correspond to because of the shuffling, while $\mathcal{B}$ knows the shuffled order but not the keys themselves.

**Online phase:** $\mathcal{A}$ can use the keys from the offline phase in the online phase to encrypt the secrets without $\mathcal{B}$ being able to have cheated as the intermediate elements of the keys depended on $\alpha_1$ and $\alpha_2$. $\mathcal{B}$ on the other hand can query the TEEs for any key share that $\mathcal{B}$ requests without $\mathcal{A}$ being able to revert the permutation $\sigma$ At the beginning of the online phase, $\mathcal{A}$ is assumed to have access to the set of secrets $S$ and the vector of prepared keys from the offline phase $K$. $\mathcal{B}$ is assumed to have access to the permutation $\sigma$ that was used in the offline phase and additionally to have chosen a set of $m$ indices that he wants to choose from $S$. The online phase starts with $\mathcal{A}$ preparing $S_K = [S_{K_1}, ..., S_{K_n}]$ with $S_{K_i} = S_i \oplus K_i = S_i \oplus Q_{\sigma(i)} \oplus R_{\sigma(i)}$ and sending $S_K$ to $\mathcal{B}$. At the same time $\mathcal{B}$ prepares a vector $J = [J_1, ..., J_n]$ with $J_i = \sigma^{-1}(M_i)$ which contains the indices that need to be queried from $\mathcal{T}_1$ and $\mathcal{T}_2$ to retrieve the key shares to index $i$. After receiving $S_K$ from $\mathcal{A}$, $\mathcal{B}$ sends $J$ to $\mathcal{T}_1$ and $\mathcal{T}_2$. These will accept a $J$ once and, if its length is a maximum of $m$, return the requested indices of Q and R respectively. Finally, $\mathcal{B}$ can retrieve the desired vector $S_M = [S_{M_1}, ..., S_{M_n}]$ with $S_{M_i} = S_{K_i} \oplus Q_{J_i} \oplus R_{J_i}$.

### Notes on oblivious transfer with Combined TEE

It is not immediately apparent why the protocol requires each step to provide all security guarantees. Explicitly, we now explain the difference between the values of $\alpha$ and $A$, the use of the Goldwasser-Micali crypto system, and the necessity of $\sigma$.

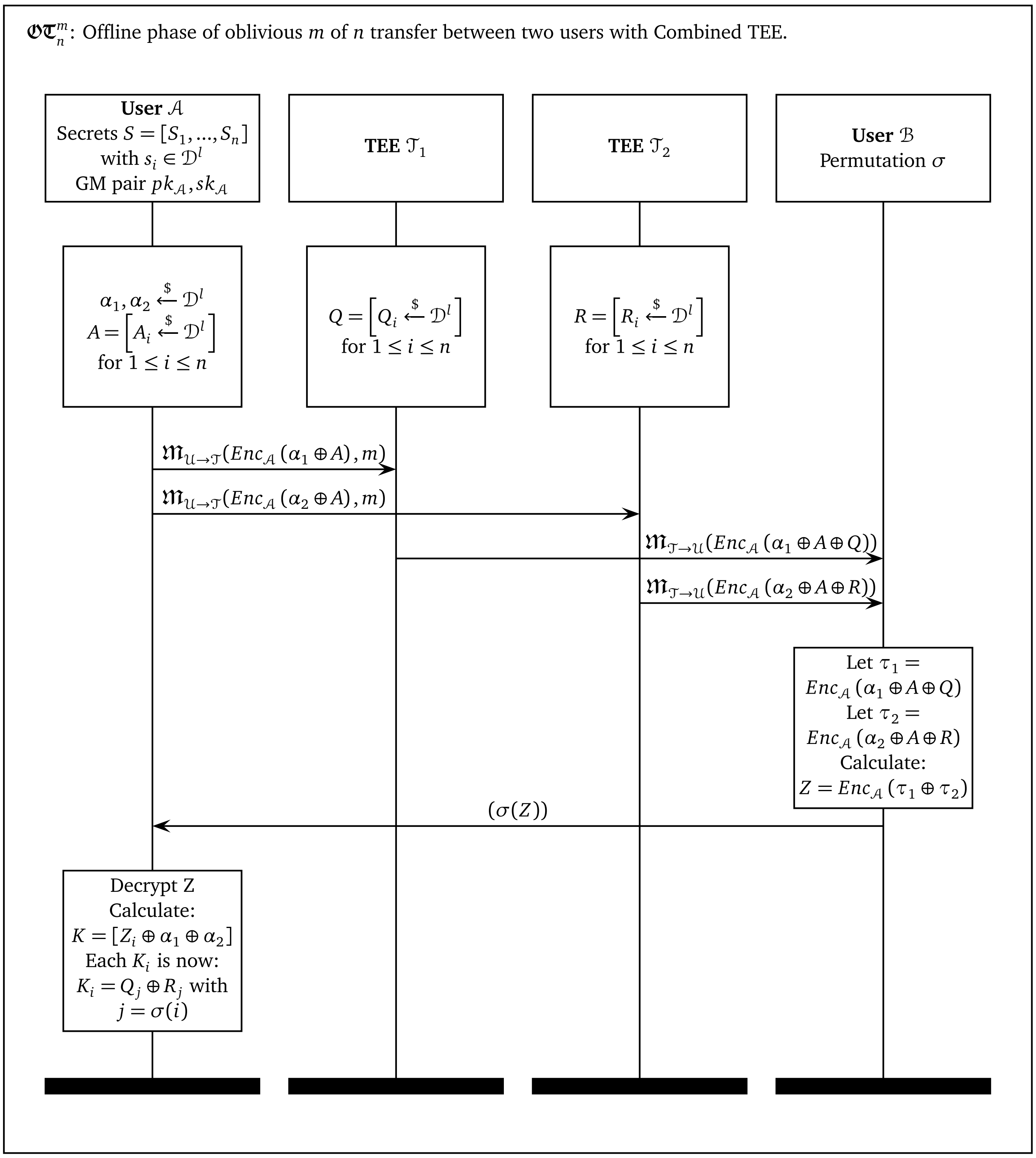

$\mathfrak{OT}_n^m$: Offline phase of oblivious $m$ of $n$ transfer between two users with Combined TEE.
User $\mathcal{A}$
Secrets $S = [S_1, ..., S_n]$ with $s_i \in \mathcal{D}^l$
GM pair $pk_\mathcal{A}, sk_\mathcal{A}$
TEE $\mathcal{T}_1$
TEE $\mathcal{T}_2$
User $\mathcal{B}$
Permutation $\sigma$
$\alpha_1, \alpha_2 \xleftarrow{\$} \mathcal{D}^l$
$A = \left[A_i \xleftarrow{\$} \mathcal{D}^l\right]$ for $1 \le i \le n$
$Q = \left[Q_i \xleftarrow{\$} \mathcal{D}^l\right]$ for $1 \le i \le n$
$R = \left[R_i \xleftarrow{\$} \mathcal{D}^l\right]$ for $1 \le i \le n$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(Enc_\mathcal{A}(\alpha_1 \oplus A), m)$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(Enc_\mathcal{A}(\alpha_2 \oplus A), m)$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(Enc_\mathcal{A}(\alpha_1 \oplus A \oplus Q))$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(Enc_\mathcal{A}(\alpha_2 \oplus A \oplus R))$
Let $\tau_1 = Enc_\mathcal{A}(\alpha_1 \oplus A \oplus Q)$
Let $\tau_2 = Enc_\mathcal{A}(\alpha_2 \oplus A \oplus R)$
Calculate: $Z = Enc_\mathcal{A}(\tau_1 \oplus \tau_2)$
$(\sigma(Z))$
Decrypt Z
Calculate: $K = [Z_i \oplus \alpha_1 \oplus \alpha_2]$
Each $K_i$ is now: $K_i = Q_j \oplus R_j$ with $j = \sigma(i)$

**Figure 5.12.:** Offline phase of oblivious $m$ of $n$ transfer between to parties, $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{A}$ wants to send $S = [S_i]$ to $\mathcal{B}$ without revealing $S \setminus \{S_j\}$ for $j$ in $1 \le j \le m$ while $\mathcal{B}$ does not want to reveal any $j$.

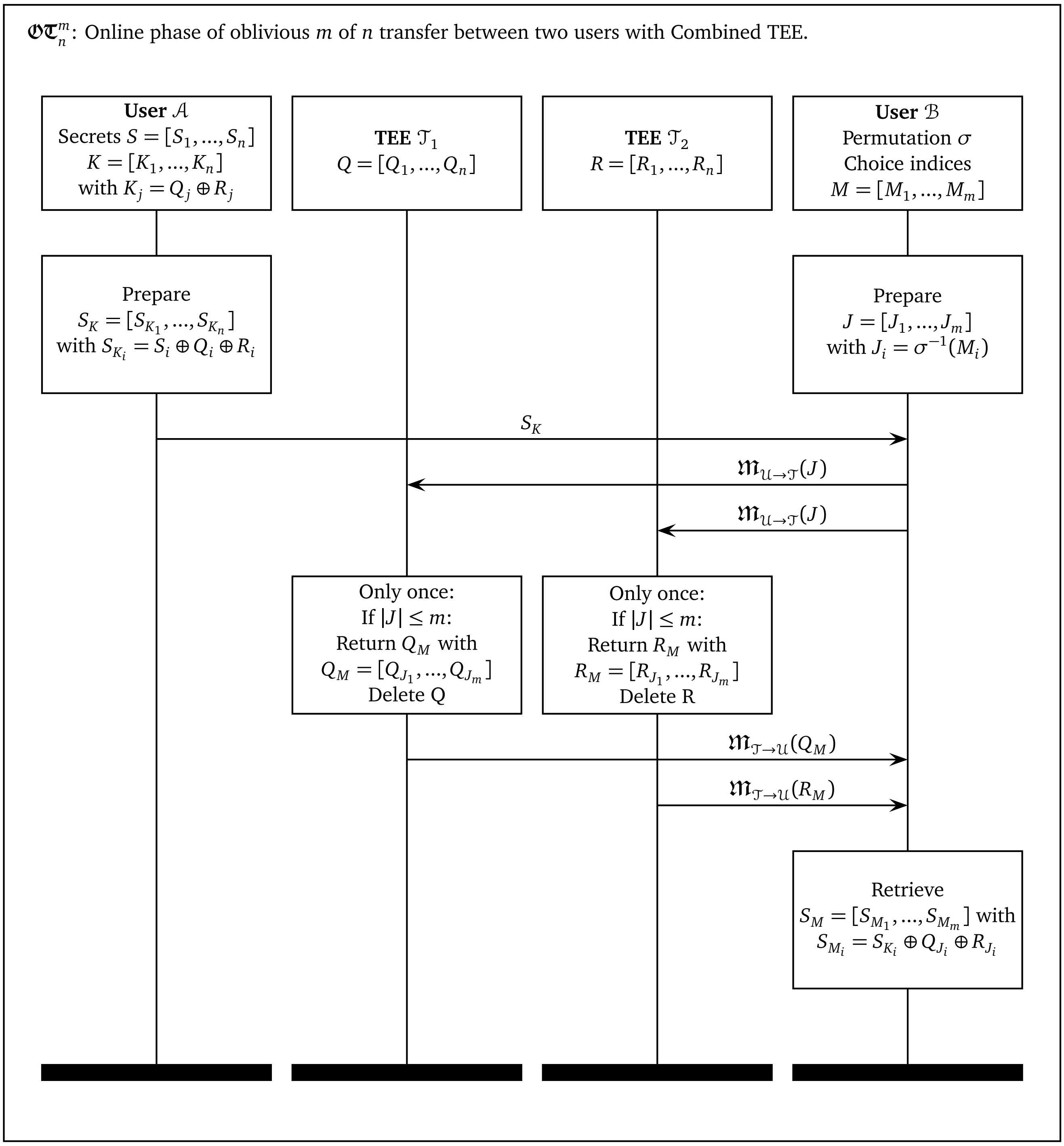

$\mathfrak{OT}^m_n$: Online phase of oblivious $m$ of $n$ transfer between two users with Combined TEE.
**User** $\mathcal{A}$
Secrets $S = [S_1, ..., S_n]$
$K = [K_1, ..., K_n]$
with $K_j = Q_j \oplus R_j$
**TEE** $\mathcal{T}_1$
$Q = [Q_1, ..., Q_n]$
**TEE** $\mathcal{T}_2$
$R = [R_1, ..., R_n]$
**User** $\mathcal{B}$
Permutation $\sigma$
Choice indices
$M = [M_1, ..., M_m]$
Prepare
$S_K = [S_{K_1}, ..., S_{K_n}]$
with $S_{K_i} = S_i \oplus Q_i \oplus R_i$
Prepare
$J = [J_1, ..., J_m]$
with $J_i = \sigma^{-1}(M_i)$
$S_K$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(J)$
$\mathfrak{m}_{\mathcal{U}\to\mathcal{T}}(J)$
Only once:
If $|J| \leq m$:
Return $Q_M$ with
$Q_M = [Q_{J_1}, ..., Q_{J_m}]$
Delete Q
Only once:
If $|J| \leq m$:
Return $R_M$ with
$R_M = [R_{J_1}, ..., R_{J_m}]$
Delete R
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(Q_M)$
$\mathfrak{m}_{\mathcal{T}\to\mathcal{U}}(R_M)$
Retrieve
$S_M = [S_{M_1}, ..., S_{M_m}]$ with
$S_{M_i} = S_{K_i} \oplus Q_{J_i} \oplus R_{J_i}$

**Figure 5.13.:** Online phase of oblivious $m$ of $n$ transfer between to parties, $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{A}$ wants to send $S = [S_i]$ to $\mathcal{B}$ without revealing $S \setminus \{S_j\}$ for $j$ in $1 \leq j \leq m$ while $\mathcal{B}$ does not want to reveal any $j$.

**$\alpha$ and $A$:** Both the blinding values $\alpha_i$ and the vector of blinding values $A$ are essential in this protocol. First, the values of $\alpha_1$ and $\alpha_2$ enforce that $\mathcal{B}$ correctly uses the values he receives from $\mathcal{T}_1$ and $\mathcal{T}_2$. If $\mathcal{B}$ could simply use his own random values and forward these as $Z$ to $\mathcal{A}$, he would trivially be able to revert the encryption that is used in $S_K$ during the online phase. With blinding through $\alpha_i$ however, $\mathcal{A}$ herself eliminates the blinding factors at the end of the offline phase. If $\mathcal{B}$ did not properly correctly send the values he received from $\mathcal{T}_1$ and $\mathcal{T}_2$, then the result of the offline phase will again be blinded by $\alpha_i$ and $\mathcal{B}$ learns nothing. It is possible for $\mathcal{B}$ to collude with $\mathcal{M}_S$ and omit the use of either $Q$ or $R$ and instead directly forward $\alpha_1 \oplus \alpha_2 \oplus \{Q, R\}$ depending on which TEE was compromised (i.e. either $Q$ or $R$ is all zeros). In this case, the security of the protocol relies on the values that were chosen by $\mathcal{T}_S$ which is enough to ensure the confidentiality of the key shares.

Furthermore, the necessity of the vector of $A$ is not straightforward as it cancels itself out during the calculation of $Z$ that is performed by $\mathcal{B}$. However, it is crucial to note that the $A$ ensures the correct mapping of $\mathcal{T}_1$ and $\mathcal{T}_2$ elements. $\mathcal{B}$ is forced to correctly map each item from $\mathcal{T}_1$ to its counterpart of $\mathcal{T}_2$ so that the values of $A$ will be removed from the result. If $\mathcal{B}$ would not be forced to do so, he could collude with $\mathcal{M}$ and match one element from $\mathcal{T}_S$ to many elements from $\mathcal{T}_M$. Then, he could query the one subkey from $\mathcal{T}_S$ and use it to decrypt all elements of $S_K$ with his knowledge of all subkeys of $\mathcal{T}_M$.

**Goldwasser-Micali crypto system:** The Goldwasser-Micali encryption is used to hide the evolving subkeys while they are passed through the TEEs and $\mathcal{B}$. It is especially important to hide the elements of $Z$ from $\mathcal{B}$ as he could use these values to decrypt all elements of the vector $S_K$ that he receives in the online phase after completing the protocol. To do this, he would complete the protocol normally and use the gained knowledge of any $Q_k$ and $R_k$ to retroactively calculate the blinding factors $\alpha_1$ and $\alpha_2$ used in $Z_k = \alpha_1 \oplus \alpha_2 \oplus Q_k \oplus R_k$. With these blinding factors, $\mathcal{B}$ could unblind all values of $Z$ and effectively decrypt all elements of $S_K$ himself without needing to query the TEEs.

**Permutation $\sigma$:** Lastly, $\sigma$ ensures that $\mathcal{A}$ can not deduce $\mathcal{B}$'s choices from the subkeys he requests from the TEEs. Since $\mathcal{B}$ needs to query $\mathcal{T}_1$ and $\mathcal{T}_2$ for the $Q$ and $R$ values to reveal a secret $S_i$, it would be trivial for $\mathcal{A}$ to detect the choices if she colludes with $\mathcal{M}$ who can read which item is requested from $\mathcal{T}_M$. If the requested index would be equal to the index of the desired secret, $\mathcal{A}$ would instantly know which items $\mathcal{B}$ picked for the oblivious transfer. By using a permutation in the offline phase before $\mathcal{B}$ sends $Z$ back to $\mathcal{A}$, $\mathcal{A}$ is not able to map indices of the keys to indices known by the TEEs for their $Q$ and $R$ values.

## 5.4 Expanding from two to N TEEs

All of the above protocols can be extended from two TEEs (TEE$^2$) to $N$ TEEs (TEE$^N$). We give a short description how each of the protocols have to be adjusted to work within the context of TEE$^N$. It is important to note that for any extension of TEE$^2$ to more than two participating TEEs, the user $\mathcal{U}$ needs to specify which and how many TEEs are participating in the Combined TEE at the beginning of the protocol run. If either the amount or the exact TEEs are not fixed from the beginning of any protocol run and $\mathcal{U}$ colludes with $\mathcal{M}$, they can simply compromise $k$ TEEs and pretend afterwards to have chosen $k$ as a parameter for TEE$^N$. As such, we assume that it is publicly known which TEEs participate in TEE$^N$ beforehand.

### Key exchange (Figure 5.2)

To extend the TEE$^2$ key exchange to the TEE$^N$ case, the user $\mathcal{U}$ simply has to establish a session with each element of $N$ elements of the TEE$^N$ system. The resulting combined session key then consists out of a vector of $N$ session keys, one for each TEE. As such, the key exchange scales constantly in time with the number of TEEs included in the Combined TEE as all participating TEEs can be queried concurrently. However, the key exchange scales linearly in storage as the number of session keys that need to be stored increases with every participating TEE.

**$\mathfrak{R}(n)$: Authenticated random number generation in the presence of $\mathcal{M}_W$ (Figure 5.5)**

To retrieve an authenticated random number from the Combined TEE system for a weak adversary, the user can simply query all $N$ TEEs for a random number and include these in $r_c$. As such, the $\mathfrak{R}$ scales constantly in time and linearly in storage with the number of TEEs included in the Combined TEE as all participating TEEs can be queried concurrently and their results are combined in one single $r_c$ variable.

**$\mathfrak{R}_{\mathcal{M}_S}(n)$: Authenticated random number generation in the presence of $\mathcal{M}_S$ (Figure 5.6)**

If the generated random number is to be an authenticated random number in the presence of $\mathcal{M}_S$, this means that all random number shares need to be linked to all other shares. Ideally, the algorithm should be executable with only $N$ communications (one round) and the authentication of the random number only require $N$ verifications. Figure 5.6 contains three messages for two involved TEEs:

1. $initiate\_random$: Initial request that starts a random number generation chain without a predecessor.

2. $add\_random$: Middle element in a random number generation chain that takes a previous commitment and outputs its own random value bound to the previous commitment.

3. $reveal\_random$: Last element in a random number generation chain that takes the last commitment and reveals the random value on which the initial commitment was based on. Effectively closing the chain by linking its last element to the first element.

A trivial solution to scaling this to more TEEs would be to extend the amount of $add\_random$ to $N$ TEEs. It is critical to note however, that this approach breaks when a majority of the TEEs is compromised. This is because any two compromised TEEs in succession could collaborate to change the commitment of the first TEE after the uncompromised TEE already revealed its random number.
While Figure 5.6 depicts a secure solution for TEE$^2$ with three messages, TEE$^N$ requires an algorithm that first takes a commitment from all participating TEEs, combines these commitments, and then forwards the combined commitment to all TEEs to reveal their number. Such construction separated into commitment and reveal phase is similar to the construction by Mavroudis et al. [51]. Whenever a third party wants to attest $r_c$, it has to attest all $N$ $bound\_random$ messages and recalculate the combined random number. Additionally, it has to recalculate the combined commitment based on the single commitments from each TEE and check if all TEEs use this combined commitment when revealing their number. While our TEE$^2$ protocol uses $N+1$ messages for $N=2$, the TEE$^N$ case requires $2N$ messages, $N$ for the commitment phase and $N$ for the reveal phase. As such, the strong random number generation scales linearly with TEE$^N$.

**$\mathfrak{G}$: Authenticated ElGamal key generation in the presence of $\mathcal{M}_W$ (Figure 5.7)**

Similar to $\mathfrak{R}$, $\mathfrak{G}$ can simply be expanded to TEE$^N$ by querying all $N$ TEEs with the $request\_key$ message and assembling the final public key from all $N$ public key shares. As with random number generation, $\mathfrak{G}$ scales linearly with TEE$^N$.

**$\mathfrak{G}_{\mathcal{M}_S}$: Authenticated ElGamal key generation in the presence of $\mathcal{M}_S$ (Figure 5.8)**

Similar to $\mathfrak{R}_{\mathcal{M}_S}$, $\mathfrak{G}_{\mathcal{M}_S}$ can be extended from the two-party case to $N$ parties by first performing a commitment phase followed by a reveal phase. See the explanation on authenticated random number generation above for more details. $\mathfrak{G}_{\mathcal{M}_S}$ also scales linearly with TEE$^N$.

**$\mathfrak{D}$: ElGamal decryption (Figure 5.9)**

Expanding the decryption of an ElGamal ciphertext to $N$ TEEs is trivial as the user simply has to send it to all $N$ participating TEEs and combine the received shares. This operation can be performed concurrently for all TEEs and scales constantly in time but linearly in storage as each response needs to be cached until the decryption can be calculated.

**$\mathfrak{S}$: Signing (Figure 5.10)**

Signing can be scaled similarly to $\mathfrak{R}$ where $\mathcal{U}$ simply queries more TEEs to generate a random number. Any verifier needs to check all $N$ signature shares accordingly and abort the verification if not all $N$ signatures are correct.

**$\mathfrak{SF}(s)$: Store-and-forward with Combined TEE (Figure 5.11)**

Extending $\mathfrak{SF}$ to multiple participating TEEs is directly related to extending secret sharing from 2 to $N$ parties. This can be done by generating $N-1$ random values with $\alpha_i \xleftarrow{\$} \{0,1\}^n$ where $n$ is the length of the secret to be shared, $s$. $\alpha_N$ is then calculated with $\alpha_N = (\bigoplus_{i=1}^{N} \alpha_i) \oplus s$ and each TEE $\mathcal{T}_i$ receives its share $\alpha_i$. To retrieve the secrets, $\mathcal{B}$ queries all $N$ TEEs and recombines them to retrieve $s$. The resulting message complexity scales constantly in time if all $N$ messages can be sent simultaneously, and linearly in storage as all responses need to be cached. The combination of the messages scales linearly.

**$\mathfrak{OT}_n^m$: Oblivious transfer (Figures 5.12 and 5.13)**

To extend $\mathfrak{OT}_n^m$ for the TEE$^N$ case, there are two adjustments to make. First, $\mathcal{B}$ trivially has to query $N$ TEEs for their key shares in the online phase and assemble them to retrieve the secrets $S_M$. Secondly, $\mathcal{A}$ has to adjust the blinding values for $N$ TEEs in the offline phase. This can be done similarly to extending the secret sharing shown for $\mathfrak{SF}$ above. Each blinding value consists of a TEE-static value $\alpha_i$ and a vector of random values $A_i$ of which the XOR is sent to the TEE $\mathcal{T}_i$. In the case of TEE$^2$, $A_1$ is chosen at random while $A_2$ is chosen to be $A_1$ in order for the equation $A_1 \oplus A_2 = [0]^N$ to hold, i.e. so that $A_1$ and $A_2$ cancel each other out when an element-wise XOR is calculated over the vetor. We can extend this to the case of TEE$^N$ by choosing $N-1$ random vectors and defining $A_N$ as $A_N = \bigoplus_{i=1}^{N-1} A_i$. With this definition, each TEE $\mathcal{T}_i$ will be assigned a vector of random values $A_i$ which is eliminated once the XOR with last vector $A_N$ is calculated. Since all blinding values also depend on the TEE-static values $\alpha_i$, the blindings can only be uncovered by $\mathcal{A}$ as it is the case in the TEE$^2$ version explained above.

Overall, the system scales linearly in time and storage complexity with each TEE added. If all TEEs can be contacted in parallel, this scaling is still very efficient as the XOR operations on the blinding values and the ciphertexts can be calculated efficiently. As such, this linearity can be seen as near-constant scaling for small $N$. For each element added to the oblivious transfer set $n$, one more key has to be set up during the online phase which results in a linear scaling for increases in the set size. It is important to note that there is no overhead for the increase in $m$, i.e. for the increase in items that $\mathcal{B}$ receives from the oblivious transfer, besides the increased amount of subkeys that are sent by each TEE.

# 6 Implementation

We implemented a proof of concept of our proposed design based on two instances of an Intel Software Guard Extensions (SGX) TEE written in C++ that communicate via TCP with a client written in Python. The case of two SGX enclaves is relevant as described in Section 4.2 since there might exist different implementations that are not equally trusted by two parties. With TEE$^2$, these different implementations can both be incorporated in one Combined TEE system as long as they follow the same protocols as defined in Section 5. Intel SGX is a widely deployed TEE that is available on all Intel end-user processors since the Skylake generation (2015)[1]. Additionally, Intel SGX provides an open Software Development Kit (SDK) with which trusted applications can freely be developed, requiring no further set up than an Intel SGX enabled processor. As we explained in Section 2.3.1, Intel SGX provides trusted regions of code called enclaves that can not be accessed from the untrusted operating system but instead can only be entered through predefined interfaces called ECALLs. We refer the reader to Section 2.3.1 for more details on details on Intel SGX.

## 6.1 Implementation overview

Figure 6.1 gives an overview of our implementation and shows the process of a regular operation request. A Python client communicates with a C++ application via TCP sockets and communicates directly with the TEE via encrypted messages. In detail, the client creates a JSON string that contains the command code as an integer and additional input data depending on the command type. Additional binary data in this input data is encoded as base64 and stored as a string in JSON which prevents complicated structure encodings and allows for compatibility across different implementations. The client encrypts the JSON string with Advanced Encryption Standard (AES) in combination with Galois/Counter Mode (GCM) using the shared session key that is established during the key exchange phase. AES-GCM has the advantage that it is an authenticated encryption which prevents the untrusted host from modifying messages or performing bit flips in the ciphertext without the TEE noticing these modifications. The encrypted data is sent together with a message type and the unique session ID that is assigned by the TEE during key exchange to the untrusted server application. A message type is required to distinguish between messages during key exchange where the last message requires no response, and encrypted messages where the TEE sends a response based on the query. The unique session ID is used internally by the TEE to map the correct session key to a ciphertext in order to perform the decryption. As such, it does not leak any crucial information to the untrusted server application besides meta information which it also obtained without such session ID (e.g. time, size, and amount of encrypted messages). While the untrusted application could change the session ID before forwarding the encrypted message to the TEE, this is equivalent to a DoS attack and is considered to be out of scope as the untrusted application can simply choose to ignore incoming messages. After receiving a message from a client, the untrusted server application performs the ECALL that matches the message type and passes the encrypted data and the session ID into the enclave. Upon receiving an encrypted message, the trusted application decrypts the message and executes the function that correspond to the requested command. To return the response to the client, the same order of events is executed in reverse.
We use standardized algorithms and protocols such as Elliptic Curve Diffie-Hellman (ECDH) for key exchange, Elliptic Curve Digital Signature Algorithm (ECDSA) for cryptographic signatures, AES-GCM for authenticated encryption, JSON for messages, and base64 encodings to place binary data in JSON messages. This simplifies reuse of all implemented parts and allows to extend our code base without much effort. It additionally detaches the client implementation from the TEE implementation and allows compatibility with other TEE implementations in the future. We implement the core functionality of key exchange (Figure 5.2) and random number generation with a weak adversary (Figure 5.5). During key exchange, both parties perform the ECDH exchange that is also depicted in Figure 5.2 and use this shared key to derive a shared session key that can be used by AES-GCM. The key derivation is performed with a simple SHA256 hash

[1] https://ark.intel.com

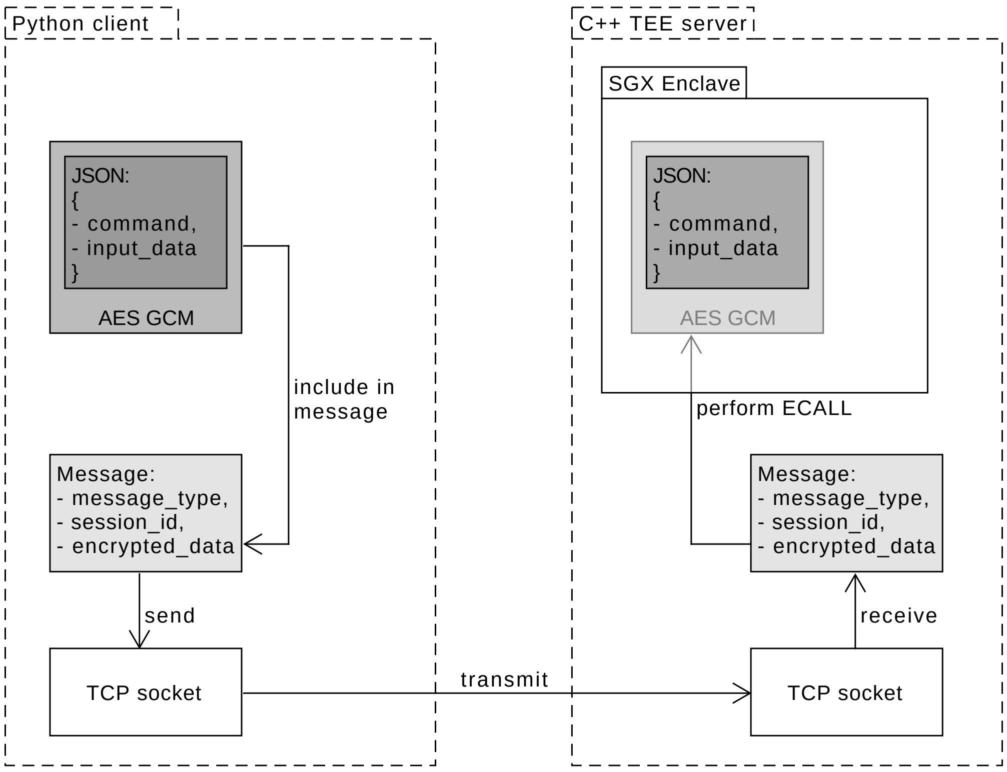


**Figure 6.1.:** High level overview of the implementation concept. A Python client encodes commands and their input as a JSON string, encrypts it with AES-GCM and appends it to a message that is sent via TCP to the C++ based TEE server. There, the encrypted message is passed as an ECALL into the enclave which processes the command. The response is handled in a reverse order and sent back to the client.

over the ephemeral ECDH key where the first half of the hash generates the 128-bit AES key and the second half is used for the 96-bit initialization vector (IV) that is necessary for GCM. Of the 96-bit IV, 64 bit are originating from this second half of the generated hash while the remaining 32 bit are used as a counter to prevent IV reuse. After every encrypted or decrypted message, both parties increment their counter on a per-message basis, effectively enabling both parties to send an overall amount of $2^{32}-1$ messages before it is necessary to generate a new IV. With such a counter, the untrusted host can reorder messages and effectively perform a DoS between these two parties (and similar problems might arise during normal, innocent behavior of the host). However, this does not affect the security as the parties can simply perform a key exchange again to reestablish their session keys. Any performance modification to handle these issues such as buffering gaps in the counter messages and waiting for delayed messages is out of scope for this work and would need to be considered when deploying such a system in practice.

## 6.2 TEE$^2$ Service with Intel SGX

We implement the service side of TEE$^2$ with C++ and use Intel SGX in its Linux SDK version 2.2.1[2]. Additionally, we utilize Boost libraries[3] to perform asynchronous networking and for utility functions such as program options of the binary. The networking based on Boost's Asio library can handle an arbitrary amount of nonblocking client connections and handles them via synchronous ECALLs whenever a client requests an operation. With $\tilde{1}000$ lines of code, the prototype TEE implementation has a small TCB and can be verified and reviewed with comparably low developer effort.

[2] https://01.org/intel-software-guard-extensions/downloads
[3] https://www.boost.org/

Listing 6.1 shows the three ECALLs that are implemented in the TEE[2] SGX enclave. The first two ECALLs handle both steps of the key exchange and the last message handles an encrypted message. While the key exchange ECALLs both handle a string input as the handled data is a JSON string, the third ecall takes in a ciphertext in binary form. The actual SGX implementation contains more pointers that define the maximum sizes of the handled pointers which are omitted from this Listing and are only used by SGX internally to prevent buffer overflows. All ECALLs are associated with a unique *session_id* that is assigned and returned by the enclave itself in the first ECALL *ecall_ke_process_msg_1*. Once assigned, this session ID is used by the enclave to keep track of open sessions and to use the correct session key for an encrypted message. Note, that this session ID does not create any vulnerability as there are only three ECALLs where the untrusted host could mishandle the session ID:

1. *ecall_ke_process_msg_1* returns the session ID to the untrusted host, so here he can only change the session ID before passing the message to the client. This has no immediate effect and will be detected in the following step.

2. There are two options for the untrusted host to misbehave during the *ecall_ke_process_msg_2* ECALL in regards to the session ID. Either he changes a message's session ID to an invalid one or he changes it to the session ID of another session. While the enclave will simply refuse to service session IDs that it does not know, an incorrect session ID will only be noticed through the incorrect signature that originates from the client (See the second client message in Figure 5.2). In both cases, the TEE will abort the key exchange.

3. If the untrusted host sends an incorrect session ID with a ciphertext in the *ecall_handle_encrypted_message* ECALL, the TEE will be unable to decrypt the message and return an error.

Thus, no matter how the untrusted host misbehaves in regards to the session ID, any misbehavior will be detected by the client as the TEE will not respond to his request at all or will respond with an error.

```
public TEE_STATUS ecall_ke_process_msg_1(
    [in, string] const char* input,
    [out] void* output,
    [out] size_t* output_size,
    [out] session_id_t* id);

public TEE_STATUS ecall_ke_process_msg_2(
    session_id_t id,
    [in, string] const char* input);

public TEE_STATUS ecall_handle_encrypted_message(
    session_id_t id,
    [in] void* msg,
    [out] void* output,
    [out] size_t* output_size);
```

**Listing 6.1:** ECALLs of the SGX implementation. Abbreviated notation of the Intel SGX EDL language, all pointer sizes are omitted from this listing.

The quote that is sent as a response to the first ECALL *ecall_ke_process_msg_1* contains the public ephemeral Diffie-Hellman key of the enclave and the public signing key of the client that was sent in the initial message. As both keys have the size of 256 bit, they can both be placed in the available 512 bit of the report data structure that the SGX quote provides and that can be filled with user data. This serves the combined purpose of communicating and authenticating the user signing key and the enclave public key.

## 6.3 TEE$^2$ Python Client

We implement the client in Python 3 and utilize different cryptographic libraries to perform each step of the key exchange and encrypted messages. For elliptic curve operations, we utilize the Rubenesque library[4] and use it for Diffie-Hellman and DSA operations during key exchange. Once the session key is established, we utilize Python Cryptography[5] to perform AES-GCM operations to encrypt and decrypt messages that are passed to the enclave. For all other operations such as hashing, json, or base64 encodings, we utilize standard Python libraries. TCP connections are implemented with Python TCP sockets and message sending and receiving is parallellized with standard Python threads.
Listing 6.2 shows an example of a request called *TEE2_COMMAND_RANDOM*. The shown message requests to generate 10 random bytes and return them to the client. This is the implementation of protocol $\mathfrak{R}$ and is assigned the internal command code 33 that is known by both the client and the TEE. With the general *args* value that contains a JSON with the function specific *byte_count*, arbitrary functions can be realized as the command code is detached from the actual data that is sent and data can be handled separately by each function implementation. Listing 6.3 shows the response to such a request with code 32 which is called *TEE2_COMMAND_SUCCESS* and denotes a successful execution of the command. The result is then a simple string that encodes the random bytes as base64.

```json
{
    "args":
    {
        "byte_count": 10
    },
    "command": 33
}
```

**Listing 6.2:** JSON of command request sent by the client.

```json
{
    "code": 32,
    "result": "o0xg6b0t7sNavg=="
}
```

**Listing 6.3:** JSON of command response sent by the TEE.

[4] https://github.com/latchset/python-rubenesque
[5] https://cryptography.io

## 7 Security Evaluation

In order to evaluate the security of our work, we model a subset of the Ideal and Combined TEE protocols with the Tamarin tool[1] [53] [65]. We furthermore discuss modeling the rest of the protocols in Section 7.3 and give security arguments for the unmodeled protocols.
The Tamarin Prover is a symbolic model checker and can be used to analyze security protocols. In a Tamarin model, protocols are typically split up in actions such as "sending a message" or "receiving a message and sending a reply". These actions are then modeled as a *rule* in Tamarin where all rules of a model are forming a labelled transition system. Security properties are specified as trace properties which are checked against the traces of the transition system[2]. For an unbounded number of participants and protocol runs, Tamarin can automatically construct a proof based on the given security properties using a multiset to model the adversary's knowledge. Tamarin provides built-in modules for ideal cryptographic functions and Dolev-Yao style network adversaries and can simulate Diffie-Hellman, hashing, and symmetric as well as public key operations.
We model key exchange and random number generation in the presence of $\mathcal{M}_W$ for both the Ideal TEE and the Combined TEE. Both the models for Ideal and Combined TEE contain security related lemmas for each protocol that are verified by the Tamarin tool and are proven to be correct with the given adversary model. While we model the two core protocols key exchange and random number generation with a weak adversary, we also discuss which other protocols can be modeled formally and how one would do so.

### Modeling TEEs

Due to the unique properties required by TEE attestation, we model private TEE keys used for attestation as the output of a *private function* named *quote_key* in the Tamarin tool. A private function can not be used by the Tamarin adversary but can only be used explicitly as part of a rule stated in the model. We use this limitation of the adversary to model a publicly verifiable TEE attestation key that can not have been generated by the adversary. Instead, all instances of a TEE in our Tamarin models can be used as an input into the *quote_key* function which deterministically returns the private key of this TEE. This private key can again be used as an input into the default Tamarin function *pk* which returns a public key based on an input. All rules make an equality check to verify that the TEE instances they use are attached to a proper signing key that belongs to the stated TEE instead of being controlled by the adversary. This formalization is different to Pass et al. [61] as discussed in Section 9. In the Combined TEE model, we then model TEE compromise with specific TEE and session key reveal rules. While it would also be possible to use a standard signing key bound to a TEE that can only be registered once, our approach simulates that the keys used for TEE attestation are not arbitrary keys but are actually verified by a trusted party (i.e. the manufacturer) and can be linked to one specific TEE instance.
The first rule in all models is called *register_tee_key* and maps any name via the *quote_key* function to a public key pair so that this TEE can be used internally by Tamarin. While we restrict the use of this rule to one time per TEE name, this has no actual influence on the use of the TEE keys by the adversary as the rule always behaves deterministically based on the given TEE name. Instead, the restriction to one time per TEE name is solely due to performance in order to force the Tamarin tool to not execute this rule more than once.

## 7.1 Ideal TEE model

The full Tamarin model of the Ideal TEE can be seen in Appendix B.1 and consists of 8 Tamarin rules and 7 lemmas. While the first rule is used as a setup and registers a TEE in the Tamarin model, four rules are needed for key exchange and three rules for random number generation. Of the 7 lemmas, one is a source lemma required for the Tamarin-internal

[1] https://tamarin-prover.github.io
[2] https://tamarin-prover.github.io/manual/book/001_introduction.html

restriction of the adversary, three are used for functional tests if the model can behave correctly, and three are security lemmas that verify the security properties guaranteed by each protocol. In addition to the private *quote_key* function and its associated *register_tee_key* rule mentioned above, the Ideal TEE model requires no adjustments to our adversary model and setting.

## 7.1.1 Key exchange

The complete key exchange as depicted in Figure 5.1 is modeled with the rules *ke_message_1* to *ke_message_3* and one final message *ke_message_3_verify* that is used by the TEE to complete the protocol after receiving the third message. Thus, each rule represents one message in the given protocol with the last rule representing the work on the side of the TEE to perform the last nonce checks and calculate the session key. Quote checks are modeled by the user enforcing the equality of received TEE quotes with expected quotes based on known content and the publicly known key that is mapped to this specific TEE.
To formally verify the security of the key exchange protocol, we use the security notion of Lowe [49]. Since our protocol already uses nonces on each side of the protocol to ensure freshness, we verify the two security properties key agreement and key secrecy. Both security lemmas verify that the defined properties hold for all possible traces of the protocol and are specified as *all-traces* lemmas in the Tamarin syntax.

#### Key agreement

The first security lemma verifies that all runs of the protocol where a user supposedly establishes a key with a TEE and a TEE supposedly establishes a key with the user result in both parties agreeing on the same key. This ensures that the adversary cannot perform a Man-in-the-Middle attack where he places himself in the middle of the communication and establishes a key with both parties while pretending to be each others correct counterpart. Listing 7.1 depicts the lemma *ke_correct_agreement* that encodes this property. It specifies that for all users and TEEs that supposedly establish a key, the implication is that these keys must be identical. This lemma uses the actionfacts *User_Established* and *TEE_Established* that are only generated by Tamarin during the *ke_message_3* and *ke_message_3_verify* rules.

```
lemma ke_correct_agreement: all-traces
"
// For all users and TEEs that established a session
All U_pk TEE k1 k2 #i #j .
    (  User_Established(U_pk, TEE, k1) @#i
    &  TEE_Established(TEE, U_pk, k2) @#j
    )
    ==>
    // They agreed on the same key
    k1 = k2
"
```

**Listing 7.1:** Key agreement lemma of the Ideal TEE model. All sessions between $\mathcal{U}$ and Ideal TEE agreed on the same session key.

#### Key secrecy

The second security lemma verifies that for any key that is agreed upon by $\mathcal{U}$ and Ideal TEE, the adversary does not know this key, modelled in Tamarin syntax with the $K(x)$ notation. Listing 7.2 depicts this lemma. It states that again, for all users that established a key with a TEE and for all TEEs that established a key with a user, the implication is that the adversary does not know the key they agreed upon. In combination with the lemma above, the adversary can neither influence nor obtain the key resulting from the given key exchange protocol.

```
lemma ke_secure: all-traces
"
All U_pk TEE k #i #j .
    // For all users and TEEs that established a session
    (  User_Established(U_pk, TEE, k) @#i
    &  TEE_Established(TEE, U_pk, k) @#j
    )
    ==>
    // The adversary does not know the session key
    (not (Ex #l . K(k) @l ))
"
```

**Listing 7.2:** Key secrecy lemma of the Ideal TEE model. All sessions that have been established between $\mathfrak{U}$ and Ideal TEE have a key that is unknown by the adversary.

## 7.1.2 Random number generation

Random number generation with the Ideal TEE follows the protocol $\mathbb{R}$ depicted in Figure A.3. However, we do not model any remote party that authenticates the random number in Tamarin as the trust properties required by this party go beyond the scope of the Tamarin tool. Instead, we only model the random number generation and let the user verify the correctness of the received TEE signature herself. While this does not model a user colluding with the adversary, we can still reason about the secrecy of the generated random number itself.

In the model, random number generation is realized with the three rules *rng_request*, *rng_response*, and *rng_complete*. While the first two rules model each message that is sent from $\mathfrak{U}$ to Ideal TEE and vice versa, the last rule models the work that $\mathfrak{U}$ has to perform after receiving the response from the Ideal TEE. Specifically, $\mathfrak{U}$ has to verify that the random number was signed by the TEE (which models a verification that a remote party would perform) and has to check the returned nonce for its correctness. The result of the last rule is then a generated random number with the Ideal TEE.

To formally verify this protocol with Tamarin, we define the *rng_secure* lemma as depicted in Listing 7.3 which is again defined as an *all-traces* lemma in the Tamarin tool. The lemma states that for all users that established a key with a TEE and have generated a random number $r$ with that TEE, this implies that $r$ is not known by the adversary. Implicitly, this also states that $r$ is not influenced by the adversary.

```
lemma rng_secure: all-traces
"
    All U_pk TEE r sessionkey #i #j #k .
    // For all users that generated a random number with a TEE
    ( User_Established(U_pk, TEE, sessionkey) @ #i
    & TEE_Established(TEE, U_pk, sessionkey) @ #j
    & RandomGenerated(TEE, U_pk, r) @ #k
    )
    ==>
    // The adversary does not know this number
    (not (Ex #l . K(r) @#l ) )
"
```

**Listing 7.3:** Random number generation secrecy lemma of the Ideal TEE model. The adversary does not know the generated random number if it is the result of a session between $\mathfrak{U}$ and an Ideal TEE and if it has been generated correctly.

## 7.2 Combined TEE models

We model the Combined TEE protocols for key exchange and random number generation in order to formally verify their security and compare them with their Ideal TEE equivalent. Due to the combined complexity of key exchange and random number generation with multiple involved TEEs, we create separate Tamarin models for each protocol and simulate a secure key exchange in the beginning of the random number generation as a substitute for performing the actual key exchange. The model for key exchange can be seen in Appendix B.2 while the model for random number generation is given in Appendix B.3

In contrast to the adversary of the Ideal TEE, the Combined TEE exists in the presence of the adversary $\mathcal{M}$ with his two variations $\mathcal{M}_W$ and $\mathcal{M}_S$. To model this, we create the rule *reveal_tee_key* to reveal the TEE long term key that is the result of the *quote_key* function to the attacker. This models the strong adversary $\mathcal{M}_S$ that is able to fully break a TEE by retrieving its internal keys which allows $\mathcal{M}_S$ to simulate it completely. Furthermore, we model a weak attacker $\mathcal{M}_W$ in the random number generation model by additionally adding a *reveal_session_key* rule that reveals the session key between $\mathcal{U}$ and $\mathcal{T}_M$ to the attacker. Note that in our protocols, revealing the session key is sufficient to model the weak adversary's ability to read all data in the TEE's memory, since our protocols do not use any additional asymmetric key operations or one-way functions in the TEE. We do not add this rule to the key exchange Tamarin model as this model never uses the session key after establishing it, making a session key reveal unnecessary. Both added rules trigger the Tamarin action facts called *REVEAL_HAPPENED* and *TEE_REVEALED($tee)* that can be used by all Combined TEE security lemmas to check if any key reveal has happened or if one specific TEE $tee$ has been broken (either through revealing its session key or its long term key).

### 7.2.1 Key exchange model

The Tamarin model for key exchange with the Combined TEE models the protocol shown in Figure 5.2. It consists of 6 rules and 4 lemmas of which two rules are used for the TEE setup and key reveal while two lemmas are used for functional tests of the model. The four rules required for key exchange are based on the same rules of the Ideal TEE model explained above where three rules are used for each type of message that is sent and one last rule is used by the TEEs to verify the final user input and complete the protocol. In contrast to the Ideal TEE protocol, the user's rules in this model always create two outputs, one for each TEE, and perform an additional check that the TEEs are not identical. All TEE rules then have to be executed twice by the tool, once for each TEE. The output on the user's side is a combined session key that consists of two individual session keys with the two TEEs.

To formally verify key exchange with the Combined TEE, we use the same security properties as for the Ideal TEE model, namely key agreement and key secrecy.

#### Key agreement

Listing 7.4 shows the lemma *ke_correct_agreement* which verifies that $\mathcal{U}$ and $\mathcal{T}_i$ agreed on the same pairwise keys. However, in our security model we allow $\mathcal{M}_S$ to compromise the long term TEE key through the use of the *reveal_tee_key* rule. This allows $\mathcal{M}_S$ to influence the keys or perform a Man-in-the-Middle attack which would result in non-matching keys between a user and a compromised TEE $\mathcal{T}_M$. To accommodate this adversary, we state in the lemma that for all events where a user supposedly established a key with two TEEs and these two TEEs supposedly established a key with the user, the implication is that *either* they pairwise agreed on the same key *or* this TEE has been compromised.

#### Key secrecy

Listing 7.5 depicts lemma *ke_secure* of the Combined TEE model. Similar to the same lemma of the Ideal TEE model, the lemma aims to verify that the key that is the result of a successful key exchange is not known by the adversary. However, in the presence of the strong adversary $\mathcal{M}_S$ we can not guarantee this restriction in all cases. Instead, we can only guarantee this restriction for the uncompromised TEE $\mathcal{T}_S$. To model this, the lemma states that for all sessions between

```
lemma ke_correct_agreement: all-traces
"
(All U_pk TEE1 TEE2 user_k1 user_k2 tee1_k tee2_k #i #j #k.
    // Whenever U establishes a session key with TEE1 and TEE2...
    User_Established(U_pk, TEE1, user_k1, TEE2, user_k2) @ #i
    & TEE_Established(TEE1, U_pk, tee1_k) @ #j
    & TEE_Established(TEE2, U_pk, tee2_k) @ #k
    ==>
    (
        ( // ... U and TEE1 have the same key if it was not compromised before the KE
        user_k1 = tee1_k
        | (Ex #l . TEE_KEY_REVEAL(TEE1) @ #l & l < j)
        )
        & // AND
        ( // ... U and TEE2 have the same key if it was not compromised before the KE
        user_k2 = tee2_k
        | (Ex #l . TEE_KEY_REVEAL(TEE2) @ #l & l < k)
        )
        )
// Since this is an all-traces check, the key agreement is only ensured with non compromised TEEs.
)
"
```

**Listing 7.4:** Key agreement lemma of the Combined TEE model. Whenever $\mathcal{U}$ establishes a session key with TEE1 and TEE2, they pairwise agreed on the same key as long as the TEE has not been compromised.

$\mathcal{U}$ and the two TEEs $\mathcal{T}_1$ and $\mathcal{T}_2$, the implication is that the adversary either does not know least one of the session keys, or he has compromised both TEEs.

## 7.2.2 Random number generation model

Appendix B.3 shows the full Tamarin model of the protocol $\mathfrak{R}$ depicted in Figure 5.5. The model consists of 8 rules and 3 lemmas of which 5 rules are necessary for the setup and key reveals and two lemmas are necessary for Tamarin internal restrictions and functionality checks. In addition to the *register_tee_key* rule that is the same in all previously mentioned models, this model contains two separate key reveal rules. The first key reveal rule *reveal_tee_key* outputs the long term TEE key to the adversary and is the same key reveal that we used in the key exchange model of the Combined TEE. The second key reveal rule however outputs the session key between a user and a TEE to the adversary. This models a weak attacker $\mathcal{M}_W$ who can not fake TEE attestation but can only read secrets maintained inside the TEE such as session keys. In combination, both key reveal rules model the strong and weak adversary at the same time. It is important to note that we model the random number generation similar to the same model of the Ideal TEE. As such, there is no remote party that verifies the random number but instead the user verifies both TEEs herself and ensures the authenticity of the random number. While this does not prevent attacks where $\mathcal{U}$ colludes with $\mathcal{M}_S$ to choose a random number at her will, this model can still be used to verify that the resulting random number is secret from the adversary and any further use of the protocol has to be evaluated qualitatively outside of the Tamarin model. The result is that from within this Tamarin model we expect the weak and strong attacker to have the same limitation with the given key reveal rules as neither will be able to view both parts of a generated random number and no attacker will be able to retrieve the final output of $\mathfrak{R}$. Since this model is split from the key exchange model described above, we introduce a rule *establish_session* that simulates such key exchange. For this, it creates a new user identity and outputs a new session with new session keys to two given TEEs. The output of this function is the same that is achieved by completing the key exchange protocol in the other Tamarin model. Based on this simulated key exchange, the user can start to query the two TEEs for randomness. The process of this is modeled in three rules that have similar functions as their counterpart in the Ideal TEE model.

```
lemma ke_secure: all-traces
"
All U_pk TEE1 TEE2 k1 k2 #i #j #k .
    // For all users and TEEs that established a key
    (  User_Established(U_pk, TEE1, k1, TEE2, k2) @ #i
    &  TEE_Established(TEE1, U_pk, k1) @ #j
    &  TEE_Established(TEE2, U_pk, k2) @ #k
    )
    ==>
    (
    // The adversary does not know AT LEAST ONE key, or TWO key reveals happened
    not (Ex #j . K(k1) @ #j)
    | not (Ex #k . K(k2) @ #k)
    | (Ex #o #p . TEE_KEY_REVEAL(TEE1) @ #o & TEE_KEY_REVEAL(TEE2) @ #p)
    )
"
```

**Listing 7.5:** Key security lemma of Combined TEE. Whenever $\mathcal{U}$ establishes a session key with TEE1 and TEE2, either key is unknown by the adversary or the adversary has compromised both TEEs.

*rng_request* queries the two TEEs from a generated session for a random number and assigns each of them a nonce. *rng_response* is executed by each TEE and receives a request and responds with a random number share. The final rule *rng_complete* varies more from the same rule in the Ideal TEE model as it not only has to verify each nonce, TEE key, and quote but also has to verify that the two TEEs are not identical. It also combines the two received random numbers into the resulting output. In this Tamarin model, we deviate from protocol $\mathfrak{R}$ by using a hash function on the client to combine the two random number shares instead of using an exclusive or (XOR). While the Tamarin tool supports the use of XOR since an extension by Dreier et al. [20], the underlying XOR representation is still not sufficient to successfully model our use of the randomness that allows the adversary to obtain half of the input of the random number. To prevent difficulties with the adversary's freedom by using a hash function (e.g. by preventing him from ever obtaining the random number with the given constraint systems if he can not combine the two random number parts), we add a special rule *combine_randoms* that simply combines two inputs into a combined random number by using the hash function. While the adversary can use the hash function himself, this specific rule allows him to explicitly combine the random number after obtaining both parts and ensures that we do not restrict him.
To formally verify that the adversary is not aware of the combined random number $r$ after a successful run of the protocol as long as he has not compromised both TEEs, we define lemma *rng_secure* depicted in Listing 7.6. The lemma states that for all users that have established a session with two TEEs and have completed a protocol run of the random number generation, the implication is that either the adversary does not know the generated number or he has compromised both TEEs. We explicitly do not limit this lemma to protocol runs where the TEEs have completed the protocol or are even aware of a session key. This gives the attacker more freedom in choosing his attack and limits our view of the security of the generated random number to the user's perspective.

## 7.3 Other protocol models

We only model a subset of the protocols defined in Section 5. While it is possible to model several of the remaining protocols with the Tamarin tool, this is not the case for all of our discussed protocols. However, note that we presented security arguments in the respective section of each protocol.
$\mathfrak{R}_{\mathcal{M}_S}$: Random number generation with a strong adversary can be modeled analog to $\mathfrak{R}$ and can utilize the same key reveal rules.
**$\mathfrak{G}$, $\mathfrak{G}_{\mathcal{M}_S}$, and $\mathfrak{D}$:** ElGamal key generation can theoretically be modeled similarly to random number generation. However, Tamarin neither supports modeling elliptic curve operations nor does it specifically support ElGamal cryptographic oper-

```
lemma rng_secure: all-traces
"
    All U_pk TEE1 TEE2 r sessionkey1 sessionkey2 #i #j .
    // For all established sessions with 2 TEEs and a generated random number,
    ( User_Established(U_pk, TEE1, sessionkey1, TEE2, sessionkey2) @ #i
    & RandomGenerated(TEE1, TEE2, U_pk, r) @ #j
    )
    ==>
    // The generated random number is secret ...
    ( not (Ex #k . K(r) @#k )
        // ... or BOTH TEE keys were revealed
    | (Ex #m #n . TEE_REVEALED(TEE1) @ #m & TEE_REVEALED(TEE2) @ #n )
    )
"
```

**Listing 7.6:** Random number generation secrecy lemma of the Combined TEE model. The adversary does not know the generated random number as long as he has not compromised both TEEs.

ations. While modeling elliptic curves can be overcome by modeling a variation of the mentioned protocols by utilizing the Diffie-Hellman Tamarin module, this might not suffice to model ElGamal operations and further work has to be done to model ElGamal with Tamarin.

**$\mathfrak{S}$:** Due to the built-in *signing* module, it is possible to model protocol $\mathfrak{S}$ for a generic, unspecified, signature algorithm. The model can be built similar to our model of random number generation for a weak adversary where the content of the sent messages is a signature instead of a random number.

**$\mathfrak{SF}$:** The presented store and forward protocol utilizes XOR operations to perform secret sharing between the two TEEs. This can be modeled with Tamarin's XOR extension [20].

**$\mathfrak{OT}_n^m$:** Oblivious transfer has two phases that could be modeled separately. The offline phase relies on Goldwasser-Micali cryptographic operations which are currently not supported by Tamarin. The online phase however only relies on XOR once the key vector is established and could be modeled with Tamarin.

# 8 Performance Evaluation

We evaluate the performance of our implementation in two ways. First, we vary the length of the random number requested in the $\mathfrak{R}$ and $\mathfrak{R}_{\mathcal{M}_S}$ protocols, whilst keeping the number of TEEs constant. This benchmark indicates how TEE$^2$ performs as the size of the protocols' inputs, outputs and intermediate messages increases. Second, we keep the input size constant and vary the number of TEEs that are participating in the Combined TEE. In general, increasing the number of TEEs in the Combined TEE increases security, but also inherently decreases performance. This type of benchmark can be used to reach an acceptable balance between security and performance. In both evaluations, the Combined TEE consists of two or more SGX enclaves, as implementing and benchmarking other types of TEEs is future work. All evaluations were run on a single machine with an Intel i5-6550 processor and 8GB RAM. Note that the used processor has four cores and no Hyper-Threading Technology. While evaluating TEE$^N$ on one single physical machine eliminates network latency, it also simulates a worst case scenario as the machine can not always execute all involved TEEs in parallel.

## 8.1 Evaluation methodology

All evaluations are performed by repeatedly executing the Python client described in Section 6.3 in an evaluation mode. In this evaluation mode, the client first performs a key exchange to all TEEs that are involved in this evaluation run. Next, the client executes protocol $\mathfrak{R}$ and follows this with protocol $\mathfrak{R}_{\mathcal{M}_S}$. Afterwards, the client closes the connection to each TEE and logs the measured run time. Before and after the execution of each step, i.e. before and after key exchange, simple, and complex randomness, the client measures the current system time. As such the measured duration is the overall latency that the Combined TEE requires to perform the given request including the time taken by the client to process the responses. This is important to note, as both randomness protocols require the client to combine the received random numbers an in the case of $\mathfrak{R}_{\mathcal{M}_S}$ the client has to perform additional tasks such as additionally hashing the commitments and checking all received messages for their correct commitments and quotes. All TEEs are contacted in parallel for which the client utilizes Python threads (using the `multiprocessing.pool.ThreadPool` package). While the SGX enclaves are being executed on the same physical machine as separate processes, utilizing Python threads ensures that the TEEs can be contacted in parallel instead of in succession.

## 8.2 Increasing the size of the queried random string

We first evaluate how our system scales in regards to the queried random string from TEE$^N$. For this, we use a fixed TEE$^2$ system and increase the amount of bytes requested from the protocols $\mathfrak{R}$ and $\mathfrak{R}_{\mathcal{M}_S}$ from 1,000 to 10,000 bytes in steps of 1,000 bytes. Figure 8.1 shows this evaluation. Each data point is an average of 100 runs and the error bars are ranging from the 1% percentile to the 99% percentile. As such, 98% of the data falls within the two error bars. $\mathfrak{R}$ is depicted as *simple randomness* while $\mathfrak{R}_{\mathcal{M}_S}$ is depicted as *complex randomness*. The graph shows a very slight linear increase for both $\mathfrak{R}$ and $\mathfrak{R}_{\mathcal{M}_S}$ which is caused by larger intermediate messages that are sent during the protocol and a larger amount of random data that is generated, hashed, and encrypted.

## 8.3 Increasing the number of TEEs involved in TEE$^N$

In addition to scaling the number of requested bytes from the TEE$^N$ random number generation protocols, we also evaluate our implementation with an increasing number of TEEs participating in TEE$^N$. To do this, we first perform a key exchange with all TEEs and then run the protocols $\mathfrak{R}$ and $\mathfrak{R}_{\mathcal{M}_S}$ with 10,000 requested random bytes. We repeat this process 100 times per number of evaluated TEEs. Figure 8.2 depicts the evaluation of one to ten TEEs involved in TEE$^N$

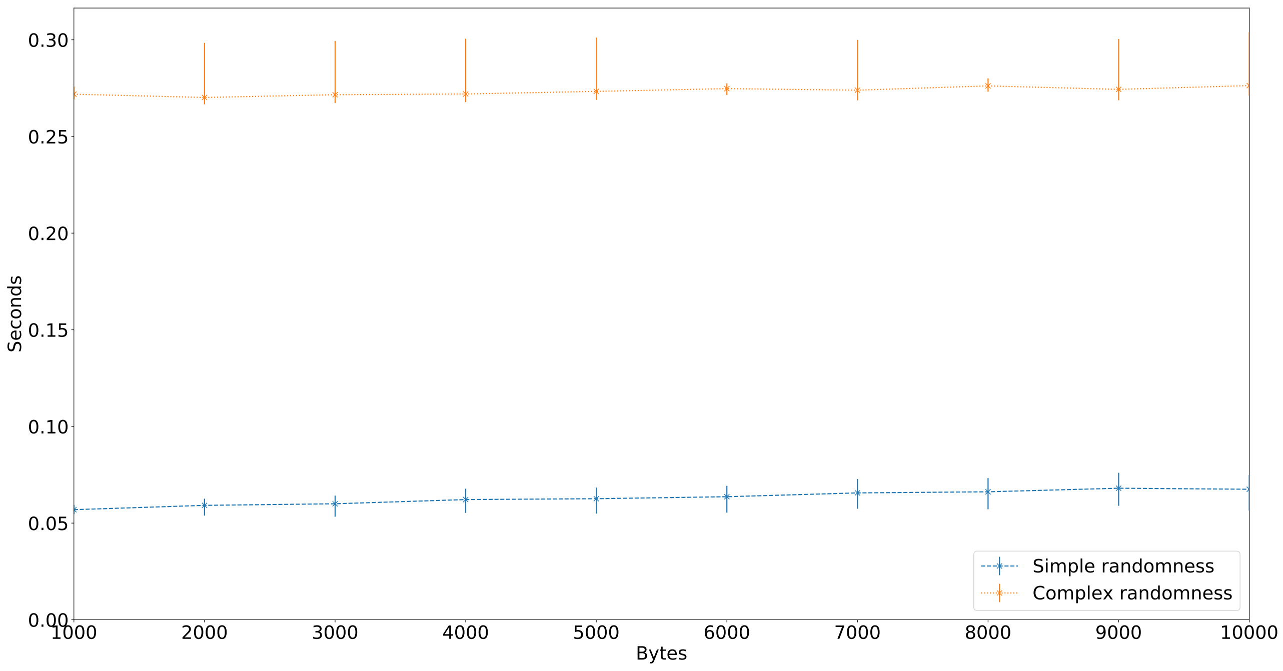


**Figure 8.1.:** Evaluation of an increase in the input parameter to random number generation for TEE$^2$. The error bars depict the range from the 1% percentile to the 99% percentile.

and shows the average time to complete the protocol. The error bars range from the 1% percentile to the 99% percentile. $\mathfrak{R}$ is again depicted as *simple randomness* while $\mathfrak{R}_{\mathcal{M}_S}$ is depicted as *complex randomness*. The graph shows a linear scaling for all involved protocols. This linear increase is consistent when comparing TEE$^N$ to the case where only a single TEE is used. This means that TEE$^N$ does not suffer a larger than linear increase in performance in comparison to the trivial use of a single TEE.

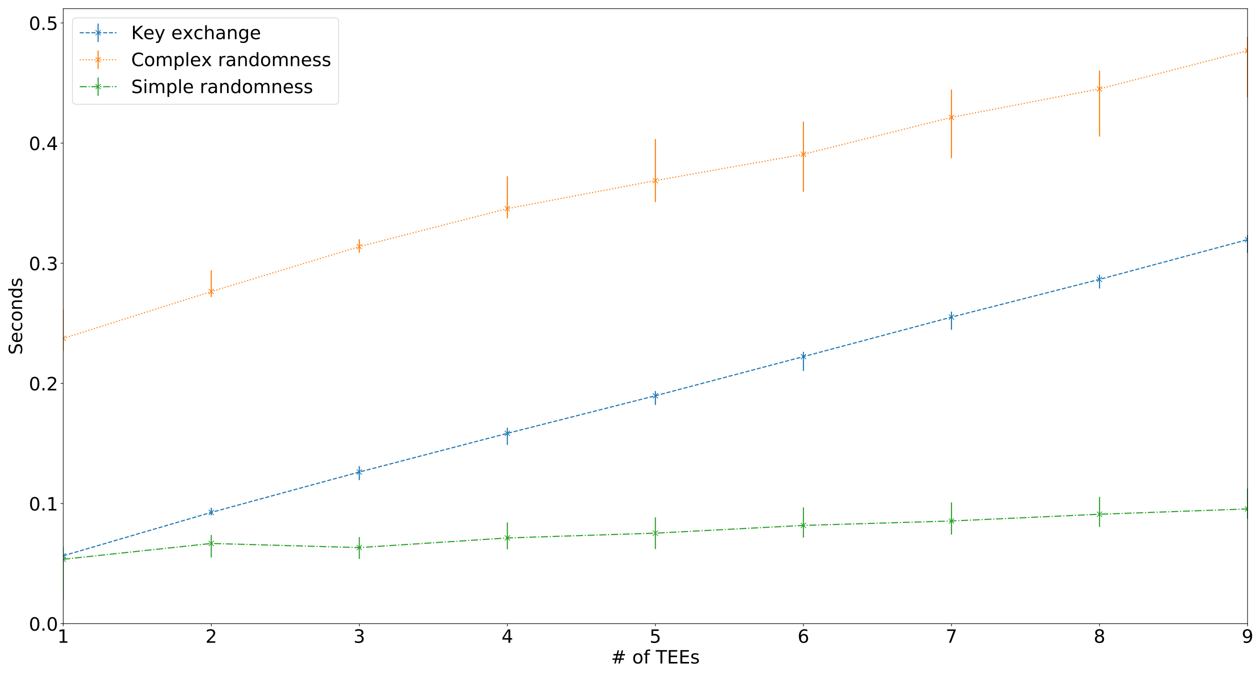


**Figure 8.2.:** Evaluation of an increase in the numer of TEEs that are involved in TEE$^N$. The error bars depict the range from the 1% percentile to the 99% percentile.

# 9 Related Work

We discuss three types of related work: Related work in regards to our adversary model that tolerates up to $N-1$ compromised participants, related work on combining multiple TEEs or trusted hardware components, and finally related work on modelling trusted hardware protocols.

## 9.1 Adversary model

Our design of a Combined TEE allows for $N-1$ compromised TEEs in a TEE$^N$ system. This concept of a majority of compromised parties is not new, but difficult to achieve without leveraging trusted parties.

DiSe is an independently developed system by Agrawal et al. [1] which allows its users to perform symmetric key encryptions in a distributed manner. Built as a threshold symmetric-key encryption system, it allows for $N-1$ users to be malicious. DiSe bases on distributed pseudo-random functions introduced by Micali [55] to generate a string that is used as input into a pseudo random generator. The output of this pseudo random generator is then combined per XOR with the message. An additional random number appended to the message serves as an authenticating tag of the decrypted message. The use case of DiSe is equivalent to the store and forward scheme discussed in Section 5.3.1 and can be realized with protocol $\mathfrak{S}\mathfrak{F}$ by sharing the symmetric encryption keys with other users.

In regards to general multiparty computation, Franklin et al. [25] discussed how to perform fair exchange with a semi-trusted third party. Fischlin et al. [23] discussed secure set intersection in the context of non-programmable hardware tokens and also considered side-channel attacks that compromise one or several of these hardware tokens. Naor et al. [57] discussed oblivious transfer in a distributed setting and Gang et al. [26] described a solution to oblivious transfer where a client uses proxies to query the information.

## 9.2 Combining TEEs

To the best of our knowledge, we are the first to combine TEEs for the specific purpose of resilience against compromises. However, there exists prior work that developed such hardware combiners for other types of secure or trustworthy elements. The most notable of this prior research was performed by Mavroudis et al. [51] who proposed Myst, a system that combines a quorum of possibly untrusted cryptocoprocessors. Myst is targeted at Integrated Circuits (ICs) and aims to alleviate the impact of fabrication errors or hardware trojans in customer off-the-shelf hardware. Its system setup includes more than one hundred ICs sharing one IC-controller that is controlled by a remote host. While Myst ensures the integrity and confidentiality of messages between IC-controller and remote host, it does not attest the ICs to the host. This is a result of the used hardware that only features a secure random number generator and enough persistent memory to store keys, but does not provide the capability of remote attestation like a fully-fledged TEE. Myst provides multiple operations such as distributed random number generation, a distributed ElGamal crypto system, and distributed multi-signatures based on Schnorr signatures. According to the authors, Myst can tolerate up to 100% of compromised components if not all components are compromised by the same adversary [51]. This also holds true for our Combined TEE as any adversary has to have knowledge of all participating TEEs in order to break the security, no matter how many other adversaries might be compromising the system at the same time.

In the context of TEEs, there exists prior research that combines TEEs for various purposes and touches on the same properties that we aim to achieve for the Combined TEE. Matetic et al. [50] presented ROTE which prevents rollback attacks against SGX enclaves by maintaining a distributed counter with $f+2u+1$ enclaves where $f$ is the number of failed and $u$ is the number of unavailable enclaves.

Ankele et al. [5, 4] discussed how the Trustworthy Remote Entity introduced by Paverd [62] can be used for multiparty computation and also discussed the performance that could be expected of such a system. Similarly, Bahmani et al. [7]

discussed multiparty computation in the context of Intel SGX and Gupta et al. [31] discussed secure function evaluation with Intel SGX.
There are several open-source and industry projects that are aimed at simplyfing the developmet of TEEs. Most of these projects are targeted at multiple TEE types simultaneously and allow developers to deploy their written program to more than one TEE at the same time. Such a principle can be used for $TEE^2$ since a common interface is essential in developing $TEE^2$ applications. Notable projects in this category are Google Asylo[1], Microsoft Open Enclave[2], the Keystone project[3], and Baidu MesaTEE[4]. Furthermore, GlobalPlatform also published several whitepapers aimed at standardizing TEEs and TEE development [28, 29].

**Cryptographic combiners**

Our approach of combining multiple TEEs is similar to the concept of cryptographic or robust combiners. This concept was first introduced by Harnik et al. [33] and aims to combine multiple cryptographic primitives in order to increase security in the event of a compromise of one primitive. The straightforward robust combiner is a *Cascade* combiner that simply cascades multiple encryptions into each other, effectively forcing the adversary to break all layers in order to regain the plaintext. Based on this initial work, Herzberg described different types of combiners [34] and evaluated the tolerance of such combiners in regards to attacks [35]. Others used robust combiners for a multitude of MPC operations such as Fischlin et al. [22] who developed obfuscation combiners or Harnik et al. [33] who discussed robust combiners for oblivious transfer.

## 9.3 Modelling trusted hardware

There exists some prior work using formal methods to verify protocols that make use of trusted hardware components. Pass et al. [61] discuss the formalization of attested execution and assume that all quotes are signed by the manufacturer of the hardware. This is slightly different from our approach but can be simulated with our model by modelling a certifation of the quote keys by the manufacturer. However, their approach does not allow to model TEE compromise. Brzuska et al. [14] discussed the formalization of physically uncloneable functions. Basin et al. [8] formalized different types of compromise and discussed their impact on protocol analysis. Their types of compromise are close to our compromise rules described in Section 7. Cohn-Gordon et al. [17] study post-compromise security and discuss weak and full compromise scenarios and their formalization.

[1] https://asylo.dev/
[2] https://openenclave.io/sdk/
[3] https://keystone-enclave.org/
[4] https://www.mesatee.org/

# 10 Discussion & Conclusion

## 10.1 Discussion

Based on this research, we present the following topcics of discussion.

**Adversary considerations**

It is interesting to note that while the two discussed adversary models $\mathcal{M}_W$ and $\mathcal{M}_S$ differ greatly in their capabilities, the resulting protocols do not differ as much as one might expect. Taking the protocols $\mathfrak{X}$ and $\mathfrak{X}_{\mathcal{M}_S}$ as an example, their complexity only differs in one additional message round to all involved TEEs (Figures 5.5 and 5.6). While one additional message round might already be resource intensive, it can be argued as being a reasonable trade-off if one needs to protect themselves against a strong, TEE compromising adversary. At the same time, it might even be reasonable for a party that only considers $\mathcal{M}_W$ to implement a protocol that grants security in the presence of $\mathcal{M}_S$. As the stronger protocols provide security even if all but one involved TEE is malicious, they can bring a higher degree of trust to users than protocols that rely on more security guarantees of the involved TEEs.

**Deployment considerations**

As previously mentioned, there exists earlier work and projects that aim to streamline the development of trusted applications on TEEs. Such frameworks can also be utilized for the develompment of TEE$^2$ applications and protocols. While most of these frameworks are not targeted at the interaction between different types of TEEs, they can certainly be adjusted for this use case and common APIs can be established. Such common APIs and standards can be developed similarly to the GlobalPlatform TEE specifications and could be streamlined in a similar fashion as the Open TEE project. Besides Intel SGX that we used for our prototype implementation, ARM TrustZone should also be considered as a second vendor for a TEE$^N$ enabled Combined TEE due to its wide deployment in smartphones and mobile hardware. Such deployment scenario also creates challenging new opportunities due to the difference in performance of mobile devices and desktop CPU based TEEs like Intel SGX. While most cryptographic tasks discussed in this thesis can be realized on hardware with lower performance, other tasks such as MPC or full disk encryption might be more challenging with a large performance difference of the involved TEEs in TEE$^N$.

**Other TEE$^N$ scenarios**

Besides the presented use case of TEE$^N$ where all TEEs are equal parts of the Combined TEE, there are also other scenarios where combining several TEEs can be beneficial. We discuss four of such possible scenarios.

**Combining TEEs of different performance and scale:** One scenario is the challenge of full disk encryption where a small trusted entity holds the encryption keys and passes them to the entity that performs the disk decryption. Usually, such small entity would effectively be a smart card while the larger trusted entity could be an ARM TrustZone-based TEE. In practice, the decrypting entity can be a TEE that is trusted enough to perform the encryption and decryption operations on the drive but that is not trusted enough to protect the encryption keys on its own. At the same time, the small trusted entity does not have the capabilities for large scale I/O operations and can not be used to decrypt the whole disk. With the classic approach, the small trusted party hands off the keys to the larger TEE when the device boots and is effectively only protecting the encryption keys while the device is shut down. The TEE then uses the encryption keys to decrypt the data on the disk. With a Combined TEE approach however, a solution could be found where the smaller device cooperates with the larger device and is continually involved in the decryption process of the disk. This would prevent the risk of a compromised TEE that can leak the decryption keys as the smaller entity would at all times hold at least a part of the necessary secret for all encrypted data.

**Combining TEEs for safety and reliability:** Besides possible attack vectors that undermine the security of one of the TEEs involved in a Combined TEE, there might also be concerns regarding safety and reliability that overshadow security concerns in a specific scenario. Consider a modern car with multiple cooperating TEEs inside that communicate with each other over an internal Controller Area Network (CAN) and that set up secure channels with each other and agree on encryption keys for data similar to secret sharing. In regards to security, such a setup might be very secure as each TEE is involved in cryptographic operations and tampering with any TEE will be immediately evident and can be acted upon. At the same time however, since all TEEs are involved in the security of the underlying system, such system also needs to account for unexpected behavior in the setup of the Combined TEE. Such unexpected scenario could be a failure of an involved TEE that then gets swapped out by a certified mechanic, or even a scheduled update of a module with new software or even new hardware. Resilience to failures and mechanical updates become of the utmost importance in such a scenario. Any Combined TEE within such an environment would need to include features such as key redistribution, failure resilience, and allow for the revocation of a TEE participation in a Combined TEE. It is furthermore possible to design a threshold Combined TEE similar to the concept of threshold cryptography. Such a threshold Combined TEE could rely on a threshold $K$ of the $N$ TEEs involved in TEE$^N$. This means that the system could tolerate $N-K$ TEE failures or temporary outages if a TEE is slow to respond to a request. In regards to its protocol functions, such a threshold system can be built with threshold cryptography and mechanisms known from byzantine fault tolerance research.

**Stateless TEEs:** We only considered stateful TEEs that maintain a state of clients' requests and that can act upon such state. It might also be interesting to consider a case of TEE$^N$ where the Combined TEE does not maintain a state but only acts similar to a microservice. Such microservice could be contacted with one message by a client who receives a single response message by the Combined TEE system. This eliminates the necessity for multiple round trips during key exchange where the TEEs need to always verify the sender's signing keys. Similar systems exist in prior work [2] and utilize prepublished public keys to set up secure and uni-directionally attested channels. A stateless Combined TEE also eliminates the necessity of predefining the set of TEE$^N$ as this could be encoded in the response to the single TEE$^N$ request.

**Detecting compromise:** We mostly focused on ensuring security of the Combined TEE after an adversary compromised one or multiple TEEs. However, it would also be invaluable for a user to know when and where a compromise happened, even if this information can only be retrieved after a successful Combined TEE operation. For any operation where the adversary purely learns secret information such as shares of a secret, there remains no evidence of a compromise based on the outputs of a TEE. However, it might be possible to detect an adversary for decisional operations where a TEE either outputs a random string or influences a decision based on its own inputs. For example, if the random string returned by one TEE is trivial (null bytes) or constant throughout protocol runs, it can reasonably be assumed that this TEE is either compromised or defective. Unfortunately, an adversary can trivially prevent this type of detection but exploring different options for detecting malicious participants is an interesting opportunity for future work.

## 10.2 Summary

Trusted hardware and Trusted Execution Environments have become widely available in end-user and consumer-grade products. At the same time, a wide array of research uncovered vulnerabilities, side-channel attacks, and bugs in these TEEs that damage the trust users place in them. We identify two types of TEE compromising adversaries: A weak adversary $\mathcal{M}_W$ that compromises the confidentiality but not integrity of a TEE, and a strong adversary $\mathcal{M}_S$ that compromises both the confidentiality and integrity of a TEE. While $\mathcal{M}_W$ could be an adversary that has access to a side-channel attack on the TEE and read all secrets that are stored inside a TEE, he can not impersonate one. This stands in contrast to the capabilities of $\mathcal{M}_S$ who can fully simulate a TEE and impersonate it by generating arbitrary, valid quotes. Placing both adversaries in a scenario that allows clients to communicate with a TEE hosted by an untrusted party reveals that they would both be able to undermine the security of the clients. Disregarding DoS attacks against the TEE, both adversaries could read the communication between client and TEE in plaintext and possibly even impersonate the TEE. To resolve this issue, we propose TEE$^2$. Similar to the concept of cryptographic combiners, TEE$^2$ combines multiple TEEs to ensure

security even under the compromise of all but one TEE. By combining multiple TEE vendors or even different implementations of the same protocol using the same type of TEE, TEE$^2$ can mitigate the impact of possible vulnerabilities and reestablish trust in TEEs. We propose a multitude of protocols to perform utility operations such as key exchange and messaging, one party protocols such as random number generation, key operations, and signature generation, and two-party protocols such as a secure store-and-forward scheme and one example of MPC, namely Oblivious Transfer. All Combined TEE protocols are compared to an Ideal TEE version and can be extended to an arbitrary amount of involved TEEs, effectively creating a TEE$^N$ system. We formally verify the security of a subset of the presented protocols with the Tamarin prover. Additionally, we implement a prototype of TEE$^2$ using Python for the client software and a C++ based Intel SGX implementation as a TEE. Even without optimizations, the prototype implementation already scales linearly with regard to input parameters to the random number generation and also scales linearly with an increasing amount of TEEs in TEE$^N$.
We conclude that TEE$^2$ is a valid approach to mitigate the effect of recent attacks and the resulting loss of trust in Trusted Execution Environments. Our designed protocols have a low complexity overhead, are formally verfied, scale linearly, and can be used to combine an arbitrary amount of TEEs which allows each user to granularly choose her own security level.

## Bibliography

[1] S. Agrawal et al.: “Dise: Distributed symmetric-key encryption”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1993–2010.

[2] F. Alder et al.: “S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX”. In: *arXiv preprint arXiv:1810.06080* (2018).

[3] I. Anati et al.: “Innovative Technology for CPU Based Attestation and Sealing”. In: *HASP@ ISCA* 11 (2013).

[4] R. Ankele; A. Simpson: “On the performance of a trustworthy remote entity in comparison to secure multi-party computation”. In: *Trustcom/BigDataSE/ICESS, 2017 IEEE*. IEEE. 2017, pp. 1115–1122.

[5] R. Ankele et al.: “Applying the trustworthy remote entity to privacy-preserving multiparty computation: Requirements and criteria for large-scale applications”. In: *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences*. IEEE. 2016, pp. 414–422.

[6] ARM: “Building a Secure System using TrustZone® Technology”. In: *ARM white paper* (2009).

[7] R. Bahmani et al.: “Secure multiparty computation from SGX”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 477–497.

[8] D. Basin; C. Cremers: “Know your enemy: Compromising adversaries in protocol analysis”. In: *ACM Transactions on Information and System Security (TISSEC)* 17.2 (2014), p. 7.

[9] A. Baumann; M. Peinado; G. Hunt: “Shielding applications from an untrusted cloud with haven”. In: *ACM Transactions on Computer Systems (TOCS)* 33.3 (2015), p. 8.

[10] M. Bellare; S. Micali: “Non-interactive oblivious transfer and applications”. In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 547–557.

[11] F. Brandt: “Efficient cryptographic protocol design based on distributed El Gamal encryption”. In: *International Conference on Information Security and Cryptology*. Springer. 2005, pp. 32–47.

[12] F. Brasser et al.: “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[13] E. Brickell; J. Li: “Enhanced privacy ID from bilinear pairing for hardware authentication and attestation”. In: *2010 IEEE Second International Conference on Social Computing*. IEEE. 2010, pp. 768–775.

[14] C. Brzuska et al.: “Physically uncloneable functions in the universal composition framework”. In: *Annual Cryptology Conference (CRYPTO)*. Springer. 2011, pp. 51–70.

[15] G. Chen et al.: “SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution”. In: *arXiv preprint arXiv:1802.09085* (2018).

[16] S. Chen et al.: “Detecting privileged side-channel attacks in shielded execution with Déjá Vu”. In: *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*. ACM. 2017, pp. 7–18.

[17] K. Cohn-Gordon; C. Cremers; L. Garratt: “On post-compromise security”. In: *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE. 2016, pp. 164–178.

[18] V. Costan; S. Devadas: “Intel SGX Explained.” In: *IACR Cryptology ePrint Archive* 2016.086 (2016).

[19] V. Costan; I. A. Lebedev; S. Devadas: “Sanctum: Minimal Hardware Extensions for Strong Software Isolation.” In: *25th USENIX Security Symposium, USENIX Security 16*. 2016, pp. 857–874.

[20] J. Dreier et al.: “Automated Unbounded Verification of Stateful Cryptographic Protocols with Exclusive OR”. In: *31st IEEE Computer Security Foundations Symposium (CSF’2018)*. 2018.

[21] T. ElGamal: “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[22] M. Fischlin et al.: “Obfuscation combiners”. In: *Annual Cryptology Conference*. Springer. 2016, pp. 521–550.

[23] M. Fischlin et al.: “Secure set intersection with untrusted hardware tokens”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2011, pp. 1–16.

[24] P.-A. Fouque; A. Joux; M. Tibouchi: “Injective encodings to elliptic curves”. In: *Australasian Conference on Information Security and Privacy*. Springer. 2013, pp. 203–218.

[25] M. K. Franklin; M. K. Reiter: “Fair exchange with a semi-trusted third party”. In: *Proceedings of the 1997 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 1997, pp. 1–5.

[26] Y. Gang; F. Dengguo: “Proxy oblivious transfer protocol”. In: *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*. IEEE. 2006, 6–pp.

[27] Q. Ge et al.: “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[28] Global Platform: “Introduction to Trusted Execution Environments”. In: *Global Platform white paper* (2018).

[29] Global Platform: “The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market”. In: *Global Platform white paper* (2015).

[30] O. Goldrcich; R. Vainish: “How to solve any protocol problem-an efficiency improvement”. In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1987, pp. 73–86.

[31] D. Gupta et al.: “Using intel software guard extensions for efficient two-party secure function evaluation”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 302–318.

[32] M. Hähnel; W. Cui; M. Peinado: “High-Resolution Side Channels for Untrusted Operating Systems”. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 299–312.

[33] D. Harnik et al.: “On robust combiners for oblivious transfer and other primitives”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2005, pp. 96–113.

[34] A. Herzberg: “Folklore, practice and theory of robust combiners”. In: *Journal of Computer Security* 17.2 (2009), pp. 159–189.

[35] A. Herzberg: “On tolerant cryptographic constructions”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2005, pp. 172–190.

[36] F. Hetzelt; R. Buhren: “Security analysis of encrypted virtual machines”. In: *ACM SIGPLAN Notices*. Vol. 52. 7. ACM. 2017, pp. 129–142.

[37] M. Hoekstra et al.: “Using innovative instructions to create trustworthy software solutions.” In: *HASP@ ISCA* 13 (2013).

[38] S. Hosseinzadeh et al.: “Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization”. In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX)*. ACM. 2018, pp. 42–47.

[39] Intel: Software Guard Extensions Developer Guide. https://software.intel.com/en-us/documentation/sgx-developer-guide. 2017.

[40] Intel: Software Guard Extensions SDK Developer Reference. https://software.intel.com/en-us/sgx-sdk/documentation.

[41] S. Johnson et al.: “Intel® Software Guard Extensions: EPID Provisioning and Attestation Services”. In: *Intel white paper* (2016).

[42] D. Kaplan; J. Powell; T. Woller: “AMD memory encryption”. In: *AMD white paper* (2016).

[43] J. Kilian: “Founding crytpography on oblivious transfer”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1988, pp. 20–31.

[44] N. Koblitz: “Elliptic curve cryptosystems”. In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[45] P. Kocher et al.: “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (SP’19)*. 2019.

[46] A. Kurnikov et al.: “Keys in the Clouds: Auditable Multi-device Access to Cryptographic Credentials”. In: *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*, 40:1–40:10.

[47] S. Lee et al.: “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing”. In: *26th USENIX Security Symposium, USENIX Security 17*. 2017, pp. 16–18.

[48] M. Lipp et al.: “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium, USENIX Security 18*. 2018, pp. 973–990.

[49] G. Lowe: “A hierarchy of authentication specifications”. In: *Computer security foundations workshop, 1997. Proceedings., 10th*. IEEE. 1997, pp. 31–43.

[50] S. Matetic et al.: “ROTE: Rollback Protection for Trusted Execution.” In: *26th USENIX Security Symposium, USENIX Security 17*. 2017, pp. 1289–1306.

[51] V. Mavroudis et al.: “A touch of evil: High-assurance cryptographic hardware from untrusted components”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1583–1600.

[52] F. McKeen et al.: “Innovative instructions and software model for isolated execution.” In: *HASP@ ISCA* 10 (2013).

[53] S. Meier et al.: “The TAMARIN prover for the symbolic analysis of security protocols”. In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 696–701.

[54] P. Mell; T. Grance, et al.: “The NIST definition of cloud computing”. In: *National institute of standards and technology* 53.6 (2009).

[55] S. Micali; R. Sidney: “A simple method for generating and sharing pseudo-random functions, with applications to clipper-like key escrow systems”. In: *Annual International Cryptology Conference*. Springer. 1995, pp. 185–196.

[56] M. Morbitzer et al.: “SEVered: Subverting AMD’s Virtual Machine Encryption”. In: *Proceedings of the 11th European Workshop on Systems Security*. ACM. 2018, p. 1.

[57] M. Naor; B. Pinkas: “Distributed oblivious transfer”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2000, pp. 205–219.

[58] M. Naor; B. Pinkas: “Efficient oblivious transfer protocols”. In: *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2001, pp. 448–457.

[59] M. Naor; B. Pinkas; R. Sumner: “Privacy preserving auctions and mechanism design”. In: *Proceedings of the 1st ACM conference on Electronic commerce*. ACM. 1999, pp. 129–139.

[60] J. Noorman et al.: “Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base.” In: *22nd USENIX Security Symposium, USENIX Security 13*. 2013, pp. 479–494.

[61] R. Pass; E. Shi; F. Tramer: “Formal abstractions for attested execution secure processors”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017, pp. 260–289.

[62] A. Paverd: “Enhancing communication privacy using trustworthy remote entities”. PhD thesis. University of Oxford, 2015.

[63] M. O. Rabin: “How To Exchange Secrets with Oblivious Transfer”. In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 187.

[64] M. D. Ryan: “Enhanced Certificate Transparency and End-to-End Encrypted Mail.” In: *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. 2014.

[65] B. Schmidt et al.: “Automated analysis of Diffie-Hellman protocols and advanced security properties”. In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE. 2012, pp. 78–94.

[66] G. Shafi; S. Micali: “Probabilistic encryption”. In: *Journal of computer and system sciences* 28.2 (1984), pp. 270–299.

[67] M.-W. Shih et al.: “T-SGX: Eradicating controlled-channel attacks against enclave programs”. In: *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. 2017.

[68] S. Shinde et al.: “Preventing page faults from telling your secrets”. In: *Proceedings of the 2016 ACM Asia Conference on Computer and Communications Security*. ACM. 2016, pp. 317–328.

[69] N. P. Smart: Cryptography Made Simple. Springer, 2016.

[70] Y. Swami: “SGX Remote Attestation is not Sufficient”. In: *IACR Cryptology ePrint Archive* 2017.736 (2017).

[71] C.-C. Tsai et al.: “Cooperation and security isolation of library OSes for multi-process applications”. In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 9.

[72] J. Van Bulck; F. Piessens; R. Strackx: “Sgx-step: A practical attack framework for precise enclave execution control”. In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM. 2017, p. 4.

[73] J. Van Bulck et al.: “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *27th USENIX Security Symposium, USENIX Security 18*. 2018, pp. 991–1008.

[74] J. Van Bulck et al.: “Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution”. In: *26th USENIX Security Symposium, USENIX Security 17*. 2017, pp. 1041–1056.

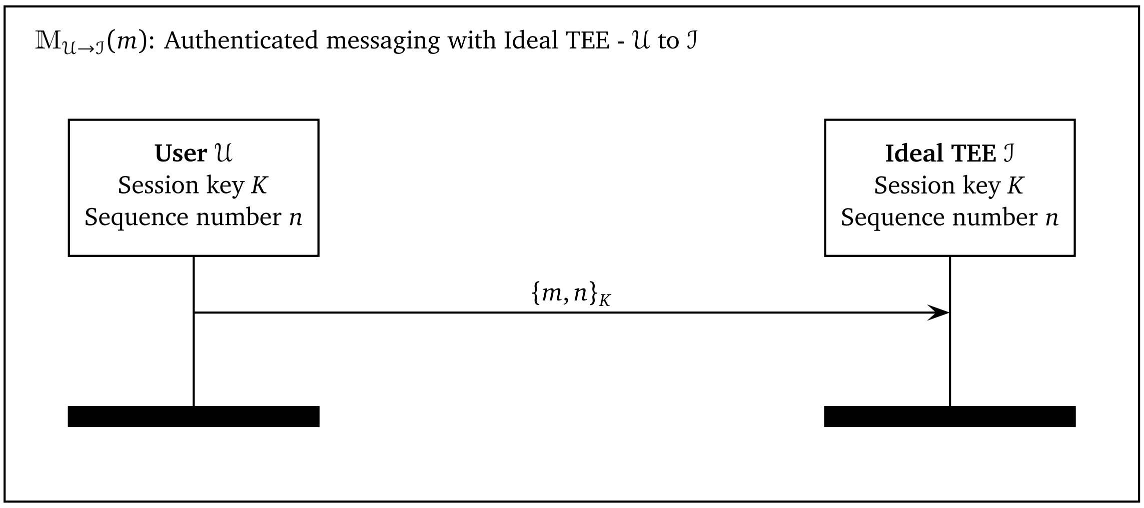
[75] A. Vasudevan et al.: “Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?” In: *International Conference on Trust and Trustworthy Computing*. Springer. 2012, pp. 159–178.

[76] O. Weisse et al.: “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. In: *Technical report* (2018).

[77] Y. Xu; W. Cui; M. Peinado: “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”. In: *36th IEEE Symposium on Security and Privacy (SP’15)*. 2015, pp. 640–656.

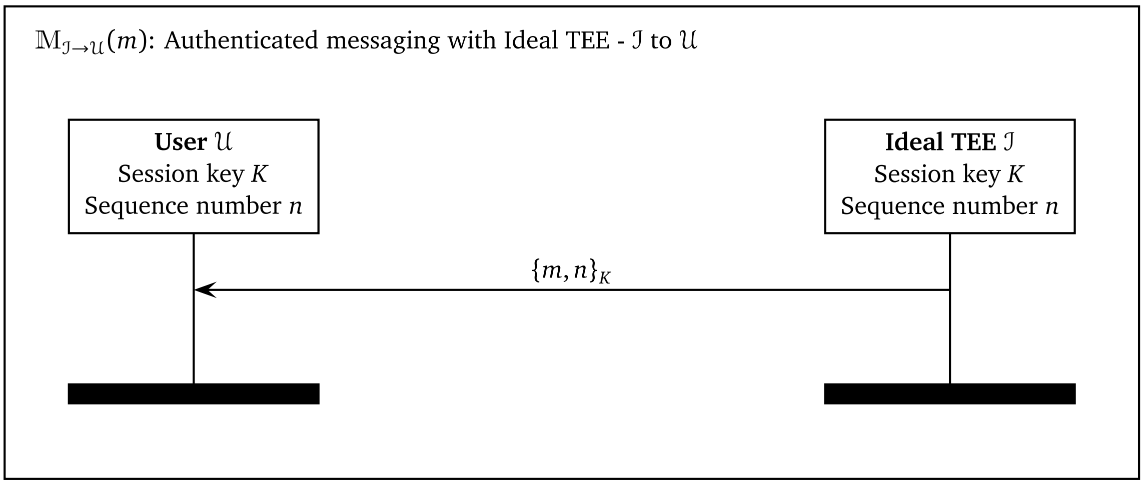
[78] N. Zhang et al.: “TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices.” In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 980.

# Appendices



$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(m)$: Authenticated messaging with Ideal TEE - $\mathcal{U}$ to $\mathcal{I}$
**User** $\mathcal{U}$
Session key $K$
Sequence number $n$
**Ideal TEE** $\mathcal{I}$
Session key $K$
Sequence number $n$
$\{m, n\}_K$

**Figure A.1.:** Messaging between user $\mathcal{U}$ and Ideal TEE instance $\mathcal{I}$. In the following denoted as $\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(m)$.



$\mathbb{M}_{\mathcal{I}\to\mathcal{U}}(m)$: Authenticated messaging with Ideal TEE - $\mathcal{I}$ to $\mathcal{U}$
**User** $\mathcal{U}$
Session key $K$
Sequence number $n$
**Ideal TEE** $\mathcal{I}$
Session key $K$
Sequence number $n$
$\{m, n\}_K$

**Figure A.2.:** Messaging between Ideal TEE instance $\mathcal{I}$ and user $\mathcal{U}$. In the following denoted as $\mathbb{M}_{\mathcal{I}\to\mathcal{U}}(m)$.

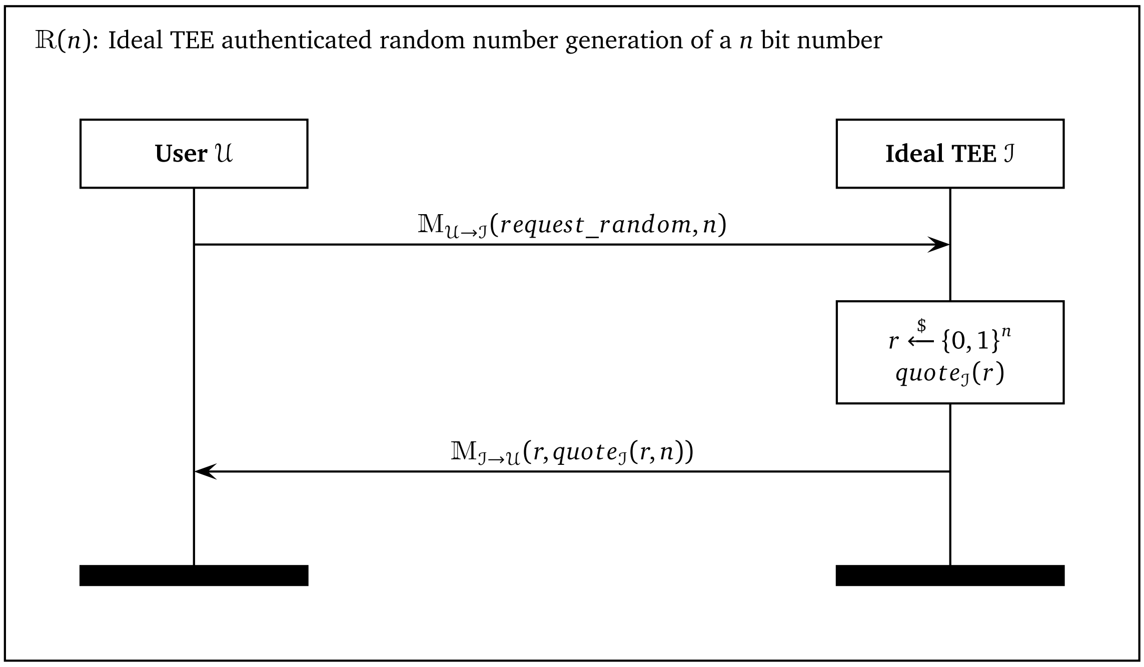

$\mathbb{R}(n)$: Ideal TEE authenticated random number generation of a $n$ bit number
User $\mathcal{U}$
Ideal TEE $\mathcal{I}$
$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(request\_random, n)$
$r \overset{\$}{\leftarrow} \{0,1\}^n$
$quote_{\mathcal{I}}(r)$
$\mathbb{M}_{\mathcal{I}\to\mathcal{U}}(r, quote_{\mathcal{I}}(r,n))$

**Figure A.3.:** Authenticated random number generation between user $\mathcal{U}$ and Ideal TEE instance $\mathcal{I}$.

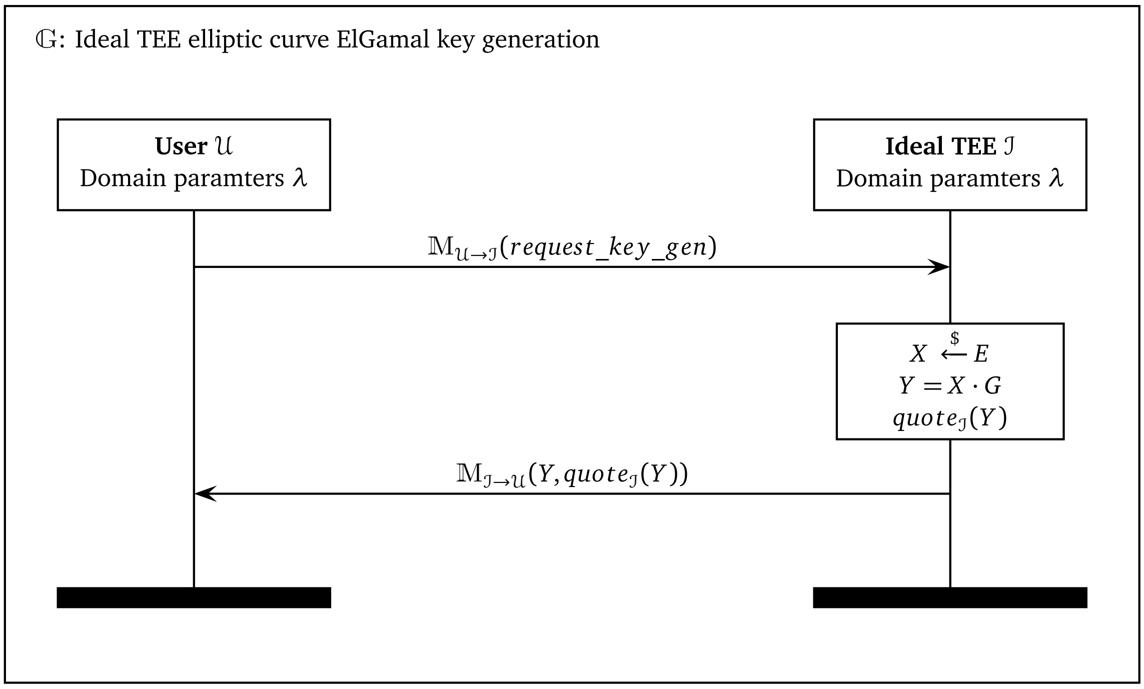

$\mathbb{G}$: Ideal TEE elliptic curve ElGamal key generation
User $\mathcal{U}$
Domain paramters $\lambda$
Ideal TEE $\mathcal{I}$
Domain paramters $\lambda$
$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(request\_key\_gen)$
$X \overset{\$}{\leftarrow} E$
$Y = X \cdot G$
$quote_{\mathcal{I}}(Y)$
$\mathbb{M}_{\mathcal{I}\to\mathcal{U}}(Y, quote_{\mathcal{I}}(Y))$

**Figure A.4.:** Ideal TEE key generation between user $\mathcal{U}$ and Ideal TEE instance $\mathcal{I}$. $\mathcal{I}$ generates a private key $X$ and its corresponding public key $Y = X \cdot G$ and sends the public key together with a quote back to $\mathcal{U}$.

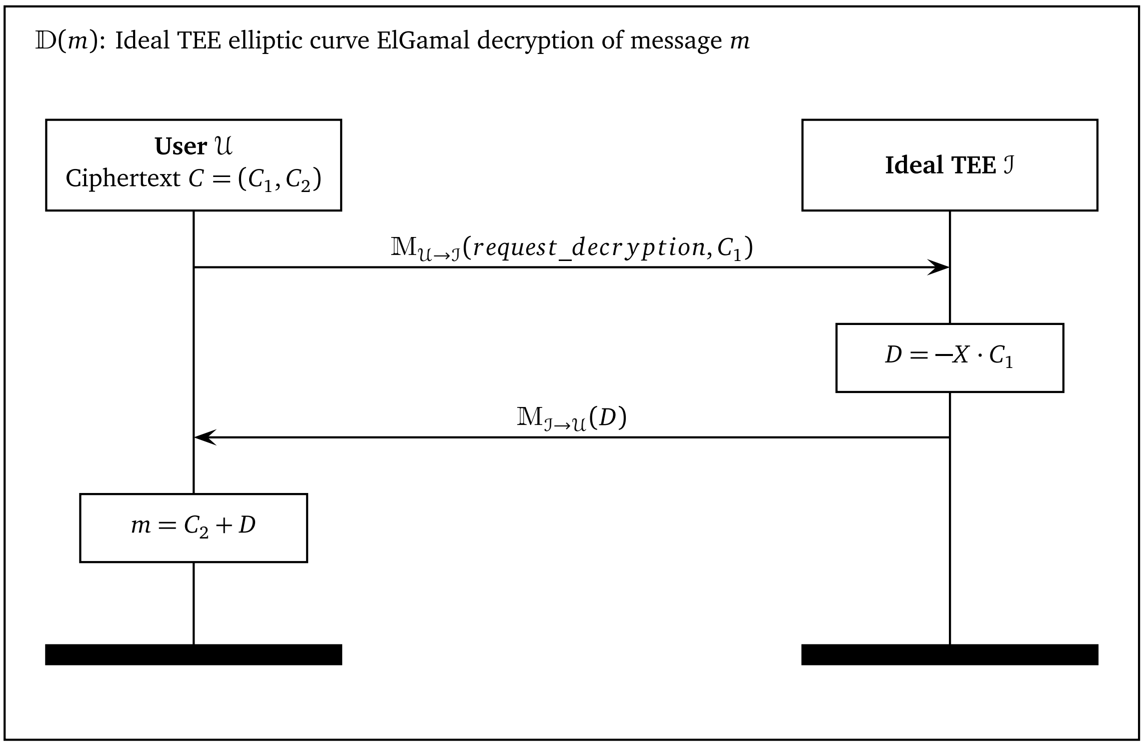

$\mathbb{D}(m)$: Ideal TEE elliptic curve ElGamal decryption of message $m$
User $\mathcal{U}$
Ciphertext $C=(C_1,C_2)$
Ideal TEE $\mathcal{I}$
$\mathbb{M}_{\mathcal{U}\rightarrow\mathcal{I}}(request\_decryption, C_1)$
$D=-X\cdot C_1$
$\mathbb{M}_{\mathcal{I}\rightarrow\mathcal{U}}(D)$
$m=C_2+D$

**Figure A.5.:** Ideal TEE decryption of message $m$ for user $\mathcal{U}$ and Ideal TEE instance $\mathcal{I}$. The Ciphertext $C$ is a tuple of $(C_1,C_2)$ with $C_1=k\cdot G$ and $C_2=m+k\cdot Y$ where $k$ is random. The decryption works by sending $C_1$ to the $\mathcal{I}$ which returns $D=X\cdot C_1$. The client then subtracts $D$ from $C_2$ to obtain the original message.

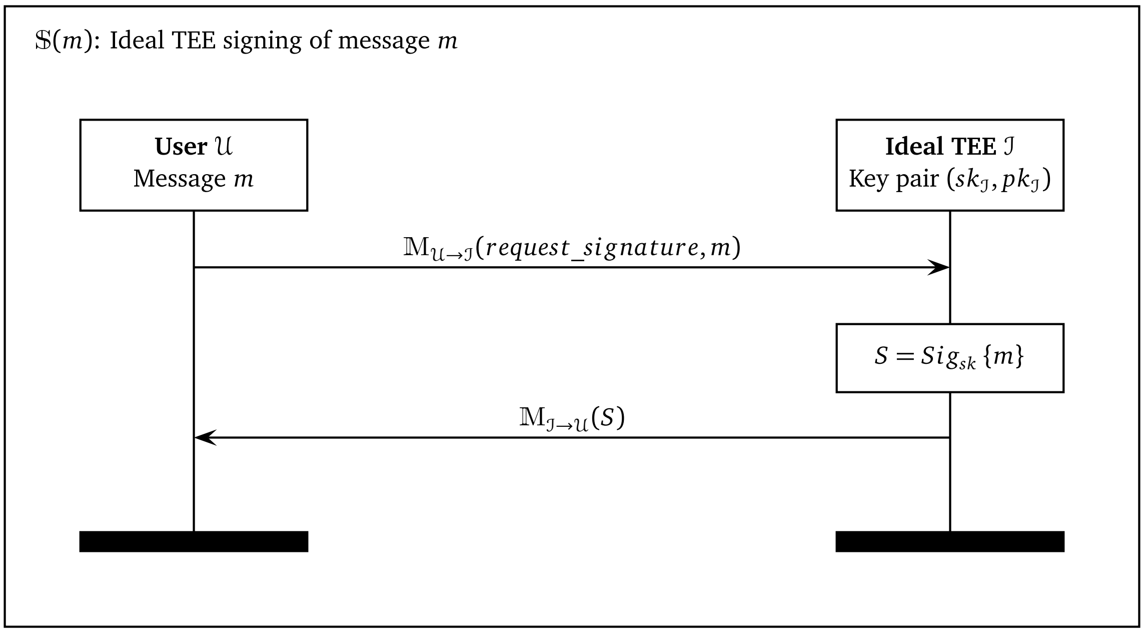

$\mathbb{S}(m)$: Ideal TEE signing of message $m$
User $\mathcal{U}$
Message $m$
Ideal TEE $\mathcal{I}$
Key pair $(sk_{\mathcal{I}}, pk_{\mathcal{I}})$
$\mathbb{M}_{\mathcal{U}\rightarrow\mathcal{I}}(request\_signature, m)$
$S=Sig_{sk}\{m\}$
$\mathbb{M}_{\mathcal{I}\rightarrow\mathcal{U}}(S)$

**Figure A.6.:** Ideal TEE signing of message $m$ for user $\mathcal{U}$ and Ideal TEE instance $\mathcal{I}$. $\mathcal{U}$ has access to $pk_{\mathcal{I}}$ and in case a remote party needs to verify that the public key belongs to an Ideal TEE, $\mathcal{I}$ would need to publish a quote containing $pk_{\mathcal{I}}$ to authenticate the public key.

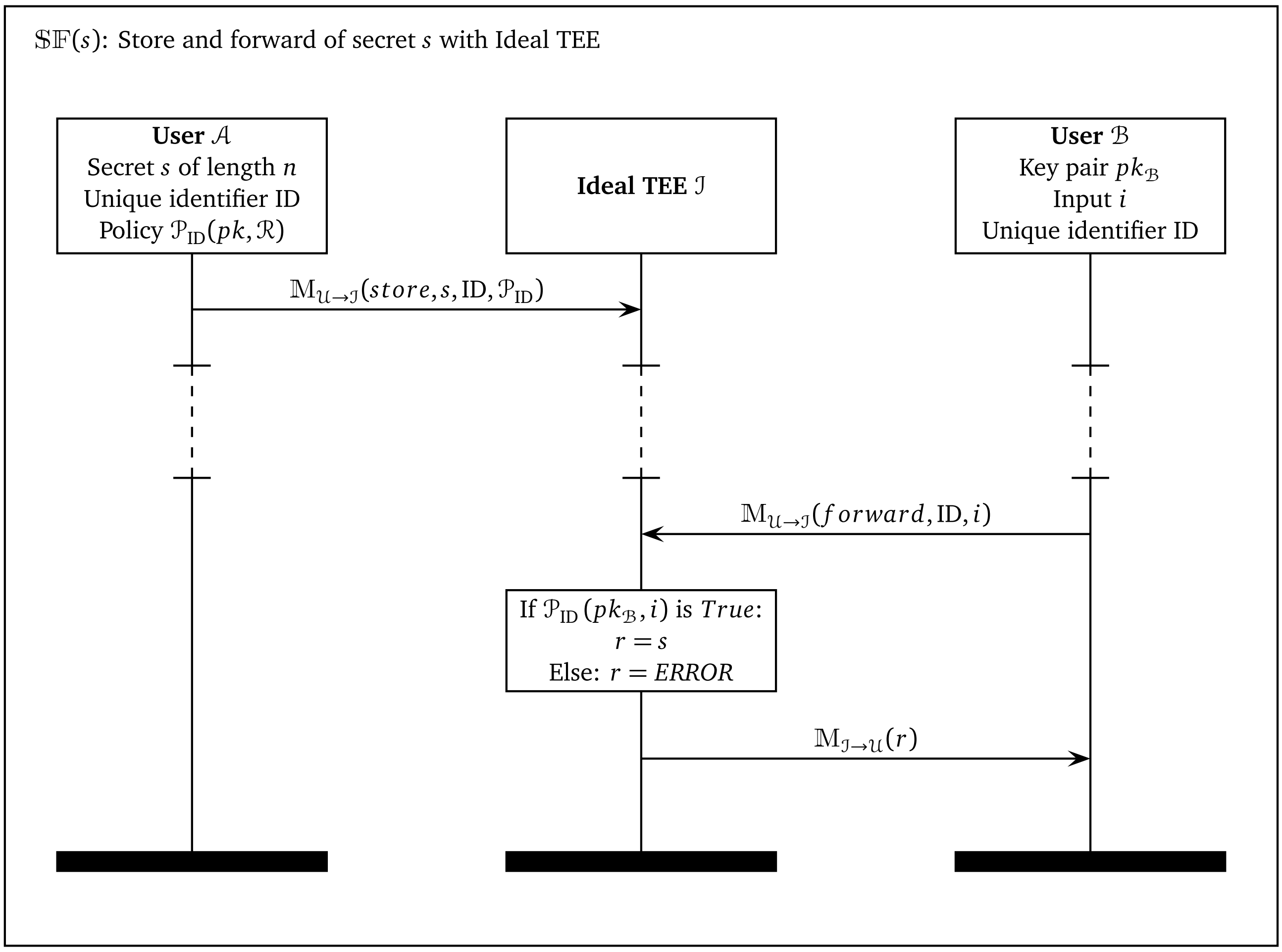

$\mathbb{SF}(s)$: Store and forward of secret $s$ with Ideal TEE
**User** $\mathcal{A}$
Secret $s$ of length $n$
Unique identifier ID
Policy $\mathcal{P}_{\mathrm{ID}}(pk, \mathcal{R})$
**Ideal TEE** $\mathcal{I}$
**User** $\mathcal{B}$
Key pair $pk_{\mathcal{B}}$
Input $i$
Unique identifier ID
$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(store, s, \mathrm{ID}, \mathcal{P}_{\mathrm{ID}})$
$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(forward, \mathrm{ID}, i)$
If $\mathcal{P}_{\mathrm{ID}}\,(pk_{\mathcal{B}}, i)$ is $True$:
$r = s$
Else: $r = ERROR$
$\mathbb{M}_{\mathcal{I}\to\mathcal{U}}(r)$

**Figure A.7.:** Policy based store and forward with Ideal TEE $\mathcal{I}$. In the following denoted as $\mathbb{SF}(s)$. User $\mathcal{A}$ stores the secret $s$ on $\mathcal{I}$ which forwards it to user $\mathcal{B}$. Furthermore, $\mathcal{I}$ enforces the policy $\mathcal{P}$ given by $\mathcal{A}$ and ensures that $\mathcal{B}$ can only retrieve the secret if his input matches $\mathcal{P}$.

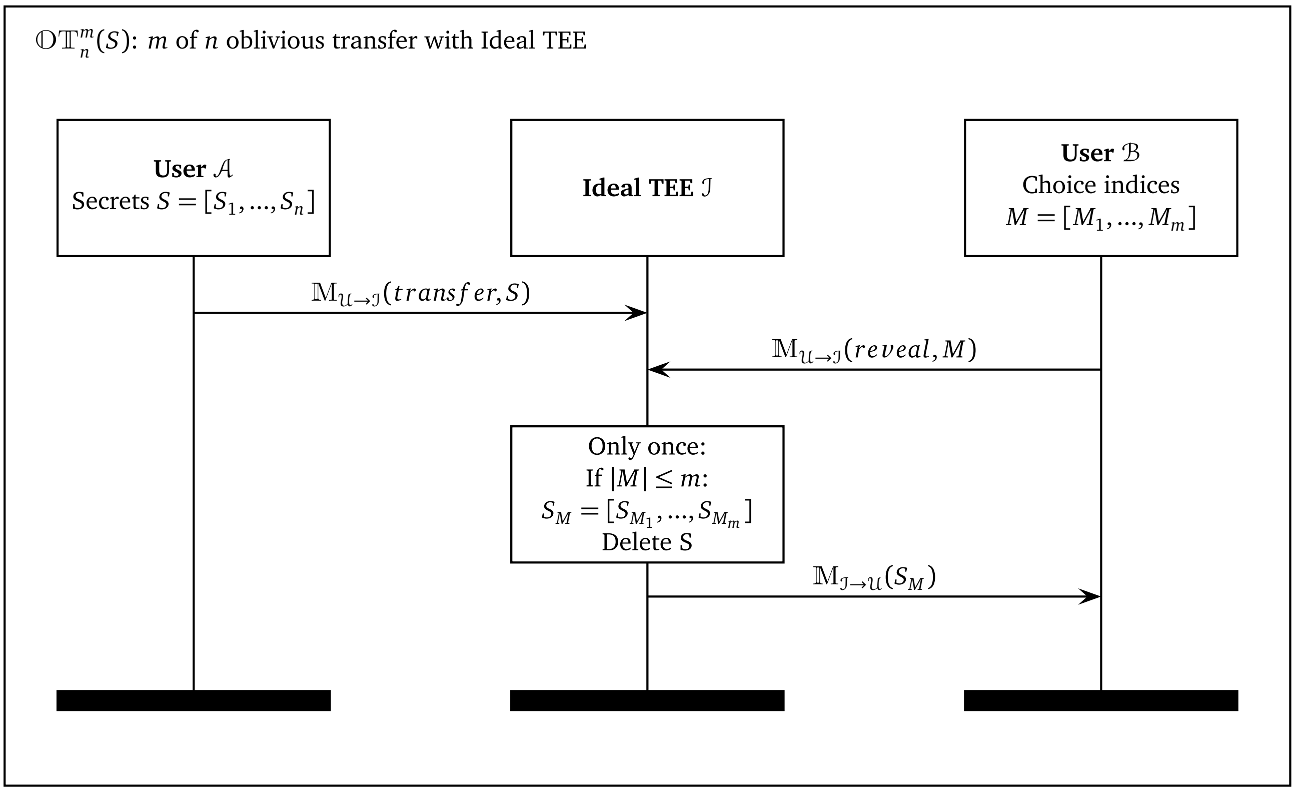

$\mathbb{OT}_n^m(S)$: $m$ of $n$ oblivious transfer with Ideal TEE
User $\mathcal{A}$
Secrets $S = [S_1, ..., S_n]$
Ideal TEE $\mathcal{I}$
User $\mathcal{B}$
Choice indices
$M = [M_1, ..., M_m]$
$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(transfer, S)$
$\mathbb{M}_{\mathcal{U}\to\mathcal{I}}(reveal, M)$
Only once:
If $|M| \leq m$:
$S_M = [S_{M_1}, ..., S_{M_m}]$
Delete S
$\mathbb{M}_{\mathcal{I}\to\mathcal{U}}(S_M)$

**Figure A.8.:** Oblivious transfer of secret vector S with Ideal TEE $\mathcal{I}$. In the following denoted as $\mathbb{OT}_n^m(S)$. User $\mathcal{A}$ obliviously shares the set of secrets $S$ with user $\mathcal{B}$ through $\mathcal{I}$. $\mathcal{I}$ enforces that $\mathcal{B}$ can only retrieve $m$ of the $n$ elements in the set and prevents $\mathcal{A}$ from detecting which secrets $\mathcal{B}$ picked.

# B Tamarin source

## B.1 Tamarin model of the Ideal TEE

**Listing B.1:** Full Tamarin Ideal TEE model for key exchange and random number generation

```
/*
Ideal TEE
*/


theory IdealTEE
begin

builtins: diffie-hellman, signing, symmetric-encryption

// TEEs have a secret quote key that the adversary can't access
functions: quote_key/1 [private], fst/1, snd/1

equations:
  fst(<x.1, x.2>) = x.1
, snd(<x.1, x.2>) = x.2


rule register_tee_key:
    let
        pubkey = pk(quote_key($tee))
    in
    [  ]
    --[ CreateTEE($tee) ]->
    [ !TEEIdentity($tee, pubkey)
    , Out(pubkey)
    ]


/*
 * ============
 * Key Exchange
 * ============
 *
 * Simple 3-way DH key exchange with signed messages.
 * Signing by user and quotes by TEE ensure authenticity of KE
 *
*/

/*
 * User to TEE
 * Sends:
 *  - Client Hello
 *  - User public key used for signing
 *  - Nonce of this request
 * This first message also creates the user. This is consistent with
 * the view of the TEE where random users contact it and establish a channel
```

```
*/
rule ke_msg_1:
    let
        user_sign_pubkey = pk(~user_sign_privkey)
    in
        [ Fr(~user_sign_privkey)
        , Fr(~nonce)
        ]
        --[
            KE_MSG_1($tee, user_sign_pubkey, ~nonce)
        ]->
        [ !UserIdentity(~user_sign_privkey, user_sign_pubkey)
        , UserSessionNonce($tee, ~nonce, user_sign_pubkey)
          // Send the client hello
        , Out(<'ke_client_hello', user_sign_pubkey, ~nonce>)
          // and publish user pubkey
        , Out(user_sign_pubkey)
        ]

/*
 * TEE to user
 * Sends:
 *  - public session key of TEE
 *  - quote over <public session key of TEE, public signing key of client, nonce>
*/
rule ke_msg_2:
    let
        tee_session_pubkey = 'g'^~tee_session_privkey
    in
        [ !TEEIdentity($tee, tee_pubkey)
        , Fr(~tee_session_privkey)
        , In(<'ke_client_hello', $tee, user_sign_pubkey, nonce>)
        ]
        --[
            KE_MSG_2( $tee, user_sign_pubkey, nonce)
            // Ensure that the TEE pubkey comes from a correct quote_key and is not generated by the adversary.
          , Eq(tee_pubkey, pk(quote_key($tee)))
        ]->
        [
          // Store tee keys and also signing pubkey for last message
          TEESessionKeys($tee, tee_session_pubkey, ~tee_session_privkey, user_sign_pubkey)
          // Send the TEE hello message
        , Out(<'ke_tee_hello', tee_session_pubkey,
              sign(<tee_session_pubkey, user_sign_pubkey, nonce>, quote_key($tee))>)
        ]

/*
 * User to TEE
 * Sends:
 *  - Client Response
 *  - User session pubkey
 *  - Signature over <public session key of user, public session key of TEE>
*/
rule ke_msg_3:
    let
        user_session_pubkey = 'g'^~user_session_privkey
        sessionkey = tee_session_pubkey^~user_session_privkey
    in
        [ Fr(~user_session_privkey)
        , !TEEIdentity($tee, tee_quote_pubkey)
        , !UserIdentity(user_sign_privkey, user_sign_pubkey)
        , UserSessionNonce($tee, nonce, user_sign_pubkey)
        , In(<'ke_tee_hello', tee_session_pubkey, quote>)
        ]
        --[
            // Ensure that the Quote pubkey comes from a correct quote_key
            //  (as in, is not generated by the adversary).
            Eq(tee_quote_pubkey, pk(quote_key($tee)))
            // Ensure that the Quote contains the expected contents and is signed with the quote key
        ,   Eq(verify(quote, <tee_session_pubkey, user_sign_pubkey, nonce>, tee_quote_pubkey), true)
        ,   KE_MSG_3($tee, user_sign_pubkey, nonce, tee_session_pubkey)
        ,   User_Established(user_sign_pubkey, $tee, sessionkey)
        ,   User_Session(user_sign_pubkey, $tee)
        ]->
        [ !Session_User($tee, sessionkey, user_sign_pubkey)
          // Send the client response
        , Out(<'ke_client_response', user_session_pubkey,
              sign(<user_session_pubkey, tee_session_pubkey>, user_sign_privkey)>)
        ]


/*
 * Last step for TEE to verify msg3
 * Takes msg 3, verifies it and outputs the session
*/
rule ke_msg_3_tee_establish: //Required for the TEE to check last nonce (tee pubkey)
    let
        sessionkey = user_session_pubkey^tee_session_privkey
    in
        [ In(<'ke_client_response', user_session_pubkey, signature>)
        , TEESessionKeys($tee, tee_session_pubkey, tee_session_privkey, user_sign_pubkey)
        ]
        --[
            // Ensure that the signature contains the expected elements and is signed by the user's
            //   pubkey encountered in first message
            Eq(verify(signature, <user_session_pubkey, tee_session_pubkey>, user_sign_pubkey), true)
        ,   KE_MSG_3_VERIFY($tee, user_sign_pubkey, tee_session_pubkey)
        ,   TEE_Established($tee, user_sign_pubkey, sessionkey)
        ,   TEE_Session($tee, user_sign_pubkey)
        ]->
        [ !Session_TEE($tee, sessionkey, user_sign_pubkey) ]


/*
 * ========================
 * Random Number Generation
 * ========================
 *
 * The user requests a random number from the TEE
 *
*/

```

```
rule rng_request:
    [ !Session_User($tee, sessionkey, user_sign_pubkey)
    , Fr(~nonce)
    ]
    --[ RNG_Request(user_sign_pubkey, $tee, ~nonce)
    ]->
    [ Out(<'rng_request', senc(~nonce, sessionkey)>)
    , UserRNGRequest(user_sign_pubkey, $tee, ~nonce)
    ]

rule rng_response:
  let
    decrypted_nonce = sdec(nonce, sessionkey)
  in
    [ !Session_TEE($tee, sessionkey, user_sign_pubkey)
    , In(<'rng_request', nonce>)
    , Fr(~random)
    ]
    --[ RNG_Response_In(user_sign_pubkey, $tee, decrypted_nonce)
      , RNG_Response_Out($tee, user_sign_pubkey, ~random)
    ]->
    [ Out(<'rng_response', senc(<~random, decrypted_nonce>, sessionkey),
          sign(<~random, decrypted_nonce>, quote_key($tee))>)
    ]

rule rng_complete:
  let
    random = fst(sdec(msg, sessionkey))
    nonce  = snd(sdec(msg, sessionkey))
  in
    [ !Session_User($tee, sessionkey, user_sign_pubkey)
    , !TEEIdentity($tee, tee_pubkey)
    , UserRNGRequest(user_sign_pubkey, $tee, nonce_original)
    , In(<'rng_response', msg, quote>)
    ]
    --[ Eq(nonce, nonce_original)
        // Ensure that the Quote contains the expected contents and is signed with the quote key of the TEE
      , Eq(verify(quote, <random, nonce>, tee_pubkey), true)
      , RandomGenerated($tee, user_sign_pubkey, random)
     ]->
    [ RandomNumber($tee, user_sign_pubkey, random)
    ]

/*
 * ============
 * Lemmas
 * ============
 *
*/

/*
 * We restrict our attention to traces where all equality checks succeed.
*/
restriction Equality_Checks_Succeed: "All x y #i . Eq(x,y) @ i ==> x = y"
restriction TEEs_Unique:
  "
  All TEE #i #j .
      CreateTEE(TEE) @ #i
    & CreateTEE(TEE) @ #j
    ==>
    #i = #j
  "

/*
 * Source lemma to restrict nonce use by adversary
 *
*/
lemma random_types [sources]:
  "
  (All U_pk TEE n #i .
    RNG_Response_In(U_pk, TEE, n) @i
    ==>
    ( ( Ex #j . KU(n) @j & j < i)
    | ( Ex #j . RNG_Request(U_pk, TEE, n) @j)
    )
  )
 "


/*
 * Key Exchange:
 * - Functionality test / sanity check: Are all rules reachable and is the agreed key the same for both parties?
 * - Key secrecy test - Can adversary learn the key? Are the keys always equal?
 *
*/

/*
 * Functional lemmas check sanity of the rules.
 * 1) Check if all rules are reachable
 * 2) Check if key exchange can calculate same key for both parties
*/
lemma ke_functional_reachable: exists-trace
  "
  // Functional lemma: The last rule is reachable
  (Ex U_pk TEE n gb #i #j #k #l .
      KE_MSG_1(TEE, U_pk, n) @ #i
      & KE_MSG_2(TEE, U_pk, n) @ #j
      & KE_MSG_3(TEE, U_pk, n, gb) @ #k
      & KE_MSG_3_VERIFY(TEE, U_pk, gb) @ #l
      & i < j
      & j < k
      & k < l
  )
  "

lemma ke_functional_agreement: exists-trace
  "
  // Agreement lemma: U and TEE can agree on the same key
  (Ex U_pk TEE k1 k2 #i #j .
      User_Established(U_pk, TEE, k1) @ #i
    & TEE_Established(TEE, U_pk, k2) @ #j
    & k1 = k2
  )
  "

/*
 * First security lemma:
 * Whenever a user agrees on a key with a TEE, they agreed on the same key.
 * This is the all-traces equivalent of the previous lemma
*/
lemma ke_correct_agreement: all-traces
  "
  All U_pk TEE k1 k2 #i #j .
    (  User_Established(U_pk, TEE, k1) @#i
    &  TEE_Established(TEE, U_pk, k2) @#j
    )
    ==>
     k1 = k2
  "


/*
 * Second security lemma:
 * If both user and TEE have established a session, the adversary will not know the key.
*/
lemma ke_secure: all-traces
  "
  All U_pk TEE k #i #j .
    (  User_Established(U_pk, TEE, k) @#i
    &  TEE_Established(TEE, U_pk, k) @#j
    )
    ==>
    (not (Ex #l . K(k) @l ))
  "


/*
 * Random Number Generation
 * - Functional sanity test: Is Random Number reachable?
 * - Security: Does adversary know the generated number?
*/
lemma rng_functional: exists-trace
  "
    Ex U_pk TEE k n r #i #j #k #l #m .
       User_Established(U_pk, TEE, k) @ #i
       & TEE_Established(TEE, U_pk, k) @ #j
       // User requested a random number
       & RNG_Request(U_pk, TEE, n) @ #k
       // TEE responded
       & RNG_Response_In(U_pk, TEE, n) @#l
       & RNG_Response_Out(TEE, U_pk, r) @#l
       // User received response and constructs the result
       & RandomGenerated(TEE, U_pk, r) @#m

       // Temporal ordering:
       // Keys were established first
       & i < j
       & j < k
```

```
        // Then request and response happened
        & k < l
        & l < m
  "

lemma rng_secure: all-traces
  "
    All U_pk TEE r sessionkey #i #j #k .
      ( User_Established(U_pk, TEE, sessionkey) @ #i
      & TEE_Established(TEE, U_pk, sessionkey) @ #j
      & RandomGenerated(TEE, U_pk, r) @ #k
      )
      ==>
      (not (Ex #l . K(r) @#l ) )
  "


end
```

## B.2 Tamarin model of the Combined TEE key exchange

**Listing B.2:** Full Tamarin Combined TEE model for key exchange

```
/*
Combined TEE
This is the file to handle the Key Exchange for the Combined TEE

*/


theory CombinedTEE_KeyExchange
begin

builtins: diffie-hellman, signing, symmetric-encryption

// TEEs have a secret quote key that the adversary can't access
functions: quote_key/1 [private]


/*
 * Rule to create a TEE. This generates a new key based on the TEE name.
 * The quoting key bases on quote_key which is a private function not accessible by the adversary.
 * As such the quote_key stays hidden from the attacker until he performs a key reveal.
*/
rule register_tee_key:
    let
        pubkey = pk(quote_key($tee))
    in
    [  ]
    --[ CreateTEE($tee) ]->
    [ !TEEIdentity($tee, pubkey)
    , Out(pubkey)
    ]


/*
 * Key reveal rule
 * We have one key reveal for the key agreement protocol:
 * 1) Revealing the long term TEE key (Compromise of Integrity)
 * This rule affects future secrecy as well as integrity of the KE
 * There is no key reveal for the session key here as the session key
 *   is never actually used in this Tamarin file.
*/
rule reveal_tee_key:
  let
    privkey = quote_key($tee)
  in
    [ !TEEIdentity($tee, pubkey) ]
  --[ TEE_KEY_REVEAL($tee)
    , REVEAL_HAPPENED()
    ]->
    [ Out(privkey) ]

/*
 * ===========
 * Key Exchange
 * ============
 *
 * Simple 3-way DH key exchange with signed messages.
 * Signing by user and quotes by TEE ensure authenticity of KE
 *
*/

/*
 * User to TEE
 * Sends:
 *  - Client Hellos containing:
 *   -- User public key used for signing
 *   -- Nonce of this request
 * This first rule also creates the user. This is consistent with
 * the view of the TEE where random users contact it and establish a channel
 * It would also allow the users to have different identities for every Combined TEE.
*/
rule ke_msg_1:
    let
        user_sign_pubkey = pk(~user_sign_privkey)
    in
        [ Fr(~user_sign_privkey)
        , Fr(~nonce1)
        , Fr(~nonce2)
        ]
        --[
          KE_MSG_1($tee1, $tee2, user_sign_pubkey, ~nonce1, ~nonce2)
        ]->
        [
          // User identity object
          !UserIdentity(~user_sign_privkey, user_sign_pubkey)

          // User nonces object
        , UserSessionNonces(user_sign_pubkey, $tee1, ~nonce1, $tee2, ~nonce2)

          // Send the client hellos
        , Out(<'ke_client_hello', user_sign_pubkey, ~nonce1>)
        , Out(<'ke_client_hello', user_sign_pubkey, ~nonce2>)

          // and publish user pubkey
        , Out(user_sign_pubkey)
        ]

/*
 * TEE to user
 * Sends:
 *  - public session key of TEE
 *  - quote over <public session key of TEE, public signing key of client, nonce>
*/
rule ke_msg_2:
    let
        tee_session_pubkey = 'g'^~tee_session_privkey
    in
        [ !TEEIdentity($tee, tee_pubkey)
        , Fr(~tee_session_privkey)
        , In(<'ke_client_hello', user_sign_pubkey, nonce>)
        ]
        --[
            // Ensure that the TEE pubkey comes from a correct quote_key and is not generated by the adversary.
            Eq(tee_pubkey, pk(quote_key($tee)))

          , KE_MSG_2($tee, user_sign_pubkey, nonce)
        ]->
        [
          // Store TEE keys and also signing pubkey for verification in last message
          TEESessionKeys($tee, tee_session_pubkey, ~tee_session_privkey, user_sign_pubkey)

          // Send the TEE hello message
        , Out(<'ke_tee_hello', tee_session_pubkey,
          sign(<tee_session_pubkey, user_sign_pubkey, nonce>, quote_key($tee))>)
        ]

/*
 * User to TEE
 * Sends:
 *  - Client Responses containing:
 *   -- User session pubkey
 *   -- Signature over <public session key of user, public session key of TEE>
*/
rule ke_msg_3:
    let
        // Establish two sessionkeys with the two fresh private keys. One for each TEE.
        user_session_pubkey1 = 'g'^~user_session_privkey1
        sessionkey1 = tee1_session_pubkey^~user_session_privkey1

        user_session_pubkey2 = 'g'^~user_session_privkey2
        sessionkey2 = tee2_session_pubkey^~user_session_privkey2
    in
        [
          // Generate two new privkeys for the sessionkeys
          Fr(~user_session_privkey1)
        , Fr(~user_session_privkey2)

          // Take in both TEE identities and user identitiy
        , !TEEIdentity($tee1, tee1_quote_pubkey)
        , !TEEIdentity($tee2, tee2_quote_pubkey)
        , !UserIdentity(user_sign_privkey, user_sign_pubkey)

          // Take user session object and TEE hellos
        , UserSessionNonces(user_sign_pubkey, $tee1, nonce1, $tee2, nonce2)
        , In(<'ke_tee_hello', tee1_session_pubkey, quote1>)
        , In(<'ke_tee_hello', tee2_session_pubkey, quote2>)
        ]
        --[
            // Ensure that the quote pubkeys come from a correct quote_key
            //  (as in, are not generated by the adversary).
            Eq(tee1_quote_pubkey, pk(quote_key($tee1)))
        ,   Eq(tee2_quote_pubkey, pk(quote_key($tee2)))

            // Ensure that the quotes contain the expected contents and are signed with the quote key
        ,   Eq(verify(quote1, <tee1_session_pubkey, user_sign_pubkey, nonce1>, tee1_quote_pubkey), true)
        ,   Eq(verify(quote2, <tee2_session_pubkey, user_sign_pubkey, nonce2>, tee2_quote_pubkey), true)
```

```

            // Ensure that we talk two two different TEEs
        ,   Neq($tee1, $tee2)

            // Actions that the user is now established and has a session
        ,   User_Established(user_sign_pubkey, $tee1, sessionkey1, $tee2, sessionkey2)
        ,   User_Session(user_sign_pubkey, $tee1, $tee2)

        ,   KE_MSG_3($tee1, $tee2, user_sign_pubkey, nonce1, nonce2, tee1_session_pubkey, tee2_session_pubkey)
        ]->
        [
          // Output user session
          !Session_User(user_sign_pubkey, $tee1, sessionkey1, $tee2, sessionkey2)

          // Send the client responses
        , Out(<'ke_client_response', user_session_pubkey1,
              sign(<user_session_pubkey1, tee1_session_pubkey>, user_sign_privkey)>)
        , Out(<'ke_client_response', user_session_pubkey2,
              sign(<user_session_pubkey2, tee2_session_pubkey>, user_sign_privkey)>)
        ]


/*
 * Last step for TEE to verify msg3
 * Takes msg 3, verifies it and outputs the session
 * This is required for the TEE to check the last nonce (tee_pubkey serves as nonce)
*/
rule ke_msg_3_tee_establish:
    let
        sessionkey = user_session_pubkey^tee_session_privkey
    in
        [
          // Take client msg_3 and object with TEE session keys (fron rule ke_msg_2)
          In(<'ke_client_response', user_session_pubkey, signature>)
        , TEESessionKeys($tee, tee_session_pubkey, tee_session_privkey, user_sign_pubkey)
        ]
        --[
            // Ensure that the signature contains the expected elements and is signed by the user's
            //  pubkey encountered in first message
            Eq(verify(signature, <user_session_pubkey, tee_session_pubkey>, user_sign_pubkey), true)

            // Action that TEE is established and has a session now
        ,   TEE_Established($tee, user_sign_pubkey, sessionkey)
        ,   TEE_Session($tee, user_sign_pubkey)

        ,   KE_MSG_3_VERIFY($tee, user_sign_pubkey, tee_session_pubkey)
        ]->
        [
          // Output TEE session
          !Session_TEE($tee, sessionkey, user_sign_pubkey)
        ]



/*
 * ============
 * Lemmas
 * ============
 *
 * Key Exchange Lemmas:
 * - Functionality test / sanity check: Are all rules reachable and is the agreed key the same for both parties?
 * - Key secrecy test - Can adversary learn the key?
 *
*/

/*
 * We restrict our attention to traces where all equality and inequality checks succeed.
*/
restriction Equality_Checks_Succeed: "All x y #i . Eq(x,y) @ i ==> x = y"
restriction Inequality_Checks_Succeed: "All x #i. Neq(x,x) @ #i ==> F"
restriction TEEs_Unique:
  "
  All TEE #i #j .
      CreateTEE(TEE) @ #i
    & CreateTEE(TEE) @ #j
    ==>
    #i = #j
  "

/*
 * Functional lemma to check sanity of the rules.
 * 1) Check if all rules are reachable (exists-trace)
*/
lemma ke_functional_reachable: exists-trace
  "
  // Functional lemma: The last rule is reachable
  Ex U_pk TEE1 TEE2 n1 n2 gb1 gb2 #i #j #k #l #m #n .
      KE_MSG_1(TEE1, TEE2, U_pk, n1, n2) @ #i
      & KE_MSG_2(TEE1, U_pk, n1) @ #j
      & KE_MSG_2(TEE2, U_pk, n2) @ #k
      & KE_MSG_3(TEE1, TEE2, U_pk, n1, n2, gb1, gb2) @ #l
      & KE_MSG_3_VERIFY(TEE1, U_pk, gb1) @ #m
      & KE_MSG_3_VERIFY(TEE2, U_pk, gb2) @ #n
      & i < j
      & i < k
      & j < l
      & k < l
      & l < m
      & l < n
      & (not Ex #o . REVEAL_HAPPENED() @o)
  "

/*
 * Functional lemma to check sanity of the rules
 * 2) Check if key exchange calculates same key for both parties
 * This is an agreement lemma: It is possible that when U establishes a sessionkey
 * with TEE1 and TEE2, they agreed on the same key (pairwise)
 * This is a functional test, so it is enough to test if there EXISTS a trace.
 */
lemma ke_functional_agreement: exists-trace
  "
  Ex U_pk TEE1 TEE2 user_k1 user_k2 tee1_k tee2_k #i #j #k.
```

```
      User_Established(U_pk, TEE1, user_k1, TEE2, user_k2) @ #i
    & TEE_Established(TEE1, U_pk, tee1_k) @ #j
    & TEE_Established(TEE2, U_pk, tee2_k) @ #k
    & user_k1 = tee1_k
    & user_k2 = tee2_k
    & (not Ex #o . REVEAL_HAPPENED() @#o)
  "

/*
 * First security lemma:
 * Whenever a user agrees on a key with a TEE, they agreed on the same key.
 * This is the all-traces equivalent of the previous lemma with the addition of a key reveal
 *  (Which would prevent the same key)
*/
lemma ke_correct_agreement: all-traces
  "
  // Agreement lemma: Whenever U establishes a sessionkey with TEE1 and TEE2, they agreed on the same key (pairwise).
  // This is the ALL-TRACES version of the second part in the previous lemma and is bound to the requirement that
  //  the respective TEE is not compromised (otherwise the attacker can easily fake messages)
  (All U_pk TEE1 TEE2 user_k1 user_k2 tee1_k tee2_k #i #j #k .
      User_Established(U_pk, TEE1, user_k1, TEE2, user_k2) @ #i
    & TEE_Established(TEE1, U_pk, tee1_k) @ #j
    & TEE_Established(TEE2, U_pk, tee2_k) @ #k
    ==>
    // Since this is an all-traces check, the key agreement only works on non compromised TEEs:
      (
        ( // User and TEE1 have the same key if it was not compromised before the KE
          user_k1 = tee1_k
        | (Ex #l . TEE_KEY_REVEAL(TEE1) @ #l & l < j)
        )
        &
        ( // User and TEE2 have the same key if it was not compromised before the KE
          user_k2 = tee2_k
        | (Ex #l . TEE_KEY_REVEAL(TEE2) @ #l & l < k)
        )
      )
  )
  "

/*
 * Security property:
 * If a user has established a session with two TEEs, the adversary will not know
 *   BOTH keys as long as one stays uncompromised.
*/
lemma ke_secure: all-traces
  "
  All U_pk TEE1 TEE2 k1 k2 #i #j #k .
    // For all users and TEEs that established a key
    (  User_Established(U_pk, TEE1, k1, TEE2, k2) @ #i
    &  TEE_Established(TEE1, U_pk, k1) @ #j
    &  TEE_Established(TEE2, U_pk, k2) @ #k
    )
    ==>
    (
      // The adversary does not know AT LEAST ONE key, or TWO key reveals happened
      not (Ex #j . K(k1) @ #j)
    | not (Ex #k . K(k2) @ #k)
    | (Ex #o #p . TEE_KEY_REVEAL(TEE1) @ #o & TEE_KEY_REVEAL(TEE2) @ #p)
   )
  "

end
```

## B.3 Tamarin model of the Combined TEE random number generation

**Listing B.3:** Full Tamarin Combined TEE model for random number generation

```
/*
Combined TEE
This file defines the randomness functions.
As it is separate from the KE rules, it has a rule that creates secret
 session keys between a user public key and two TEEs.
The session keys are unknown to the adversary (until leaked).
*/


theory CombinedTEE_Random
begin

builtins: diffie-hellman, signing, symmetric-encryption, hashing

// TEEs have a secret quote key that the adversary can't access without a key reveal
functions: quote_key/1 [private], fst/1, snd/1

equations:
  fst(<x.1, x.2>) = x.1
, snd(<x.1, x.2>) = x.2


/*
 * Rule to create a TEE. This generates a new key based on the TEE name.
 * The quoting key bases on quote_key which is a private function not accessible by the adversary.
 * As such the quote_key stays hidden from the attacker until he performs a key reveal.
*/
rule register_tee_key:
    let
        pubkey = pk(quote_key($tee))
    in
    [  ]
    --[ CreateTEE($tee) ]->
    [ !TEEIdentity($tee, pubkey)
    , Out(pubkey)
    ]


/*
 * Rule to establish a shared key between a user and two TEEs.
 * This is basically wrapping the file combined_tee_key-exchange.spthy into one rule.
*/
rule establish_session:
    let
        user_sign_pubkey = pk(~user_sign_privkey)
    in
      [
        // Take in TEE objects
        !TEEIdentity($tee1, tee1_pubkey)
      , !TEEIdentity($tee2, tee2_pubkey)

        // And generate new keys
      , Fr(~user_sign_privkey)
      , Fr(~sessionkey1)
      , Fr(~sessionkey2)
      ]
      --[
          // Must be two different TEEs
          Neq($tee1, $tee2)
          // User and TEE actions
      ,   User_Established(user_sign_pubkey, $tee1, ~sessionkey1, $tee2, ~sessionkey2)
      ,   TEE_Established($tee1, user_sign_pubkey, ~sessionkey1)
      ,   TEE_Established($tee2, user_sign_pubkey, ~sessionkey2)
      ]->
      [
        // Create user object
        !UserIdentity($user, ~user_sign_privkey, user_sign_pubkey)
      , Out(user_sign_pubkey)
        // Create session objects
      , !Session_User(user_sign_pubkey, $tee1, ~sessionkey1, $tee2, ~sessionkey2)
      , !Session_TEE(user_sign_pubkey, $tee1, ~sessionkey1)
      , !Session_TEE(user_sign_pubkey, $tee2, ~sessionkey2)
      ]


/*
 * Attacker rule that allows the adversary to combine two randoms to a combined random.
 * This is a trivial hashing operation, but we need to explicitly tell the attacker how to do it.
*/
rule combine_randoms:
    let
      random_combined = h(<random1, random2>)
    in
      [
        In(random1)
      , In(random2)
      ]
      --[

      ]->
      [
        Out(random_combined)
      ]


/*
 * Key reveal rules
 * We have two key reveals:
 * 1) Revealing the long term TEE key (Compromise of Integrity)
 * 2) Revealing the session key (Compromise of TEE confidentiality)
 * While 1 has consequences to forward secrecy, 2 only affects already established sessions.
*/
rule reveal_tee_key:
  let
    privkey = quote_key($tee)
  in
    [ !TEEIdentity($tee, tee_pubkey) ]
  --[ TEE_KEY_REVEAL($tee)
    , TEE_REVEALED($tee)
    , REVEAL_HAPPENED() ]->
    [ Out(privkey) ]

rule reveal_session_key:
    [ !Session_TEE(user_sign_pubkey, $tee, sessionkey) ]
  --[ SESSION_KEY_REVEAL($tee, user_sign_pubkey, sessionkey)
    , TEE_REVEALED($tee)
    , REVEAL_HAPPENED() ]->
    [ Out(sessionkey) ]



/*
 * ========================
 * Random Number Generation
 * ========================
 *
 * The user requests a random number from the TEE
 *
*/

rule rng_request:
    [ !UserIdentity($user, user_sign_privkey, user_sign_pubkey)
    , !Session_User(user_sign_pubkey, $tee1, sessionkey1, $tee2, sessionkey2)
    , Fr(~nonce1)
    , Fr(~nonce2)
    ]
    --[ RNG_Request(user_sign_pubkey, $tee1, ~nonce1)
    ,   RNG_Request(user_sign_pubkey, $tee2, ~nonce2)
    ]->
    [ // Outputs to TEEs and a request state
      Out(<'rng_request', senc(~nonce1, sessionkey1)>)
    , Out(<'rng_request', senc(~nonce2, sessionkey2)>)
    , UserRNGRequest(user_sign_pubkey, $tee1, $tee2, ~nonce1, ~nonce2)
    ]



rule rng_response:
  let
    decrypted_nonce = sdec(nonce, sessionkey)
  in
    [ !Session_TEE(user_sign_pubkey, $tee, sessionkey)
    , In(<'rng_request', nonce>)
    , Fr(~random)
    ]
    --[
        // Response actions for in and output of this rule
        RNG_Response_In(user_sign_pubkey, $tee, decrypted_nonce)
      , RNG_Response_Out($tee, user_sign_pubkey, ~random)
    ]->
    [ Out(<'rng_response', senc(<~random, decrypted_nonce>, sessionkey),
          sign(<~random, decrypted_nonce>, quote_key($tee))>)
    ]


```

```

rule rng_complete:
  let
    random1 = fst(sdec(msg1, sessionkey1))
    nonce1  = snd(sdec(msg1, sessionkey1))
    random2 = fst(sdec(msg2, sessionkey2))
    nonce2  = snd(sdec(msg2, sessionkey2))
    random_combined = h(<random1, random2>)
  in
    [ !UserIdentity($user, user_sign_privkey, user_sign_pubkey)
    , !Session_User(user_sign_pubkey, $tee1, sessionkey1, $tee2, sessionkey2)
    , !TEEIdentity($tee1, tee1_pubkey)
    , !TEEIdentity($tee2, tee2_pubkey)
    , UserRNGRequest(user_sign_pubkey, $tee1, $tee2, nonce_original1, nonce_original2)
    , In(<'rng_response', msg1, quote1>)
    , In(<'rng_response', msg2, quote2>)
    ]
    --[
        // Check if nonces are correct
        Eq(nonce1, nonce_original1)
      , Eq(nonce2, nonce_original2)

        // Ensure that the Quote pubkeys come from correct quote_keys (as in, are not generated by the adversary)
      , Eq(tee1_pubkey, pk(quote_key($tee1)))
      , Eq(tee2_pubkey, pk(quote_key($tee2)))

        // Ensure that the Quotes contain the expected contents and are signed with the quote keys
      , Eq(verify(quote1, <random1, nonce1>, tee1_pubkey), true)
      , Eq(verify(quote2, <random2, nonce2>, tee2_pubkey), true)

        // Ensure that we talk two two different TEEs
      , Neq($tee1, $tee2)

        // And finally, we store the state that we have two random responses and generated a combined randomness
      , RandomResponseReceived($tee1, user_sign_pubkey, random1)
      , RandomResponseReceived($tee2, user_sign_pubkey, random2)
      , RandomGenerated($tee1, $tee2, user_sign_pubkey, random_combined)
     ]->
    [ RandomNumber($tee1, $tee2, user_sign_pubkey, random_combined)
    ]

/*
 * ============
 * Lemmas
 * ============
 *
*/

/*
 * We restrict our attention to traces where all equality checks succeed.
*/
restriction Equality_Checks_Succeed:   "All x y #i . Eq(x,y) @ i ==> x = y"
restriction Inequality_Checks_Succeed: "All x #i. Neq(x,x) @ #i ==> F"
restriction TEEs_Unique:
  "
  All TEE #i #j .
```

```
      CreateTEE(TEE) @ #i
    & CreateTEE(TEE) @ #j
    ==>
    #i = #j
  "

/*
 * Source lemma to restrict nonce use by adversary
 *
*/

lemma random_types [sources]:
  "
  (All U_pk TEE n #i .
    RNG_Response_In(U_pk, TEE, n) @i
    ==>
    ( ( Ex #j . KU(n) @j & j < i)
    | ( Ex #j . RNG_Request(U_pk, TEE, n) @j)
    )
  )
  "



/*
 * Random Number Generation
 * - Functional sanity test: Is Random Number reachable?
 * - Security: Does adversary know the generated number?
*/

lemma rng_functional: exists-trace
  "
    Ex U_pk TEE1 TEE2 sk1 sk2 n1 n2 r r1 r2 #i #j #k #l #m .
       // All three instances are established simultaneously in this model
       User_Established(U_pk, TEE1, sk1, TEE2, sk2) @ #i
       & TEE_Established(TEE1, U_pk, sk1) @ #i
       & TEE_Established(TEE2, U_pk, sk2) @ #i
       // User requested a random number
       & RNG_Request(U_pk, TEE1, n1) @#j
       & RNG_Request(U_pk, TEE2, n2) @#j
       // TEE1 responded
       & RNG_Response_In(U_pk, TEE1, n1) @#k
       & RNG_Response_Out(TEE1, U_pk, r1) @#k
       // TEE2 responded
       & RNG_Response_In(U_pk, TEE2, n2) @#l
       & RNG_Response_Out(TEE2, U_pk, r2) @#l
       // User received responses and constructs the result
       & RandomResponseReceived(TEE1, U_pk, r1) @#m
       & RandomResponseReceived(TEE2, U_pk, r2) @#m
       & RandomGenerated(TEE1, TEE2, U_pk, r) @#m
       // While no key reveal has happened
       & (not Ex #p . REVEAL_HAPPENED() @p)

       // Temporal ordering:
       // Request came first
       & i < j
      // Then responses
      & j < k
      & j < l
      // Then responses are received and assembled
      & k < m
      & l < m
  "

/*
 * Security of the random number:
 * From the user's perspective:
 * Whenever a user is established and has generated a random number,
 * the adversary does not know this random number (regardless of what other actions may have happened).
*/
lemma rng_secure: all-traces
  "
    All U_pk TEE1 TEE2 r sessionkey1 sessionkey2 #i #j .
      // For all established sessions with 2 TEEs and a generated random number,
      ( User_Established(U_pk, TEE1, sessionkey1, TEE2, sessionkey2) @ #i
      & RandomGenerated(TEE1, TEE2, U_pk, r) @ #j
      )
      ==>
      // The generated random number is secret ...
      ( not (Ex #k . K(r) @#k )
        // ... or BOTH TEE keys were revealed
      | (Ex #m #n .
          TEE_REVEALED(TEE1) @ #m
        & TEE_REVEALED(TEE2) @ #n )
      )
  "


end
```