SECURING MULTI-PARTY CRYPTO WALLETS

by

Raouf Rokhjavan

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2023

I dedicate this work to my parents who supported me unconditionally throughout my life.

Table of Contents

List of Tables

List of Figures

List of Abbreviations Used

Acronyms

AEP Asynchronous Exit Pointer

AEX Asynchronous Enclave Exit

API Application Programming Interface

AS Attestation Service

aTls attested TLS

BIOS Basic Input/Output System

BLS Boneh–Lynn–Shacham

CPU Central Processing Unit

DAO Decentralized Autonomous Organization

Dapp Decentralized Application

DCAP Data Center Attestation Primitives

DKG Distributed Key Generation

DNS Domain Name System

DoS Denial of Service

DTLS Datagram Transport Layer Security

ECDSA Elliptic Curve Digital Signature Algorithm

EdDSA Edwards-curve Digital Signature Algorithm

EDMM Enclave Dynamic Memory Management

EDR Endpoint Detection and Response

EPID Enhanced Privacy Identification

GB Giga Bytes

HD Hierarchical Deterministic

I/O Input/Output

IaC Infrastructure as Code

IAS Intel Attestation Service

IaT Integrated Attested Tunnel

LibOS Library OS

LTS Long Term Support

MalUser Malicious User

ME Management Engine

MITM Man In The Middle

MPC Multi-party Computation

MT/S Million Transfers per Second

MtA Multiplicative to Additive

Multi-Sig Multi-Signature

NFT Non-Fungible Token

NIC Network Interface Card

ORAM Oblivious RAM

OS Operating System

PFS Perfect Forward Secrecy

PSW Platform Software

QE Quoting Enclave

RAM Random Access Memory

RPC Remote Procedure Call

SDK Software Development Kit

SECS SGX Enclave Control Structure

SEV Secure Encrypted Virtualization

SGX Software Guard Extension

SPOF Single Point of Failure

SR Security Requirement

SSA State Save Area

SSD Solid State Drive

SVN Software Version Number

Sys System-level Attacker

TaT Transparent Attested Tunnel

TCB Trusted Computing Base

TCP Transmission Control Protocol

TEE Trusted Execution Environment

TLB Translation Lookaside Buffer

TLS Transport Layer Security

TSS Threshold Signature Scheme

TxIn Input Transaction

TxOut Output Transaction

TX Transaction

UDP User Datagram Protocol

USB Universal Serial Bus

UTXO Unspent Transaction Output

VPN Virtual Private Network

VSS Verifiable Secret Sharing

Abstract

Multi-party Computation (MPC) has been studied for decades to enhance the security and privacy of computer systems. Threshold Signature Scheme (TSS) is considered a special form of MPC in which multiple parties participate in generating digital signatures without revealing any information related to the corresponding private key involved in the process. Due to the popularity of cryptocurrencies and blockchain in recent years, many studies have given special attention to TSS to benefit from its advantages. Avoiding a single point of failure (SPOF) is essential in any financial system; therefore, cryptocurrencies that are heavily based on the digital signature need a system similar to TSS to eliminate the SPOF, *the private key of the crypto wallet*. Even though TSS seems fabulous at first glance, the deployment of such a system is not flawless; in fact, many bugs and vulnerabilities related to TSS protocols and their implementations have been reported over the past years. In this study, we will propose security solutions based on Trusted Execution Environment (TEE) for key generation and signing in multi-party crypto wallets. We leverage TEE technology to bind a verifiable identity to a TSS party that is based on trusted hardware. It allows parties to authenticate other TSS players and prevent malicious actors from joining the protocol. For this study, first, we will discuss the published attacks against threshold Elliptic Curve Digital Signature Algorithm (ECDSA) signatures and define a threat model associated with multi-party wallets, strictly speaking, key generation and signing in the multi-party wallet. Considering the defined threat model, we will introduce the security requirements, which pave the road for designing and implementing the prototypes. We will also propose two prototypes for this research and evaluate their security through a comprehensive security analysis. Considering all the security advantages of the implemented prototype, the results indicate that the overhead incurred by our prototype is acceptable, thereby making it a feasible option for deployment in production environments.

Acknowledgements

I am deeply grateful to my supervisor, Dr. Sampalli, who patiently guided and mentored me in this project despite all my mistakes.

Chapter 1

Introduction

After the economic crisis in 2008, Bitcoin [1] was introduced as a decentralized append-only ledger to provide a trustless model secured by replications and incentives. Satoshi Nakamoto, the inventor of Bitcoin, stated in the mailing list of Bitcoin, "The root problem with conventional currency is all the trust that's required to make it work. The central bank must be trusted not to debase the currency, but the history of fiat currencies is full of breaches of that trust." [2] As he mentioned, the lack of trust in central entities needs to be replaced with a distributed trustless system. Bitcoin was the first system that delivered such features, and it laid the foundation of blockchain technology. If there is a single trusted party, we do not need a blockchain, but in reality, in many systems like financial systems, there is no trusted player because all players try to gain the most from the system. Other blockchain platforms like Ethereum [3] utilized Bitcoin's trustless model to invent a programmable public environment or the blockchain computer on which users can develop applications known as smart contracts. From 2017 to 2022, many decentralized applications (DApp) emerged on blockchain based on the new programming paradigm like Decentralized Finance (DeFi) [4], Non-Fungible Tokens (NFT) [5], and Decentralized Autonomous Organizations (DAO) [6]. In other words, a DApp on the blockchain allows us to build an auction without auctioneers, an E-voting system without tallying authorities, a trading platform without a broker, etc. One of the best use cases of blockchain technology is the storage of value which is also known as cryptocurrency. All transactions are committed on the blockchain in a distributed and trustless model. Despite a big plunge in the cryptocurrency market ($3 trillion all-time high), the total market capitalization of the cryptocurrencies was around $1.1 trillion in August 2022. It means 2.5% of the U.S. equity market, the biggest equity market in the world. Considering all use cases and market capitalization of cryptocurrencies, we can imagine that blockchain technology will stay in financial markets for a long time [7].

Blockchain keeps track of all transactions in a sequence of blocks which are connected through the cryptographic hash. Strictly speaking, each block holds a list of transactions (Tx) that are secured by digital signatures. Digital signature provides 3 essential functionalities of authentication, integrity, and non-repudiation for each transaction. For instance, Bitcoin implements Unspent Transaction Output (UTXO) model to manage account balances; accordingly, input transaction (TxIn) and output transaction (TxOut) entries in UTXO model are related to each other utilizing digital signature [1]. Considering the volume of transactions on running blockchains, the most common signing scheme are ECDSA secp256k1, Ed25519, and Schnorr's signing scheme. Recent blockchain projects like Ethereum 2.0 have started utilizing BLS (Boneh–Lynn–Shacham) signature scheme[8] which is a more efficient option for blockchain technology.

As mentioned earlier, immutability is a massive advantage of blockchain technology which makes it a reliable ledger containing all transactions without being tainted. On the other hand, it raises a grave concern; what if a hacker can access the private key of an account and sign transactions on behalf of the real user? The answer is that stolen crypto assets cannot be restored because transactions on the blockchain are irreversible. In this case, SPOF is a serious security concern for blockchain technology. TSS is a technique that enables a group of players to sign a transaction collaboratively without revealing the private key. In TSS signatures, (t, n) in which $1 \leq t < n$ means n parties having key shares, and $t + 1$ of them can issue the digital signature together. It means that TSS can eliminate the risk of SPOF in traditional crypto wallets. Taking this into account, multi-party crypto wallets leverage TSS protocols to distribute the rights of signing blockchain transactions among multiple parties like different persons, teams, software platforms, hardware modules, etc.; consequently, it mitigates the risk of external and internal attacks. This technology can reassure the customers of crypto custody and financial institutions who transfer billions of dollars in crypto assets because their customers do not need to trust the institutions completely to do transactions on their behalf. Multi-Signature Scheme (Multi-Sig) provides the same functionality, but TSS is more flexible, private, and efficient than Multi-Sig. Furthermore, to enhance the security of TSS, TSS implementations support the resharing feature in which parties update their key shares

periodically without changing their corresponding actual address, the public key. All the abovementioned advantages make the multi-party wallet a perfect solution for signing transactions on the blockchain.

TSS is not a new concept, and it was introduced more than 30 years ago, but blockchain applications and their growing market cap incentivized researchers to find faster and more efficient protocols. Even though TSS provides many benefits, the complexity and lack of maturity in its implementations can be the Achilles' heel of this valuable technology [9].

1.1 Motivation

To store and keep cryptocurrencies, we need a crypto wallet. Generally speaking, crypto wallets are divided into hot and cold wallets. A hot wallet is online and connected to the public network. While it is fast and easy to use, because of connecting to the Internet, it is vulnerable to online attacks and stealing funds. In contrast, the cold wallet is typically not connected to the Internet which makes it less flexible but more secure. It raises a legitimate question which one is the best option? Most giant crypto exchanges and custodians choose a hybrid solution; in fact, they keep most of their crypto assets in cold wallets and store only a certain amount of them in hot wallets to fulfill the withdrawal requests of customers [10]. Most cold wallets are like special-purpose USB devices, and because the private key does not leave them, malicious code on compromised computers cannot sign a transaction on their behalf. It provides a huge security advantage; however, the high security in cold wallets sacrifices ease of use and assessability which are essential for crypto custodians and market adoption.

TSS can be a helpful technique for crypto custodians. It not only allows crypto custodians to have the versatility of hot wallets but also enhances their overall security by avoiding SPOF. Additionally, not having SPOF can partially protect crypto exchanges from losing crypto funds and Denial of Service (DoS) attacks through the loss of signing capability. In the case of cold wallets, air-gapped or not, TSS distributes trust among multiple teams to complete a transaction; hence, TSS is a perfect choice to mitigate the risk of both inside and outside attacks on wallets cryptographically.

(a) TSS Hot Wallet

(b) TSS Cold Wallet

Figure 1.1: TSS Hot and Cold Wallet

As shown in Figure 1.1a, in a hot wallet supporting TSS, the K_u and K_e are generated without revealing any information related to the other party. Each transfer order must be signed by the two-party signing protocol to be valid. In the cold wallet with TSS (Figure 1.1b), after generating K_a, K_b, and K_c key shares collectively, the $(1, 3)$ scheme is deployed which means 2 out of 3 parties can sign transactions.

ECDSA is the common signing scheme in cryptocurrencies, but due to ECDSA properties, the threshold ECDSA was inherently too slow, but over the past years, threshold ECDSA could make dramatic improvements. In 2018, Gennaro and Goldfeder [11] proposed an efficient threshold ECDSA in the dishonest majority environment; then, they improved this protocol [12] to identify aborts with one-round online signing. Due to their efficacy, the GG protocols [11, 12] are adopted as threshold ECDSA protocols in the industry [13, 14, 15]. Threshold ECDSA uses some cryptographic primitives and protocols which are non-standard and complex like verifiable secret sharing (VSS), homomorphic encryption, commitments, and zero-knowledge proofs. The complexity and the lack of mature implementations have introduced some bugs in the implementations of threshold ECDSA protocols [16, 9]. These vulnerabilities have shown us that despite all advantages of threshold ECDSA, we need to add preventive measures to mitigate any potential risks of exploiting the protocol, weak assumptions, and bad implementations. In theory, threshold TSS must be resilient against protocol deviation and bad input, but in practice, a malicious party could exploit vulnerabilities by changing the expected behavior. What if we could attest to the trustworthiness of TSS players before allowing them to join the protocol? This

can be achieved by using a TEE.

Since launching the 6th generation of Intel Core processor, in 2015, Intel has made available Software Guard Extension (SGX) [17] technology. This technology was an effort to provide a TEE or enclave in SGX vocabulary. TEE enables developers to protect their applications from high-privileged software like operating systems (OS) and hypervisors running on the same hardware. Over the past years, this hardware feature has been used to shield many applications and network services [18, 19, 20, 21]. The drawback of utilizing SGX to protect applications is that the developer must use SGX Software Development Kit (SDK) in its code; in other words, the code base must be modified which incurs huge costs. Library OSes (LibOS) like Gramine [22] can solve this problem. In fact, Gramine is a LibOS that is tailored to support SGX and protect unmodified code from the local privileged attacker [23]. On the other hand, attestation is an integral part of TEE and SGX in particular in which the attester tries to convince the challenger that it is a trusted enclave running on TEE platform. At the end of this process, the challenger can trust the attester based on provided values during the attestation. Therefore, if an untrusted client wants to communicate with a server, the server can accept or refuse the connection based on the attestation result.

Regarding the aforementioned threshold ECDSA attacks, the attacker runs an ingenuine instance of threshold ECDSA and tries to exploit the software or protocol vulnerabilities. In practice, the attacker can inject its malicious code into a TSS code [24]. To enhance the security of the threshold ECDSA implementations, each player should authenticate other participating players in the TSS protocol. One solution would be authentication at the network (IP/Port) or system level, but the attacker can evade these preventive measures. To solve this problem, we have to not only protect each threshold ECDSA party from system-level attacks but also authenticate the traffic generated by each player. Both of these goals are attainable through TEE and integrating authentication in the attestation process. According to this notion, in this research, we utilize TEE and remote attestation to secure the threshold ECDSA protocols used in multi-party crypto wallets.

1.2 Contribution

This research aims to mitigate the attacks on threshold ECDSA protocols through TEE and attested tunnels. According to this notion, our contributions are as follows:

- We define a threat model for multi-party wallets based on key generation and signing protocol. Then according to the defined model, we present their security requirements related to multi-party crypto wallets in our system.
- Given the security requirements, we develop prototypes that protect TSS players from known attacks and prevents adversarial parties from participating in TSS protocol. In fact, we provide a mutual authentication mechanism based on remote attestation to provide a fully honest environment for threshold ECDSA protocols.
- We present a comprehensive security analysis regarding our prototypes.
- We propose a framework to deploy and update honest parties in the TSS protocol.
- We evaluate the overhead and performance of implemented prototypes.

Figure 1.2 depicts the workflow of our research. We found that despite the vast number of research related to TSS, there is a gap regarding practical solutions to mitigate attacks against TSS in general and threshold ECDSA in particular. To capture the development of this field over the past years, we conducted an in-depth literate survey. The literature survey explores research work in four domains: 1)TSS and Threshold ECDSA, 2)Threshold ECDSA Attacks, 3)TEE and Intel SGX. During the initial part of the literature survey, we study TSS based on different signature algorithms like ECDSA, BLS, and Schnorr. Since most cryptocurrencies use ECDSA for signing transactions, threshold ECDSA is at the center of attention. Due to its characteristics, generating ECDSA signatures collectively has serious challenges, so we primarily focus on [11], and [12] protocols which are widely implemented in the industry [14, 13, 15] because of their unique features; however, introduced concepts apply to other protocols. In the next part of the literature review, we examine the conducted attacks against threshold ECDSA [9, 16]. According to our findings, the

root of attacks on threshold ECDSA is the complexity of cryptographic primitives and TSS protocols, plus the lack of standard implementations. Strictly speaking, an adversarial player involved in a protocol can exploit these vulnerabilities and forge the signature on his own. The proposed threshold ECDSA protocols were designed to work in a dishonest majority environment where adversaries are less than $t+1$; however, dishonest players can game the protocol and forge the signature while they are less than $t+1$. In light of this fact, we decided to establish a fully honest environment for threshold ECDSA to mitigate TSS attacks. Hence, we studied TEE technology, Intel's SGX notably. To make the proposed solution applicable for implemented applications using threshold ECDSA, we deploy a LibOS which supports Intel's SGX; in fact, it allows us to run legacy code on SGX without modification. We reviewed available LibOSes compatible with SGX like Gramine [22], Scone [25, 26], and Occulum [27] and chose Gramine for our work. To establish a fully honest environment, we opt for ra-tls [28] which combines mutual authentication, remote attestation, and secure channel. Finally, we defined a threat model for multi-party crypto wallets.

In the next phase of our research, we define a threat model for the TSS environment and extract security requirements that must be met. Considering the threat model and security requirements, we design a prototype for threshold ECDSA where players establish secure attested tunnels for mutual authentication. We implemented a prototype based on LibOS and Datagram Transport Layer Security (DTLS) to establish secure attested tunnels between TSS client applications transparently. To implement the second prototype, we integrate tss-lib, an industry-accepted TSS library, with the remote attestation process to provide an attested tunnel.

To evaluate the efficacy of our prototypes, we implemented a benchmark frame to conduct micro and macro-benchmarks. We executed benchmarks on three execution environments, native execution, gramine execution running only on LibOS, and tunnel utilizing an attested tunnel. Regarding the first prototype, we benchmark the network throughput of unmodified iperf3 [29] and Distributed Key Generation (DKG)/Signing latency of ZenGo-X threshold ECDSA implementation [14]. For the second prototype, we measured the duration of key generation and signing phases in three environments; native, gramine, and tunnel. We also propose a micro-benchmark in which we diagnose different steps of key generation and signing to find the source

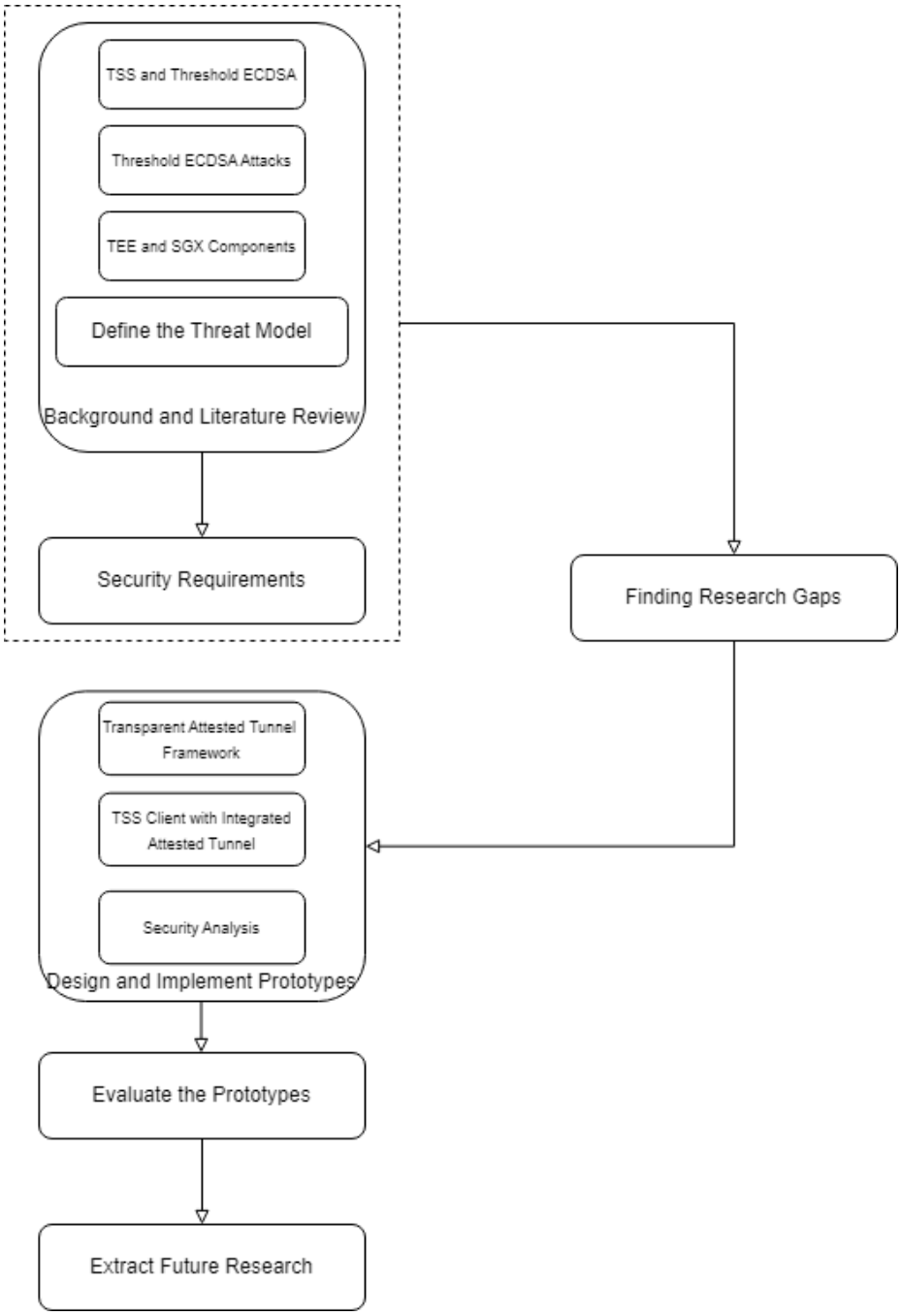


Figure 1.2: Research Workflow

of overhead.

1.3 Organization of the Thesis

The following chapters are as follows:

Chapter 2 covers backgrounds and related studies. In this chapter, we review the required definitions and primitives of TSS; additionally, we explain DKG and signing phases in threshold ECDSA protocols. Then we delve into TEE, SGX technology, and its essential component, remote attestation. We also present research on threshold ECDSA attacks. We conclude the chapter by introducing the threat model of multi-party crypto wallets with threshold ECDSA protocols.

Chapter 3 focuses on the methodology of our research. First, we introduce the security requirements for multi-party crypto wallets in this study based on the given threat model. In the following, we illuminate the overall design of the transparent and integrated attested tunnels. Then we will go through the details of their implementation and how they can be deployed in the real world. In the end, we analyze the security of prototypes regarding security requirements.

Chapter 4 presents a benchmark framework and benchmarks to evaluate the implemented prototypes. We benchmark the network throughput, key generation, and signing latency in different scenarios for the transparent attested tunnel. Regarding the integrated attested tunnel, we measure key generation and signing latency as macro-benchmarks similarly; additionally, we add micro-benchmarks to understand more about the TSS client execution in an enclave. For each benchmark, we will discuss the results and their implications.

Chapter 5 summarizes the research with remarks, limitations, and future work.

Chapter 2

Background Knowledge and Literature Review

In the initial segment of this chapter, we demonstrate the building blocks of threshold ECDSA signing utilized in most multi-party wallets; additionally, we delve into DKG and threshold ECDSA signing protocols widely accepted in the crypto industry. After understanding foundations, we explain some published attacks against threshold ECDSA signing protocols. Subsequently, we introduce TEE technology and its features that we incorporate into our research. Finally, we outline the threat model associated with multi-party wallets that we address in our study.

2.1 Threshold ECDSA Signing

2.1.1 Introduction

TSS allows multiple parties to generate a digital signature collectively in which they cannot know about the private key. In a (t, n) threshold signing, $t + 1 \leqslant n$ can issue a valid signature, while any subset of parties less than or equal to t cannot. After 30 years from introducing threshold signing [30] and subsequent protocols [31, 32, 33, 34, 8], blockchain caused a new interest in TSS which led to a lot of research in this domain [35, 11, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 12]. These studies focus on ECDSA and Edwards-curve Digital Signature Algorithm (EdDSA) signatures which are used in blockchain protocols.

As mentioned earlier, in this research, we concentrate on the ECDSA signing algorithm that is extensively utilized in blockchains and crypto wallets. Due to inherent ECDSA properties, designing threshold signing is more complicated, so many papers have been published to enhance its security and performance. In the following, we first introduce various cryptographic building blocks that we need to understand threshold ECDSA signing in general and [11] and [12] in specific which gained considerable acceptance in the crypto industry. Moreover, we elucidate the metrics used

to compare TSS protocols, and ultimately, we consolidate all these aspects in the context of DKG and signing protocols.

2.1.2 Cryptographic Building Blocks

Definition 1 - Elliptic Curve Digital Signature Algorithm (ECDSA) [46] The ECDSA signature scheme (G, S, V) uses the group of points $\mathbb{G}$ of an elliptic curve over a finite field $\mathbb{F}_p$. Let g be a generator of $\mathbb{G}$ and let q be the order of the group $\mathbb{G}$, which we assume is prime. The hash function H is defied over $(\mathcal{M}, \mathbb{Z}^*{}_q)$.

The scheme works as follows:

- $G()$: Choose $\alpha \xleftarrow{\mathrm{R}} \mathbb{Z}^*_q$ and set $u \leftarrow g^\alpha \in \mathbb{G}$. Output $\mathsf{sk} := \alpha$ and $\mathsf{pk} := u$.
- $S(\mathsf{sk}, m)$: To sign a message $m \in \mathcal{M}$ with secret key $\mathsf{sk} = \alpha$ do:

  Repeat :

  $\alpha_t \xleftarrow{\mathrm{R}} \mathbb{Z}^*{}_q, u_t \leftarrow g^{\alpha_t}$

  let $u_t = (x, y) \in \mathbb{G}$ *where* $x, y \in \mathbb{F}_p$

  treat x as an integer in $[0, p)$ and set $r \leftarrow [x]_q \in \mathbb{Z}_q$ - reduce x to mod q

  $s \leftarrow (H(m) + r\alpha)/\alpha^t \in \mathbb{Z}_q$

  until $r \neq 0$ *and* $s \neq 0$

  output $(r, s) \in \mathbb{Z}_q^2$

- $V(pk, m, \sigma)$: To verify a signature $\sigma = (r, s) \in \mathbb{Z}_q^2$ on $m \in \mathcal{M}$ with $pk = u \in \mathbb{G}$ do:

  if $r = 0$ *or* $s = 0$ *then output reject and stop*

  $\alpha \leftarrow H(m)/s \in \mathbb{Z}_q,\ b \leftarrow r/s \in \mathbb{Z}_q$

  $\hat{u}_t \leftarrow g^a u^b \in \mathbb{G}$

  if $\hat{u}_t$ *is the point at infinity in* $\mathbb{G}$ *then output reject and stop*

  let $\hat{u}_t = (\hat{x}, \hat{y}) \in \mathbb{G}$ *where* $\hat{x}, \hat{y} \in \mathbb{G}$

  treat $\hat{x}$ *as an integer in* $[0, p)$ *and set* $\hat{r} \leftarrow [\hat{x}]_q \in \mathbb{Z}_q$ - reduce $\hat{x}$ to mod q

  if $r = \hat{r}$ *output accept; else output reject*

Definition 2 - Commitment scheme [46] A commitment scheme for a finite message space $\mathcal{M}$, is a pair of efficient algorithms $C = (C, V)$ where:

- Algorithm C is invoked as $(c, o) \xleftarrow{\mathrm{R}} C(m)$, where $m \in \mathcal{M}$ is the message to be committed, c is the commitment string, and o is an opening string.
- Algorithm V is a deterministic algorithm invoked as $V(m, c, o)$ and outputs $acceptorreject$

Correctness Property: if $(c, o) \xleftarrow{\mathrm{R}} C(m) \iff \Pr[V(m, c, o) = accept] = 1$
Hiding Property: The commitment c does not reveal information about m.
Binding Property: An adversary $\mathcal{A}$ cannot find (m', c') in which $(m', c') \neq (m, c)$ and $V(m', c', o) = accept$.

Definition 3 - Additive Homomorphic Encryption [47] An additively homomorphic encryption scheme consists of three algorithms KGen, $\mathsf{Enc_{pk}}$, $\mathsf{Dec_{sk}}$ and two operations.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}$ generates public/private keys.
- $\mathsf{Enc_{pk}} : \mathcal{M} \rightarrow \mathcal{E}$ is a probabilistic algorithm.
- $\mathsf{Dec_{sk}}\mathcal{E} \rightarrow \mathcal{M}$ is a deterministic algorithm.
- $\oplus$ and $\odot$ are its two operations in which

$$\begin{aligned} &\oplus : \mathcal{E} \times \mathcal{E} \rightarrow \mathcal{E} \ / \ m_1 + m_2 = \mathsf{Dec_{sk}}(\mathsf{Enc_{pk}}(m_1) + \mathsf{Enc_{pk}}(m_2)) \\ &\odot : \mathbb{Z} \times \mathcal{E} \rightarrow \mathcal{E} \ / \ k.m = \mathsf{Dec_{sk}}(k \odot \mathsf{Enc_{pk}}(m)) \end{aligned} \tag{2.1}$$

Paillier homomorphic encryption which is mostly used in TSS is as follows:

- KGen- Generate two large prime numbers p and q having the same length such as

$$\begin{aligned} &N = p.q \\ &\lambda = (p-1)\,(q-1) \\ &\mathsf{sk} = \lambda \ and \ \mathsf{pk} = N \end{aligned} \tag{2.2}$$

- $\mathsf{Enc_{pk}} : \mathbb{Z}_N \rightarrow \mathbb{Z}_{N^2}$ - For a message $m \in \mathbb{Z}_N$, and random number $r \in \mathbb{Z}_N^*$ (integer from 1 to N-1)

$$c = (N+1)^m r^N \ mod\ N^2 \ s.t.\ c \in \mathbb{Z}_{N^2} \tag{2.3}$$

- $\mathsf{Dec_{sk}} : \mathbb{Z}_{N^2} \rightarrow \mathbb{Z}_N$ - Define as follows

$$\begin{aligned} L(u) = (u-1)/N \ over\ all\ u \in \mathbb{Z}_{N^2} \\ m = L(c^\lambda).\lambda^{-1}\ mod\ N \end{aligned} \tag{2.4}$$

- Its homomorphic properties are:

$$\begin{aligned} &\mathsf{Enc_{pk}}(m_1) \oplus \mathsf{Enc_{pk}}(m_2) = \mathsf{Enc_{pk}}(m_1)\ .\ \mathsf{Enc_{pk}}(m_2) \\ &k \odot \mathsf{Enc_{pk}}(m) = \mathsf{Enc_{pk}}(m)^k \end{aligned} \tag{2.5}$$

Definition 4 - Zero-knowledge Proof [47] In Zero-knowledge proof, the prover proves the validity of his claim without revealing any information to the validator. The prover is responsible for providing the claim, and the validator is responsible for verifying the provided claim. In TSS protocols, zero-knowledge proofs are utilized to ensure parties follow the protocol. Zero-knowledge proofs are the most computationally demanding task in the protocol compared to other steps.

Definition 5 - Threshold Signature Scheme [46] A threshold signature scheme (G, S, V, C) is a tuple of four efficient algorithms:

- G is a probabilistic key generation algorithm that is invoked as

$$(pk, pkc, sk_1, sk_2, \ldots, sk_N) \xleftarrow{n} G(N, t) \tag{2.6}$$

  to generate a t-out-of-N shared key. It outputs a public key pk, a combiner public key pkc, and N signing key shares, $\mathsf{sk}_1, \ldots, \mathsf{sk}_N$.

- S is a (possibly) probabilistic signing algorithm that is invoked as $\sigma_i^{'} \xleftarrow{R} S(\mathsf{sk}_i, m)$, where sk_i is one of the key shares generated by G, m is a message, and $\sigma_i^{'}$ is a signature share for m using sk_i.

- V is a deterministic verification algorithm as in a signature scheme, invoked as $V(pk, m, \sigma)$ and outputs either accept or reject.

- C is a deterministic combiner algorithm that is invoked as

$$\sigma \leftarrow C(pkc, m, \mathcal{J}, \{\sigma_j^{'}\}_{j\in\mathcal{J}}) \tag{2.7}$$

where pkc is the combiner public key, m is a message, $\mathcal{J}$ is a subset of $\{1,\ldots,N\}$ of size t, and each $\sigma_j^{'}$ is a signature share for m. The algorithm either outputs a signature σ, or outputs a special message $blame(\mathcal{J}^*)$, where $\mathcal{J}^*$ is a nonempty subset of $\mathcal{J}$. Intuitively, the message $blame(\mathcal{J}^*)$ indicates that the provided signature shares $\sigma_j^{'}$ fo $j \in \mathcal{J}^*$ are invalid.

- Correctness: The verification algorithm should accept a properly constructed signature; specifically, for all possible outputs $(pk, pkc, sk_1, sk_2, \ldots, sk_N)$ of $G(N,t)$, all messages m, and all $t-size$ subsets $\mathcal{J}$ of $\{1,\ldots,N\}$ we have

$$\Pr\Big[V(pk, m, C(pkc, m, \mathcal{J}, \{\sigma_j^{'}\}_{j\in\mathcal{J}})) = accept\Big] = 1 \tag{2.8}$$

Definition 6 - Shamir Secret Sharing [48] A dealer has a secret key s , and it wants to share it among n players in such a way that 1) t players cannot recover s 2) $t+1$ players can recover s.

The dealer chooses a random polynomial in $F(x) \in Z_q[x]$ of degree t in which

- $F(0) = s$
- q is a prime number and $s \in Z_q$ (Z_q is a finite field on modulo q)
- it sends to each player P_i the share $s_i = F(i) \ mod\ q$

$t+1$ players can recover the secret by polynomial interpolation while t players have no information regarding the secret in a strong information-theoretic sense. Strictly speaking, for each secret s', there is a polynomial F' which agrees with the secret and the shares held by the adversaries (players).

$F(x)$ is a linear combination of the Lagrangian Polynomial. For a set S of $t+1$ values of s_i in which $i \in S(|S| = t+1)$, we want to find $F(x)$ of degree t such that $F(i) = s_i$, $i \in S$. To do these, we use Lagrange interpolation. Lagrangian polynomial of degree t, $\mathbf{\Lambda}_{i,S}(x)$ is defined as

$$\begin{aligned} &\Lambda_{i,\ S}(i) = 1 \text{ and } \Lambda_{i,S}(j) = 0 \text{ for } i \neq j \\ &\Lambda_{i,S}(X) = \prod_{i,j\in S, i\neq j} \frac{X-j}{i-j} \end{aligned} \tag{2.9}$$

Then the $F(X)$ is a linear combination of the Lagrangian polynomial of degree t in which both sides of the equations are polynomial of degree t which agreeing on $t+1$ points:

$$F(X) = \sum_{i \in S} s_i \mathbf{\Lambda}_{i,S}(X) \tag{2.10}$$

To find the key, we should calculate the free term or 0-Lagrangian coefficient or $\lambda_{i,S}$ because $F(\mathbf{0}) = s$. Accordingly,

$$\begin{aligned} s &= \sum_{i \in S} \lambda_{i,S} s_i \\ \lambda_{i,S} &= \Lambda_{i,S}(\mathbf{0}) = \prod_{(i,j \in S, j \neq i)} j/(j-i) \end{aligned} \tag{2.11}$$

We can do it for any j-Lagrangian coefficient associated with S, it means

$$\begin{aligned} F(j) &= s_j = \sum\nolimits_{i \in S} \lambda_{j,i,S}\ s_i \\ \lambda_{j,i,S} &= \mathbf{\Lambda}_{i,S}[j] \end{aligned} \tag{2.12}$$

Definition 7 - Verifiable Secret Sharing(VSS) [49] In a secret sharing protocol, when the dealer is not trusted, and all players follow the secret sharing protocol, VSS guarantees that each player receives a unique share in which their shares interpolate into the correct secret. **Feldman's VSS** [49] is as follows:

1. For the input secret x, the dealer

   - Chooses a random polynomial $F(X)$ of degree t that $F(0) = x$ in which $[f_0, f_1, \ldots, f_t]$ are the coefficients of $F(X)$ and $f[0] = x$.
   - Broadcasts $F_j = g^{f_j}$, the public commitment to coefficients of polynomial $F(X)$.
   - Sends the private key $F(i) = x_i$ to player i privately.

2. Player i checks that secret x_i lies in the polynomial defined by $[F_0, F_1, \ldots, F_t]$ via Evaluation in the exponent as follows:

$$\prod_{j=0}^{t} F_j^{i^j} = \prod_{j=0}^{t} g^{f_j i^j} = g^{\sum_{j=0}^{t} f_j i^j} = g^{x_i} \tag{2.13}$$

   If it does not match, each player lodges a complaint.

3. If t players have complaints, the dealer is untrusted and disqualified, and protocol will be aborted; otherwise, to find untrusted players and resolve complaints, the dealer will broadcast the shares to eliminate bad actors and find honest majorities.

Definition 8 - Multiplication of Shared Secrets Additively Assume n players have additive shares of secrets a,b in which $a = a_{\mathbf{1}} + \ldots + a_n$ and $b = b_{\mathbf{1}} + \ldots + b_n$ and player i holds a_i and b_i.

The parties compute an additive sharing of $c = ab$ in which $c = \sum_{i,j} a_i b_j$. If Parties i and j could turn $a_i b_j$ into two values d_{ij} and e_{ij} such that $d_{ij} + e_{ij} = a_i b_j$ then [50]

$$c_i = a_i b_i + \sum\nolimits_i e_{ji} + \sum\nolimits_i d_{ji} \tag{2.14}$$

Definition 9 - Multiplicative to Additive (MtA) Shares An MtA protocol allows two players Alice and Bob Who hold secrets $a, b \in Z_q$ respectively to turn them into secret $d, e \in Z_q$ respectively such that [51]

$$d + e \ = a.b \ mod \ q \tag{2.15}$$

Let E is an additive homomorphic encryption scheme. Considering the additive and scalar multiplication features of E

1. Alice sends $A = E_k(a)$ to Bob

2. Bob uses the scalar multiplication features of E and computes $E_k(a.b)$

3. Bob uses the additive feature of E, and chooses a random value m, and computes $E_k(a.b + m)$

4. Bob sends $B = E_k(a.b + m)$ to Alice in which Bob's share is $e \ = \ -m$

5. Alice decrypts B in which Alice's share is $d = D_k(B)$ because

$$d + e = (a.b + m) \ + (-m) = a.b \tag{2.16}$$

2.1.3 Distributed Key Generation

Pedersen's protocol[52] was the first DKG in the literature. In fact, Pedersen's DKG is n parallel executions of Feldman's VSS which are run by n players.

1. Each player P_i picks random polynomial $F(X) \in Z_q[X]$ of degree t.

2. Each player P_i , $1 \leq i \leq m$, runs instance of Feldman's VSS,

   (a) For $z_i \in Z_q$, the commitment to $Z_i = g^{z_i}$ is public because $z_i = F_i(0)$. (Each player P_i can verify it via interpolation in the exponent for $\prod_{j=0}^{t} F_j^{ij}$)

   (b) for z_{ij} which denotes the Shamir's share of player P_i to player P_j i.e., $1 \leq i, j \leq m$, the commitment to $z_{ij} = F_i(j)$ via $Z_{ij} = g^{z_{ij}}$. In fact, P_i plays the role of dealer, and $P_1, \ldots, P_m$ play the role of participants; hence, P_i plays a double role. Each player P_i opens its commitment to Z_i.

3. Suppose Q is the set of players which are not disqualified after running Feldman's VSS.

   (a) The key x is defined as

   $$x = F(0) = \sum_{i \in Q} F_i(0) = \sum_{i \in Q} z_i \tag{2.17}$$

   (b) Public key y is set as

   $$y = g^x = g^{\sum_{i \in Q} x_i} = \prod_{i \in Q} Z_i \tag{2.18}$$

   (c) For each player P_i,

   $$x_i = F(i) = \sum_{i,j \in Q} F_j(i) = \sum_{i,j \in Q} z_{ji} \tag{2.19}$$

   (d) For each player P_i, public key y_i is defined as

   $$y_i = g^{x_i} = g^{\sum_{i,j \in Q} z_{ji}} = \prod_{i,j \in Q} Z_{ji} \tag{2.20}$$

Note that $x_{ij} = F_i(j)$. Since $F(X) = \sum_{i \in Q} F_i(X)$, it follows that $x_i = F(i)$.

There is a problem with Pedersen's DKG; strictly speaking, it is not information-theoretically protected from adversaries. It is true that z_i is protected by Discrete Log Difficulty, but the adversaries have enough information to affect the total protocol. For this purpose, other variants of Pedersen's DKG were introduced, **Committed Pedersen's DKG** [11], **Joint-Pedersen's DKG** [53, 54].

2.1.4 Threshold ECDSA

After running the DKG protocol, all players have a share of x in a (t, n) Shamir's scheme. According to this notion, a simplified version of GG20 [12] protocol is as follows:

1. $t + 1$ players perform two additive sharings of random values k and a

$$a = a_1 + \ldots + a_k \; and \; k = k_1 + \ldots + k_n \tag{2.21}$$

   The values $A_i = g^{a_i}$ are committed with a non-malleable commitment.

2. Players performs two MtA protocols to get additive shares of $b = ka$ and $z = kx$. A few notes in this regard:

   - Each player de-commits A_i and players compute $A = g^a = \prod_i A_i$
   - Players reconstruct b by computing

$$c = inv\,(b) \;\; mod \; q \; and \; R = g^{inv(k)} = A^c \tag{2.22}$$

3. Players broadcast $s_i = k_i^{-1}(H(m) + rx_i)$ to interpolate s. Protocol aborts if the signature is not correct.

2.1.5 TSS Attacks

Threshold ECDSA signatures can significantly enhance the security of cryptocurrency wallets by protecting against losses that might occur due to a compromised system. Nevertheless, the use of TSS - and specifically threshold ECDSA - presents certain challenges. These schemes involve complex structures that are not standardized, and there are multiple different implementations available due to the intense

competition among companies seeking to establish a lead over their competitors in the cryptocurrency market [15, 13, 14, 55]. This has resulted in a rapid development cycle, with insufficient code review and audit. Given that the wallets of organizations and exchanges can hold billions of dollars, any bug in a multi-party crypto wallet could have catastrophic consequences [56]. In recent years, researchers have analyzed several multi-party wallets and TSS protocols, and they have succeeded in conducting numerous attacks against them. For example, in the Alpha-Rays attacks [16], researchers discovered two incorrect assumptions in the GG20 and GG18 protocols [11, 12], which are the most widely implemented threshold ECDSA signing protocols in the industry. The first attack relates to the MtA sub-protocol, implemented with a fast option lacking a range-proof mechanism. This assumption made it possible for an attacker to determine the number of times a victim's output was reduced modulo N, and to craft a nonce that would allow them to discover the bits of the victim's secret key. The researchers were able to extract the victim's key share using 16 signatures, claiming that it could have been done with only eight signatures. In their second attack, they exploited a vulnerability in the expensive version of the MtA sub-protocol with range proof due to the absence of size checking of the Paillier encryption key. By controlling a party, an attacker can find the key shares of all other honest parties using just one signature. The researchers claim that many public repositories were affected by this attack [14, 15]. In another paper, the authors exploited multi-party cryptocurrency wallets due to incorrect assumptions and poor TSS protocol implementation [9].

As these attacks show, most threshold ECDSA protocols are young and complex, and their rapid development in response to market competition makes them more vulnerable to similar attacks. In these attacks, the attacker seeks to take control of a system, participate in the protocol, and exploit incorrect assumptions and poor implementations. To protect multi-party wallets from these attacks, we must detect and prevent attackers and dishonest parties from joining the protocol and gaming the honest parties. This can be achieved by using TEEs. It allows the system designers to protect processes from system-level attacks and bind their identities to the hardware that is verifiable by relevant stakeholders.

2.2 Trusted Execution Environment

2.2.1 Introduction

TEE is a powerful abstraction that enables us to execute processes in isolation from the rest of the system. TEEs provide robust hardware-based protection mechanisms that ensure the integrity and confidentiality of the contained code and data; Additionally, TEEs incorporate features that facilitate the verification and validation of these security mechanisms to external entities [17].

One notable hardware-based mechanism for implementing a TEE is Intel's SGX, which is integrated into Intel's processors. SGX offers a complete isolation mechanism that allows applications to run independently of the host software stack. It does so by providing a virtual memory address space that is completely isolated from the rest of the system, thereby guaranteeing the confidentiality and integrity of the contained data. As depicted in Figure 2.1, the main promise of TEE in general and SGX in specific is that it can protect code and data at runtime from the host's software stack, even from OS and hypervisor. According to this Figure, in TEE model, the attack surface is limited to the application, system software connecting to the hardware, and the hardware. Furthermore, TEE can enhance the application's security by providing an attestation mechanism at startup and limiting the control flow to specific entry points to access the code and data.

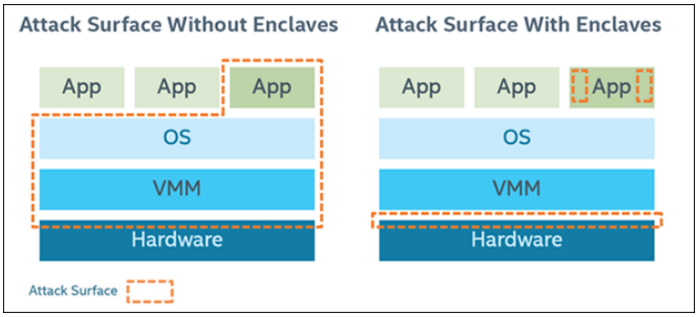


Figure 2.1: Attack Surface in TEE

2.2.2 Software Guard Extension

Intel's SGX was initially introduced in 2015 as a security mechanism that was integrated into the sixth-generation Intel Core processors, which are based on the Skylake

micro-architecture. This technology affords a protective sandbox for safeguarding enclaves against an array of threats, including those originating from the OS or hypervisor, BIOS, firmware, drivers, system management module, Intel Management Engine (ME), as well as any remote attacks. Accordingly, Intel's SGX technology provides a range of robust protections, including but not limited to the following: Firstly, the memory of the enclave is protected from external reading or writing, as it is entirely insulated from the external environment. This ensures that data isolated within enclaves can only be accessed by code that shares the enclave, thus guaranteeing its confidentiality and integrity. Secondly, if the enclave is set to a production level, it is invulnerable to software or hardware debuggers. Thirdly, entering the enclave's memory via function calls, jumps, registers, or stack manipulation is impossible, thus safeguarding the enclave from unauthorized access or manipulation. Fourthly, the memory within the enclave is encrypted using industry-standard encryption algorithms with replay protection, ensuring that the data remains secure from external threats. Finally, the memory encryption key randomly changes with every power cycle, rendering it inaccessible as it is stored within the CPU [57].

In Intel SGX architecture, an application is separated into two distinct components: a secure component and a non-secure component. The application is responsible for launching the enclave, the secure component which is an encrypted area of memory. When an enclave function is invoked, only the code within the enclave can access its data, and upon completion, the enclave data remains in the protected memory. Thus, the application comprises its own code, data, and enclave, whereas the enclave contains its own code and data, and Intel SGX safeguards the confidentiality and integrity of the enclave's code and data. Moreover, the SGX enforces pre-determined entry points for the enclave during compilation. Another important point is that while an enclave can access its application's memory, the reverse is not possible which means data within the enclave is secured from unauthorized access.

Figure 2.2 depicts the lifecycle of an enclave [58]. The application requests to load the enclave; then issues the ECREATE instruction to create and populate the SGX Enclave Control Structure (SECS). SECS is an immutable data structure containing the meta-data associated with each enclave. It is not available to secure and insecure parts of the application; instead, it is directly used by the CPU. Issuing EADD

instruction loads a page into an enclave-protected memory. Finally, EINIT finalizes the enclave creation in which INIT checks the EINITTOKEN data structure to ensure that the enclave can execute.

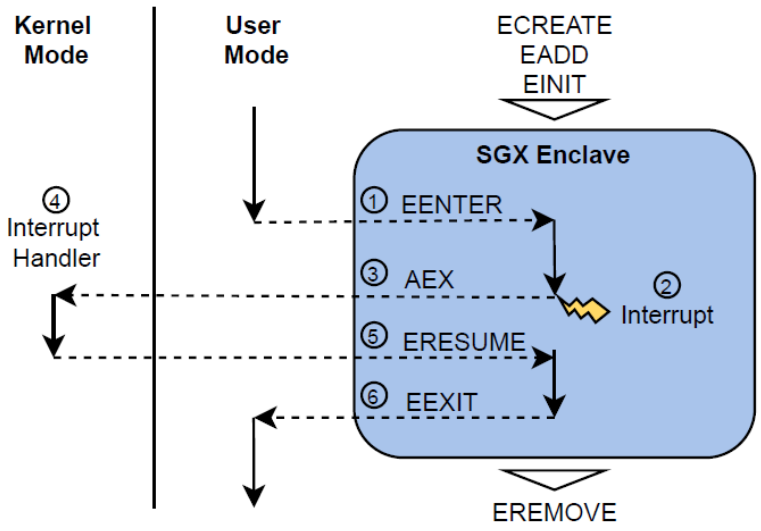


Figure 2.2: Enclave Life Cycle in Intel SGX

After the creation of the enclave, the application issues the EENTER instruction to transfer control to a pre-determined location in the enclave(1). During code execution in the enclave, if an exception or interrupt happens(2), it results in Asynchronous Enclave Exits (AEX) (3), and the Asynchronous Exit Pointer (AEP) points to the first instruction, which is executed after handling the interrupt(4). The handler can decide whether to continue the execution of the enclave or not. To resume its execution, it issues ERESUME(5). State Save Area (SSA) keeps the state of the enclave if an interrupt happens; additionally, it is important to note that Translation Lookaside Buffer (TLB) is flushed before leaving the enclave. In the next step, the application can issue a pre-defined function in the enclave and pass control to it which is known as ECALL. In contrast, OCALL allows executing a function outside the enclave in the application address space. In ECALL, data and addresses can be shared, but OCALL needs to copy data and purge TLB. At the end of enclave execution, EEXIT(6) puts the process into a normal state and flushes the TLB entries and clear registers to prevent data leaks.

When an enclave is created and initialized in Intel SGX, a cryptographic log is generated by the system. This log includes details such as the content (i.e., code, data, stack, and heap), the location of each page within the enclave, and the security flags being used. This is "Enclave Identity" which is a 256-bit hash digest of this log, stored as MRENCLAVE, representing the enclave's software Trusted Computing Base (TCB). To verify this value, one must first get the enclave's software TCB, followed by securely acquiring the expected enclave's software TCB and comparing the two values. In case two enclaves have the same hash, they are considered identical. Additionally, each enclave is signed by an author, leading to the presence of another measurement called MRSIGNER. This value is the hash of the author's public key.

2.2.3 Sealing

When an enclave is running, SGX protects the enclave's code and data from outside access, but after issuing EREMOVE, the enclave stops working and leaves the memory. It means all data associated with the running enclave will be lost. Sealing is a mechanism in which we protect data at rest in SGX applications. Strictly speaking, an enclave issues EGETKEY instruction to get the key assigned to it and uses that key to encrypt data on the disk. EGETKEY provides different keys based on the enclave request, and it is the responsibility of the enclave developer to choose an appropriate algorithm to encrypt/decrypt data on the disk. According to the given definition, SGX provides two types of sealing keys: Sealing to the current enclave and sealing to the enclave author.

Sealing to the Current Enclave – In this type of sealing, the key is bound to the identity of an enclave. In better words, the key derivation is based on MRNCLAVE, so if the code and data associated with the enclave change, a new key will be derived. Consequently, two distinct enclaves have distinct keys.

Sealing to the Enclave Author – In this type, the sealing key is bound to the enclave's author or signer. Therefore, two distinct enclave which are signed by the same person can get the same sealing key if they use this type of sealing. The same thing is true for a different version of an enclave. This feature enables enclaves to share data on disk while protecting them from outsiders.

2.2.4 Attestation

An enclave enables a process to protect and isolate its data from other active entities in a system. This model works as long as the enclave does not need any data from the other enclaves, but for any reason, if it needs to exchange data with other enclaves, they need a mechanism to prove that they can trust each other. Intel SGX calls the process in which enclaves prove their identity to each other attestation. Considering this definition, there are two types of attestation: Local Attestation and Remote Attestation.

Local Attestation – Before communicating on the same platform, two enclaves need to authenticate each other locally. This process is called local attestation. Figure 2.3 depicts the local attestation process [57].

1. Assume two enclaves A and B running on the same platform. There is a communication channel between them which does not need to be trusted. We assume B asks A to prove if it is running on the platform as B.

2. B retrieves its MRENCLAVE value and sends it to A.

3. A uses EREPORT instruction to produce a report for B using provided MRENCLAVE and sends the REPORT to B.

4. After receiving REPORT from A, B calls EGETKEY instruction to get the REPORT key to verify the REPORT. If the REPORT can be verified with REPORT key, B knows that A is on the same platform. It is possible since the REPORT key is specific to a platform.

5. B uses MRENCLAVE embedded in A's REPORT to create a REPORT for A and send it to A.

6. A can follow the same steps in phase 4 to verify that B is running on the same platform.

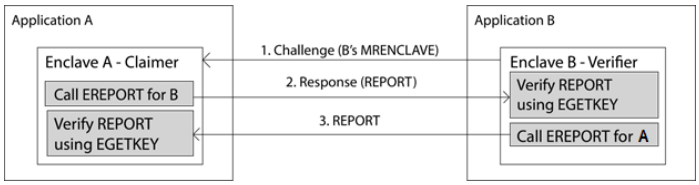


Figure 2.3: Local Attestation in Intel SGX

Remote Attestation - Remote Attestation – In the remote attestation, the attester and verifier are not on the same platform, so we need other components in the process of attestation, Quoting Enclave (QE) and Attestation Service. QE is an architectural enclave that verifies and transforms REPORT (locally verifiable) into QUOTE (remotely verifiable) utilizing Provisioning Key. On the other hand, Attestation Service validates the provided QUOTE and its data. Figure 2.4 illustrates an abstraction associated with remote attestation [57].

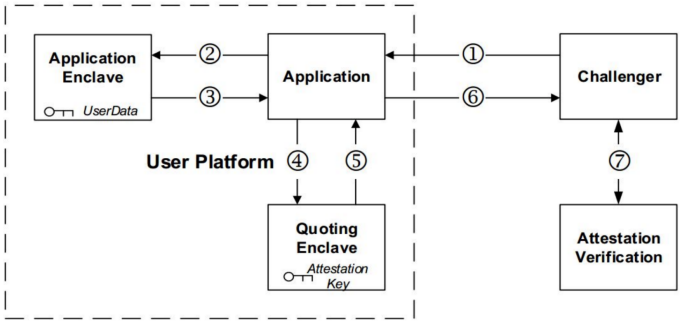


Figure 2.4: Remote Attestation in Intel SGX

1. First, the enclave needs a secret from a server, and it tells the application to request it on its behalf. The application sends a request, and the server replies with a challenge and asks the enclave to prove it has not been tampered.

2. The application gives the server's challenge and QE's identity to the enclave.

3. The enclave provides a manifest containing the challenge answer and an ephemeral public key that will be used to secure the communication between the enclave and the server later. Additionally, it calculates the manifest hash and includes it in the data section of EREPORT instruction. EREPORT instruction generates REPORT for QE which binds the manifest to the enclave, and at the end,

the enclave passes the REPORT to the application.

4. The application passes the provided REPORT to QE for verification and singing.

5. QE retrieves the report key and verifies the provided REPORT, so it generates QUOTE structure and signs it using the provisioning key. Then it passes it to the application.

6. The application sends QUOTE to the server for verification.

7. The server uses the Intel Attestation Service (IAS) to validate the signature of QUOTE; then, it validates the manifest integrity provided in QUOTE by comparing it with an expected value. If they are matched, the server accepts the answer for the challenge.

2.3 Threat Model

As mentioned in Chapter 1, threshold ECDSA signature finds common use cases in the realm of crypto exchanges. With this in mind, we shall provide a general threat model pertaining to threshold ECDSA in the context of crypto wallets.

Crypto exchanges typically maintain custody of their clients' crypto assets through a combination of hot and cold wallets. Hot wallets contain a small portion of the total funds, and require approval from both the client and the exchange to initiate a transaction. Due to the inherent security risks associated with hot wallets, the majority of the funds are stored in cold wallets that are physically isolated from the operational network of the exchange. Transactions from cold wallets are generally executed manually and require the approval of multiple stakeholders. In both scenarios, the utilization of the TSS, particularly threshold ECDSA signature, serves as a valuable tool. By distributing trust among multiple participants, the risk of significant fund loss can be mitigated.

The process of signing transactions through the utilization of TSS is multi-phased. In most implementations, TSS comprises three key phases: key generation, message signing, and key refreshing. The initial two phases are crucial for signing a transaction, while the key refreshing phase is an optional mechanism to proactively improve

security. During the key refreshing phase, participants update their key shares regularly based on the original DKG phase. This means that if an attacker gains access to a system once, they cannot utilize the same key after a certain interval of time (t). TSS employs complex cryptographic primitives and subprotocols such as VSS, homomorphic encryptions, commitments, and zero-knowledge proofs. Furthermore, TSS protocols run multiple rounds of communication in which the receiving party must validate the transmitted message. This adds more complexity to TSS protocols. To simplify the issue, we assume that during the DKG phase, the generated keys are backed up somewhere safe by each party to ensure that the attacker cannot destroy the TSS process by deleting the key shares of the involved parties. We also assume that adversaries can access the network and intercept or modify TSS messages as a result. Based on these notes, TSS must safeguard custodians against user abuse, internal malfeasance, and external malicious behaviors.

Despite the numerous security advantages provided by TSS, there have been instances of vulnerabilities and exploitations in the deployment of TSS protocols and their related subprotocols. These vulnerabilities have been documented in several studies [59, 9, 16, 56], and their exploitation can lead to the loss of hundreds of millions of crypto funds. These attacks indicate that in addition to the complexity of TSS protocols, their implementation can suffer from bugs. Although most TSS protocols were designed to operate in a dishonest majority environment, all the attacks introduced have resulted from gaming protocols in which the adversary deviates from the standard behavior of an honest party. Given this fact, one potential solution is to eliminate dishonest parties from the TSS environment. This approach would enable us to not only benefit from the separation of responsibilities among multiple parties but also mitigate the risks associated with threats to TSS protocols.

In order to protect TSS parties from dishonest players, it is essential to have robust security mechanisms in place on the machines that host the participants. Host-based anti-malware and Endpoint Detection and Response (EDR) systems are not sufficient to fully protect hosts from injected codes and privilege escalation attacks that can subvert security systems. Therefore, to ensure an honest environment in TSS protocols, we must also consider the possibility of compromising a host through malware and gaining control over its entire software stack.

To address this challenge, we rely on trusted hardware that provides a TEE. Specifically, each TSS party is shielded within an SGX enclave, which is Intel's implementation of a TEE, using a LibOS. Parties can distinguish between honest and dishonest parties based on their identities, which are given and verified by Intel's SGX. By using this approach, we can ensure that TSS parties are protected against attacks that target the host machine's software stack, and we can achieve a more secure and trustworthy TSS environment.

The research presented in this study relies on the original threat models of TSS and SGX, which means that the traditional software stack is not trusted, including the hypervisor, OS, system libraries, software packages, and applications. Instead, the hardware, LibOS, and enclave are trusted. We also consider the user as untrusted. The study does not consider DoS attacks, side-channel attacks, and physical attacks against the CPU [60] as they are addressed by other studies [61, 62, 63, 64]. It is assumed that the LibOS protects all disk Input/Output (I/O) and system Application Programing Interface (API) accesses [23]. By relying on smaller trusted components, the study aims to mitigate potential vulnerabilities and attacks against TSS protocols and ensure a secure environment for parties involved in threshold ECDSA.

Chapter 3

Methodology of the Research

In this chapter, we present the methodology employed to carry out our research. Initially, we define the security requirements of our system, taking into account the threat model identified in Chapter 2 regarding the threshold ECDSA protocol and multi-party crypto wallets. Based on these requirements, we propose an architecture that is able to satisfy them. Subsequently, we provide an overview of two different design implementations and describe their respective deployment strategies. Finally, we evaluate the efficacy of these implementations with respect to the security requirements and assess their resilience against potential attacks. Overall, this chapter outlines the structured and systematic approach undertaken in our research, which serves as a foundation for the implementation and evaluation of our prototypes.

3.1 Security Requirements

In the previous chapter, defining the threat model, we stated that we do not trust the traditional software stack, encompassing the hypervisor, OS, system libraries, software packages, applications, and users. Instead, the hardware, LibOS, enclave, and user are deemed trustworthy. The high-level goals of this research, based on this definition, are as follows:

1. Prevent a dishonest party from participating in the threshold ECDSA protocol
2. Prevent the semi-honest party from analyzing the threshold ECDSA protocol
3. Prevent an attacker from manipulating data and traffic of honest threshold ECDSA parties

As we pointed out in the previous chapter, a dishonest party can deviate from the protocol and game it according to TSS jargon; in better words, a malicious party initiates most of the attacks against threshold ECDSA protocols. On the other hand,

the semi-honest party follows the protocol, but he tries to collect and analyze the protocol messages sneakily and decipher valuable data. Based on these notions, we can define the Security Requirements (SR) of our system as follows:

1. **Protect code and data of a threshold ECDSA party (SR1)** The system must protect the confidentiality and integrity of code and data associated with running honest parties.

2. **Guarantee the confidentiality and integrity of threshold ECDSA messages on the network (SR2)** The system must guarantee the confidentiality and integrity of messages that are sent to and received from honest parties.

3. **Prevent exploitation of threshold ECDSA protocol by dishonest parties (SR3)** The system must authenticate honest parties and distinguish them from dishonest parties. It must allow just honest parties to participate in the threshold ECDSA protocol; therefore, dishonest participants cannot game the protocol and exploit it.

4. **Protect threshold ECDSA protocol from collecting data about the protocol (SR4)** The system must protect threshold ECDSA data and messages from collecting and analyzing. In other words, a semi-honest player cannot participate in the protocol and explore it to find any exploitation.

5. **Protect threshold ECDSA protocol from traffic redirection (SR5)** The system must prevent MITM or Domain Name Service (DNS) redirection techniques in which messages between honest parties are redirected to a middleman that collects valuable data.

The proposed design in this study provides SR2-5 directly compared to running TSS client only on the LibOS inside an enclave.

3.2 Design

3.2.1 Overview

As we mentioned in the previous section, our research aims to establish an honest environment for the threshold ECDSA protocol to protect multi-party crypto wallets

from potential attackers. In this section, we will provide a design overview of the proposed system and how Intel's SGX technology can help us to achieve our goals. At the high-level view, our system has two potential components 1) the threshold ECDSA party running inside an enclave and 2) the trusted (transportation) tunnel protecting the party's network traffic and shielding it from attackers. In this design, each TSS party runs on top of LibOS, and LibOS execute the opcodes inside an enclave, protected from the host software stack (SR1). The Threshold ECDSA protocol is a sophisticated piece of code. Like other security-related software, it needs multiple development cycles to be mature enough; however, it can still be susceptible to vulnerabilities [61, 62, 63, 64]. We explained in Chapter 2 that running the code inside an enclave needs calling special CPU instructions which are associated with an enclave like ECALL/OCALL. These instructions are only available through TEE-related libraries and SDKs like sgxsdk [65] in the basic form. In practice, TSS clients and most applications were not developed for an enclave, and modifying their source code will be expensive. Considering this fact, we need a solution to run the unmodified threshold ECDSA client inside an Enclave. LibOSes, in our case Gramine, help us run the unmodified mature TSS client inside an enclave and protect its code and data from malware living inside its host (SR1). Technically speaking, LibOS handles received system calls from TSS party on behalf of OS transparently and shields the client from OS attacks [66].

While LibOS encrypts disk I/O operations and guarantees the integrity of data associated with the TSS client inside an enclave, network operations rely on the TCP/IP stack of an untrusted OS. To sign a message collaboratively, TSS parties need to communicate with each other. In our system, this communication is done through a trusted tunnel established on unique features provided by the SGX platform. Strictly speaking, we establish an end-to-end trusted tunnel between players to provide confidentiality and integrity of the threshold ECDSA messages on the network (SR2). Since OS and hypervisor are considered untrusted in our threat model, the trusted tunnel component must be located inside the enclave to remove any trust in the TCP/IP stack of the OS.

Additionally, to preserve the integrity and confidentially of TSS messages over secure channels, we need a secure transportation layer like TLS (Transport Layer

Security) or DTLS. For participating in the protocol and sending a message, each party needs a valid X.509 certificate, and each TSS party can authenticate other parties based on their valid certificates. Mutual authentication between participants of threshold ECDSA protocol removes the possibility of joining any parties without a valid certificate. It is crystal clear if a malicious party has a valid certificate, he can participate in the protocol and exploit potential vulnerabilities. In our design, to prevent dishonest parties from participating in the protocol (SR3), we bind the TLS certificate to the identity of each player's code and data running inside an enclave; therefore, we can verify the identity of the party claiming honesty. In technical words, we establish an attested secure channel between TSS parties which prevents joining any dishonest parties trying to deviate and game the protocol. In this setting, even a semi-honest player cannot excavate any data from the protocol because either it must process the collected data inside the enclave that changes its SGX measurements or analyze the data outside of the enclave, which is also not attainable since data outside of enclave are encrypted (SR4). The attested and mutually authenticated channels between parties eradicate the success of message redirection or MITM attack (SR5).

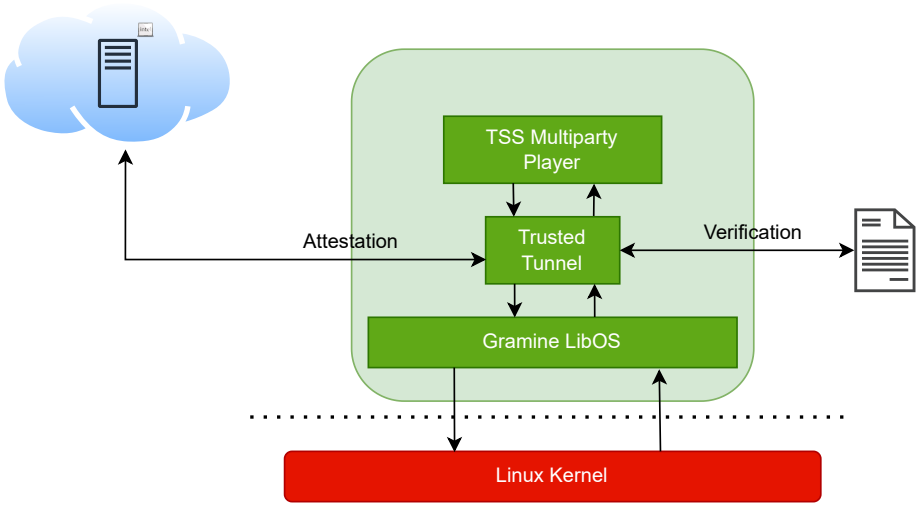


Figure 3.1: Overall Design of An Honest Party with an Attested Tunnel

Figrue 3.1 shows the overall design of multi-party crypto wallets in an honest environment using SGX enclave, proposed in this study. As depicted in this figure, the trusted tunnel and LibOS shield TSS party component from the malicious actors locally and remotely. The trusted tunnel listens for incoming connections to detect

connecting parties' identities using SGX remote attestation. According to this notion, whenever the party inside the enclave starts running, it creates a new pair of keys and binds the certificate to the enclave report [28]. Before accepting the connection, the trusted tunnel component checks whether the party's identity is on the Allowlist or not. If the remote party is authenticated, the connection is established; otherwise, the connection is rejected. When secure channels are established, honest players can continue the execution of the distributed signing protocol.

3.2.2 Authenticating An Honest Party

In chapter 2, we described the concept of the TSS in which for (t, n) threshold signing, n parties start the DKG protocol like Joint-Pederson's DKG, and each party receives its share. After having key shares, $t + 1$ out of n parties are needed to create ECDSA signature for a message. In our design, we do not change the original protocol; however, we add another layer of protection on top of the existing protocol to detect possible malicious parties and prevent them from joining the protocol.

In this hypothetical situation for 3 TSS parties, suppose that in the DKG phase, adversary party 4 masks honest party 3 from participating in the DKG protocol. Without a trusted tunnel, the adversary can impersonate a normal player and participate in the key generation phase. With the trusted tunnel component in our design, honest players can identify other honest parties and prevent dishonest party 4 from joining DKG phase and exploiting possible vulnerabilities. In the signing phase, while TSS clients run inside an enclave, and all its code and data are protected from an untrusted software stack, suppose that a dishonest party could access the key share of party 3. Even after having the key share, the adversary fails in the authentication, and he does not have the opportunity to exploit TSS protocol because of the proposed authentication mechanism. The key element in our design which enables the authentication of honest parties is the "Allowlist" generated based on SGX measurements.

As we pointed out in chapter 2, Intel's SGX technology can measure the code, data, and attributes associated with a running process which is called MRENCLAVE in SGX terminology. To authenticate honest parties, we force them to add SGX measurements and parameters in their self-signed certificates. Each honest player has a list of other honest players based on SGX parameters. Figure 3.2 displays

a sample Allowlist for an honest player. When a party tries to participate in TSS protocol, it needs a X.509 certificate containing SGX parameters. After receiving a connection request from a candidate party, an honest party validates the given certificate based on provided SGX parameters in it. First, it checks the authenticity of SGX parameters using a remote attestation process, and then it checks whether the provided SGX parameters are in its Allowlist. The honest party rejects the connection if either of these two steps fails. The given design helps us to authenticate legitimate parties during TSS protocol.


```
{
  "TEEs": [
    {
      "MRENCLAVE":"ce9a95624966ec4b85635531fa5f9540109bafc647992746a02e01f7e9f90546",
      "MRSIGNER":"6d22037420f7c410cd45b97459fa04734af15ffb574ac2b2f8710f83ef3eb269",
      "ISV_PROD_ID":"1",
      "ISV_SVN":"1",
      "app":"tss-ecdsa-gg20_genkey"
    },
    {
      "MRENCLAVE":"ce9a95624966ec4b85635531fa5f9540109bafc647992746a02e01f7e9f90546",
      "MRSIGNER":"a193f2d73c9e9085a351b787403d3de329f17df14d60a7d463dc92b4ef325348",
      "ISV_PROD_ID":"1",
      "ISV_SVN":"1",
      "app":"tss-ecdsa-gg20_signing"
    }
  ]
}
```


Figure 3.2: Trusted Tunnel Allowlist

This mechanism allows us to authenticate honest parties not only based on MRENCLAVE measurement but also based on their creators, MRSIGNER, and their software specifications, Product ID (ISV_PROD_ID) and Software Version Number (SVN). It adds another level of flexibility in the authentication process in which we can allow or reject connections based on the creator of a party, the software of a party, and the version of software running on a party.

3.2.3 Deploying the Honest Environment

In this section, we elucidate the deployment of an honest environment from multiple perspectives: **creating enclaves, managing certificates, and updating the system**.

Creating a shielded TSS party inside an enclave - In an honest environment, the party plus its dependencies are shipped as a container image. Technically

speaking, Gramine LibOS has a manifest file specifying the enclave configuration and its list of sealed files. After configuring the LibOS based on the requirement of the honest party, we create a container image containing libraries, LibOS, the TSS party component, and the trusted tunnel. Executing TSS party inside an SGX enclave is the entry point of container, so we just need to start the container. Regarding SGX measurements to implement mutual authentication, the system can extract them automatically from the container image [67] and add them to the Allowlist. Since the Allowlist contains the honest parties' measurements, we cannot put it in Gramine's manifest file to be sealed because it causes a circular dependency. This design decision is not against any security requirements we discussed earlier. Suppose the adversary compromises the software stack of an honest player. In that case, it can only modify its Allowlist and evade our proposed authentication mechanism. However, it still needs to take control of other t parties to participate in the signing phase of a message. In better words, in the worst-case scenario in which the $t+1$ parties are compromised, the dishonest party can join in the TSS protocol; it means the problem is downgraded to the original TSS protocol. Without the trusted tunnel component, if the adversary controls $t+1$ parties, it can get their keys and sign any messages. In contrast, our system guarantees the integrity and confidentiality of key shares inside the enclave and on the disk.

Managing certificates - The provided design automatically handles key generation and certificate management without user intervention. When starting the container, the trusted tunnel component uses LibOS APIs to generate a self-signed certificate with enclave measurements. The trusted tunnel uses the generated certificate to establish a secure channel to other honest parties. On the other side of the secure channel, after receiving the connection request, the honest party uses the provided certificate to do remote attestation, match it with the Allowlist, and complete the authentication process.

Update the system - Patch management and security updates are among the big concern in any system. If we update any system component (LibOS, libs, TSS component, trusted tunnel), we must build the container image again and extract its SGX measurements. In fact, we update a bundle of components. Due to ISV_PROD_ID and ISV_SVN in the Allowlist, our system can detect the crypto wallet applications

and their version and issue a security warning to a central logging and monitoring system. Gramine LibOS has the concept of protected files, allowing dynamic file contents. Since Gramine does not include them in MRENCLAVE measurement, this feature enables us to update protected files without changing the enclave's measurement; however, we should consider that it can cause some security concerns if it is not configured properly [68].

3.3 Implementation

In this section, we will explain our implementation of the proposed design, taking into account the two key components of the TSS engine and trusted tunnel, which operate within an enclave for each honest party. As illustrated in Figure 3.1, all communication of the TSS component with other parties occurs through the trusted tunnel. We leverage Gramine LibOS to execute unmodified code within the enclave, so all socket operations rely on the OS TCP/IP stack, which is untrusted in our threat model. To solve it, we must place the trusted tunnel within the enclave before transmitting traffic to the OS kernel. Based on this approach, we have two options to implement the design:

1. Creating a transparent trusted tunnel that enables all enclave traffic to pass through it, or

2. Integrating the trusted tunnel into the TSS component as the transportation layer.

In the following subsections, we will describe the architecture pertaining to each of these design choices and their implementation in our prototypes.

3.3.1 Transparent Attested Tunnel

As previously mentioned, the purpose of the transparent attested tunnel is to secure enclave network traffic and authenticate trustworthy TSS parties. However, redirecting enclave network traffic to the attested tunnel requires careful consideration and implementation. Specifically, two options are available: redirecting traffic to the attested tunnel via networking techniques on a designated port or redirecting all

socket operations of the enclave to the transparent attested tunnel without leaving the enclave.

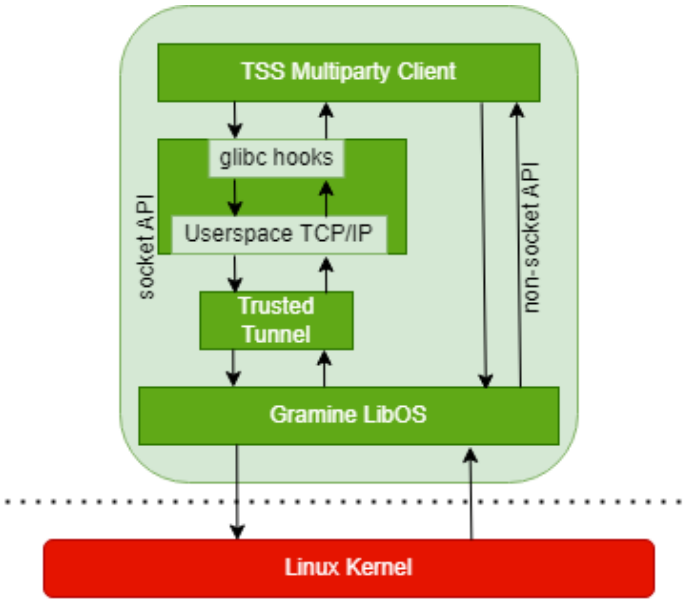


Figure 3.3: Architecture of Transparent Attested Tunnel with Userspace Networking

Upon closer inspection, the first option is not aligned with our security requirements because Gramine relies on the OS for networking, and any socket operations prior to the tunnel would violate security requirements by crossing enclave boundaries to execute networking operations on the OS kernel's TCP/IP stack. As for the second option, we opt to insert hooks into glibc to redirect socket operations to the tunnel and permit data segments to exit the enclave after establishing a secure channel. The most effective approach for achieving this goal is through the use of a userspace network stack[69], which provides networking operations directly inside the enclave and minimizes modifications to system libraries. As illustrated in Figure 3.3, all socket operations are transmitted to the userspace TCP/IP stack and subsequently routed to the trusted tunnel through the userspace networking stack. The trusted tunnel ensures the confidentiality and integrity of data leaving the enclave. Despite finding a few successful examples of porting lwIP [69] to the enclave, we could not implement the desired workflow by porting userspace TCP/IP stacks [70, 71, 72] into the enclave. One significant limitation is the restriction in the LibOS related to syscalls, which prevents the running code on LibOS from accessing all syscalls provided by the OS. For

instance, Gramine encountered issues with executing codes that called unsupported syscall netlink. Consequently, we overlooked some security requirements in the first prototype and deployed networking techniques. In this implementation, to provide a transparent trusted tunnel, we use TUN/TAP [73] interface to redirect traffic between the TSS component and the trusted tunnel and vice versa. Because the TUN/TAP functionality is a host kernel feature, the integrity and confidentiality of data may be at risk due to the untrusted nature of the host kernel (SR2). Figure 3.4 shows different components of the implementation. To rectify the compromised security requirements, we will provide resolutions in the second implementation.

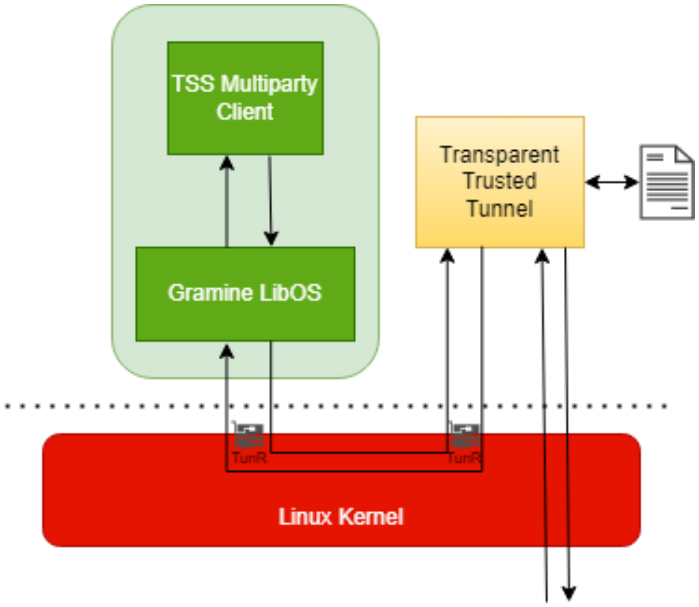


Figure 3.4: Architecture of Transparent Attested Tunnel with TUN/TAP device

Initialization – To start an honest TSS party within an enclave, it is necessary to configure Intel's SGX [17, 65] and Gramine LibOS [22] on the system. Gramine LibOS leverages SGX technology to shield Linux system calls from applications within an enclave. Assuming that the Docker image of the TSS honest party has been delivered to the target host and TSS honest parties have been deployed, we use an initialization script in the container's entry point to execute multiple steps sequentially. Initially, the script initiates the TSS component within an enclave and generates the honest party's key and certificate, which are subsequently written to disk. Next, it starts the trusted tunnel, which establishes a TUN device and waits for incoming packets

while simultaneously listening on a socket for incoming connections from other parties. Finally, the initialization script runs the TSS component to participate in the threshold ECDSA signing protocol.

Credential Generation – For a successful authentication, an honest party requires a private key and a certificate that is signed by the SGX platform. To achieve this goal, we leverage the Gramine ra-tls interface, which abstracts away the complexities associated with generating reports and quotes within an enclave, as discussed in Chapter 2. ra-tls is an implementation of Knauth et al.'s study [28] that incorporates Intel's SGX remote attestation process into the TLS connection setup. Specifically, it utilizes the X.509 certificate extension to store SGX-related information, such as the SGX Quote and Intel's SGX Certificate, which enables remote users, including other honest TSS parties in this case, to verify the authenticity of an honest party running inside an enclave. The ra-tls implementation provides ra_tls_attest.so shared library to generate an enclave's unique key and certificate [74].

Connection Setup – With the self-signed certificates and private keys generated in the previous step, the transparent attested tunnel in our implementation can establish a secure connection to other honest parties. In our implementation, we opted to use DTLS [75] to provide a secure channel. The primary reason behind this decision is that DTLS uses the User Datagram Protocol (UDP) transport protocol to transmit messages between parties, resulting in lower overhead compared to standard TLS. The role of the trusted tunnel is to provide integrity, confidentiality, and mutual authentication; therefore, the order of packets has no bearing on the transparent attested tunnel. Instead, it receives packets from the TUN devices, and it is TSS component's responsibility to use Transmission Control Protocol (TCP) connections or UDP streams and manage the order of packets. Strictly speaking, the role of transparent attested tunnel is to wrap the TSS messages in another packet and transmit it through the network.

Connection Establishment – After setting up the trusted tunnel, it waits for incoming connections. As we mentioned earlier in this section, we use the TUN/TAP virtual device feature in the kernel to establish this connection. Technically speaking, the transparent attested tunnel creates a TUN device and runs a thread of execution

to continually read from it and write to an established socket connection. Additionally, the transparent attested tunnel runs another thread of execution to listen on a specific port, welcome the new authenticated TLS connection, and write the buffer into the TUN virtual device. This implementation enables the TSS engine to pass messages to other parties and exchange messages. In essence, we create an environment similar to a local network when we establish a Virtual Private Network (VPN) connection to a VPN server [76]. Every honest party has a local DNS record of the other honest TSS parties in its sealed filesystem. This address is the virtual address used by the TUN virtual device on each container running the TSS party. Thus, when an honest TSS party wants to send a message to another party, it first resolves the other party's virtual IP address from the local DNS and then sends it, which is directed to the TUN virtual interface. The transparent attested tunnel reads the packet from the TUN device, inspects its IP header, and finds the other party's public address based on the provided mapping to each trusted tunnel. The trusted tunnel wraps the packet inside another packet and forwards it to the trusted tunnel running on the other side of the communication. The trusted tunnel on the other side of the channel waits for new incoming connections and authenticates them. After passing authentication, it unwraps the received packet, retrieves the original packet and TSS message, and forwards it to the TUN device, which is eventually received by the honest party waiting to participate in the protocol.

Authentication – The distinguishing of honest parties from other participants in the TSS protocol is a crucial aspect of this study, and it is implemented by the transparent attested tunnel. Honest parties establish connections with each other via mutually authenticated DTLS channels using the provided key and certification for the trusted tunnel. The trusted tunnels verify the correctness of the signature and attestation reports embedded inside the certificate using the remote attestation mechanism. In this implementation, we use the Enhanced Privacy Identification (EPID) remote attestation mechanism [17] and utilize Gramine's library, ra_tls_verify_epid.so. When IAS verifies that the certificate has an authentic quote and measurement, the trusted tunnel matches the given report with the Allowlist provided to it. If the matching is successful, the connection is established. The connection is rejected if the verification fails in any of the previously mentioned steps. After accepting the

connection, the transparent attested tunnel on the second party sends back its credentials to the first party. The first party validates the other side's credentials by following the same process we explained. At the end of this process, both ends of the connection are mutually authenticated, ensuring the confidentiality and integrity of the messages.

An essential concept related to the correctness of mutual authentication is SGX measurements. Enclaves measure Gramine LibOS libraries and memory-mapped manifest files (Equation 3.1). The measurement of the memory-mapped manifest file depends on all dependencies of the TSS engine, including the executable, required libraries, and local DNS files, etc. (Equation 3.2). Changing the contents of any components in Equation 2 leads to a different measurement and report of a TSS party, which is embedded in its X.509 certificate. This mechanism enables the trusted tunnel to detect the unique identity of honest TSS parties and prevent malicious actors from joining the protocol because they are executing different codes, using other data, and showing different behaviors to exploit possible vulnerabilities in the protocol.

$$MRENCLAVE \leftarrow SGX_{measure}(LibOS, MF) \tag{3.1}$$

$$MF \leftarrow H\{H(TSS), H(Tunnel), H(Lib), \ldots\} \tag{3.2}$$

3.3.2 Integrated Attested Tunnel

The current prototype of transparent attested tunnel heavily relies on the TUN/TAP virtual devices and the OS kernel's TCP/IP stack; hence, its implementation violates some of the proposed security requirements we introduced earlier. To address this issue, we have implemented a second prototype that adheres to the principles outlined in the design overview in which the trusted tunnel is natively integrated into the TSS party application in this implementation. The TSS party application consists of three modules: 1)the TSS engine, 2)the transport module, and 3)the authentication module. To build the TSS engine component, we used the tss-lib library [15], which is a Multi-Party Threshold Signature scheme library implemented by Binance. We then added a session management and transportation layer on top of the TSS engine to

facilitate message exchange between TSS parties. The uniqueness of this implementation lies in the integration of authentication for honest TSS parties based on SGX measurements, as illustrated in Figure 3.5. In this implementation, the TSS party application runs inside an enclave that is totally aligned with our security goals.

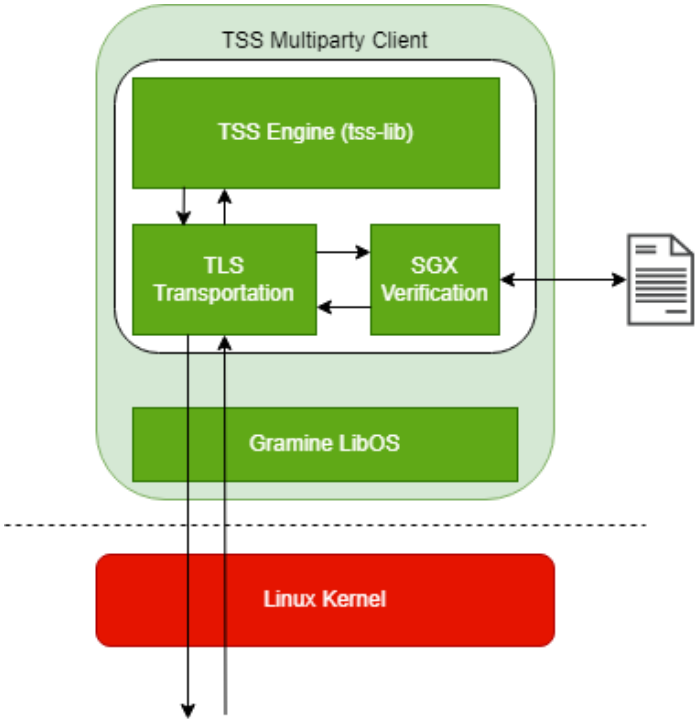


Figure 3.5: Architecture of Integrated Attested Tunnel

Integrated attested tunnel offers several advantages over the previous prototype. First, SGX and Gramine can provide measurements and quotes related to the TSS party application as a whole, which reduces the likelihood of misconfigurations and vulnerabilities resulting from initialization processes. Second, all the keys and certificates used to establish secure channels will be confidential and tamper-proof inside the enclave bundle. Additionally, all sensitive data associated with TSS party's messages will be transmitted through the trusted tunnel, ensuring their confidentiality and integrity. Simply put, no data will leave the enclave boundaries without being protected. However, the downside of this implementation is that we cannot deploy the unmodified version of the TSS party application. Integrating the trusted tunnel into an existing TSS application can be complicated and expensive. In this section,

we are going through the lifecycle of the TSS party application with the integrated attested tunnel.

Initialization – Similar to the transparent attested tunnel, the integrated attested tunnel depends on Intel's SGX and Gramine LibOS, which necessitates setting up these technologies before deploying the honest TSS parties. The container image includes the LibOS libraries, the TSS party application, and their dependencies. The container's entry point launches the honest party's application inside an enclave that integrates the trusted tunnel mechanism. Following the TSS party's execution inside the enclave, the stage is set to generate the honest party's credentials associated with the enclave.

Credential Generation - As all the components of the TSS party application are executing within an enclave, the process of generating credentials is straightforward in the case of the integrated attested tunnel. Before starting the TSS engine and trusted tunnel, the TSS application employs the ra-tls libraries offered by Gramine LibOS to generate the private key and TLS certificate linked with the SGX quotes. Since all the TSS application components operate inside the same address space, there is no need to store these credentials outside of the enclave address space to set up the secure channel; instead, we pass them to the integrated attested tunnel.

Connection Setup – Unlike the transparent attested tunnel, which is not responsible for managing the order of application messages, the transportation layer in this implementation must ensure the sequence of data segments and TSS messages. The transportation layer utilizes TCP connections to ensure the order of data segments and provide a reliable connection. Moreover, to manage the order of TSS messages generated for each round of TSS protocol, we assign a Session ID to each TSS message to preserve their order in the protocol; moreover, we implement a session management layer which queues messages and only passes messages with right Session ID to the TSS engine. We employ the TLS library to implement a secure channel that fulfills the aforementioned requirements. The implemented transportation layer utilizes the provided credentials to configure a TLS socket listening for incoming connections.

Connection Establishment – Once the trusted tunnel is established, the TSS engine can initiate communication with other TSS parties. To provide optimal performance and high concurrency, the transport layer deploys multiple threads of execution

for sending and receiving TSS messages over the wire. The integrated attested tunnel uses local DNS records, which are embedded into the container image, to resolve the hostname of honest TSS parties into their corresponding IP addresses, and then connects to an honest party. Next, it utilizes the generated credentials to authenticate with the other honest party. After successful authentication, the honest party begins exchanging TSS messages over the established trusted tunnel. Establishing connections and exchanging messages in this implementation is more straightforward and intuitive than the previous implementation.

Authentication – The authentication phase in the integrated attested tunnel is pretty similar to what we explained in the previous part; however, all of the checking is done inside an enclave protected from system-level attackers. The value of *MRENCLAVE* in the quote is the most important measurement which is used to identify a TSS party, but depending on the situation, we can modify the authentication parameters and authenticate honest parties based on *MRSIGNER* value, who create and signed the enclave. Additionally, binding *ISV_PROD_ID* and *ISV_SVN* to the measurements allows us to identify the type and version of TSS application. This approach strengthens the security of the authentication process, makes it less vulnerable to attacks, and enhances the system's overall security.

3.4 Security Analysis

This section presents a security analysis of the proposed design in light of the security requirements introduced at the outset of this chapter. The analysis begins by identifying potential threat actors within the system, followed by an overview of various attacks that can be carried out against the system. Each attack is examined in terms of who the attacker is, the attacker's methodology, and the ability of proposed designs to defend against the attack while fulfilling the given security requirements.

A multi-party crypto wallet that employs a TSS to sign transactions faces several types of attackers. A malicious user (MalUser) attempts to abuse his privileged access to sign unverified transactions and transfer funds to his own account. A system-level attacker (Sys) aims to gain control over the entire software stack of the system on which the crypto wallet operates to achieve his objectives. A dishonest TSS party attacker (Dishonest) seeks to deceive other parties and exploit the protocol to gain

Attack	Attacker	Defense Method	Protection	
			TaT	IaT
A01-Abusing the Privilege	MalUser	TSS	✓	✓
A02-Tampering with the Code or Data	Sys	SGX Enclave	✓	✓
A03-Tampering with TSS party's Dependencies	Sys	LibOS	✓	✓
A04-Participating as a Fake Enclave	Dishonest, Semi-Honest	Trusted Tunnel	✓	✓
A05-Impersonating an Honest TSS Party	Dishonest, Semi-Honest, MITM	Trusted Tunnel	✓	✓
A06-Revealing the Key Associated with Trusted Tunnel	Sys, MITM	Trusted Tunnel	✗	✓
A07-Tampering with Trusted Tunnel	Sys, MITM	Trusted Tunnel	✗	✓
A08-Spoofing DNS	Sys, MITM	LibOS, Trusted Tunnel	✓	✓
A09-Tampering with Allowlist	Sys	LibOS, TSS, Trusted Tunnel	✓	✓
A10-Exploiting TSS Protocol	Dishonest, Semi-Honest	Trusted Tunnel	✓	✓
A11-Exploiting Buggy Implementation	Dishonest, Semi-Honest	Trusted Tunnel	✓	✓
A12-Steering Input	Sys	TSS + Trusted I/O	✓	✓
A13-Compromising Crypto Wallet	Enclave	TSS, Trusted Tunnel	✓	✓

Table 3.1: Attacks on Multi-Party Crypto Wallets with TSS

access to sensitive data such as key shares or disrupt the system. A semi-honest TSS party attacker (Semi-Honest) aims to participate in the TSS protocol as a benign player but monitors all protocol messages to extract sensitive information. Finally, a Man In The Middle attacker (MITM) intercepts messages exchanged between TSS parties to decipher key shares or other sensitive data. In Table 3.1, these attackers are referred to as MalUser, Sys, Dishonest, Semi-Honest, and MITM, respectively, to specify the adversarial actor of each attack. Additionally, TaT and IaT stand for Transparent and Integrated Attested Tunnel respectively.

Abusing the Privilege (A01) – As previously noted, one of the primary concerns surrounding cryptocurrency funds is the irreversibility of transactions. A malicious insider at a cryptocurrency exchange can leverage his privileges to transfer vast amounts of funds to his personal account while tracking these transfers becomes almost impossible if he employs cryptocurrency mixers to obfuscate his traces [77]. A TSS distributes signing privileges among multiple participants to prevent a SPOF that could be exploited by a fraudulent insider. The security provided by TSS aligns with SR1.

Tampering with the Code or Data of TSS Party (A02) – A system-level attacker can potentially gain control over the entire software stack of a system hosting a cryptocurrency wallet, thereby enabling him to manipulate the execution of the wallet and sign unverified transactions. Both TSS and TEE technologies used in the proposed design mitigate against this attack. TSS prevents single-point-of-failure scenarios where a lone signer can complete a transaction, while the SGX enclave shields the running crypto wallet against code and data tampering that could lead to

invalid transactions or loss of funds. These security measures map to SR2 security requirement.

Tampering with TSS party's Dependencies (A03) – A system-level attacker can alter the expected behavior of a crypto wallet by manipulating the system's dependencies, including libraries, configuration files, LibOS binary, and LibOS manifest file. Since the trusted tunnel binding the enclave's identity and TSS party to a TLS certificate leverages SGX quotes, any changes to the TSS party's dependencies will be reflected in SGX measurements. Given this consideration, both of the suggested designs for the trusted tunnel are capable of detecting tampering with dependencies (SR2).

Participating as a Fake Enclave (A04) - A dishonest and semi-honest TSS player may run malicious codes inside an enclave and utilize ra-tls to generate a related certificate. As the code executes on an SGX platform, the self-signed certificate generated by the malicious party is verified by the remote attestation process. In this way, a dishonest or semi-honest player can participate in the distributed signing process and violate SR4 and SR5. However, both of the proposed designs can detect invalid runtime measurements by using an Allowlist containing the SGX measurements of authorized wallets.

Impersonating an Honest TSS Party (A05) – A potential attacker in the form of a dishonest or semi-honest TSS party, or a MITM may intercept the traffic of the trusted tunnel and extract embedded SGX measurements associated with an honest player. Even if the attacker fabricates a TLS certificate with valid SGX quotes, they cannot join the TSS protocol because the trusted tunnels authenticate only honest parties. Therefore, since the embedded report is not bound to the SGX self-signed certificate, the impersonated connection is rejected. Protecting the system against the aforementioned attack would satisfy security requirements SR3, SR4, and SR5.

Revealing the Key Associated with Trusted Tunnel (A06) – This attack is associated with the risks of SGX self-signed certificates and trusted tunnels in the context of two types of attackers: system-level attackers MITM attackers. A system-level attacker seeks to acquire access to the private keys used in generating the self-signed certificates. In contrast, a MITM attacker attempts to conduct rollback

attacks and manipulate the secure channel protocol employed by the trusted tunnel. The outcome of these attacks is the disclosure of the trusted tunnel key, so the attacker can decrypt captured packets in violation of SR3 or impersonate himself as a legitimate party in violation of SR4 and SR5. To counter the risk of rollback attacks and ensure the confidentiality and authenticity of the trusted tunnel, both proposed designs enforce a strong CipherSuite and incorporate Deffie-Hellman key exchange protocol to offer Perfect Forward Secrecy (PFS). Furthermore, by generating a new pair of keys for each enclave execution, the security of the system is reinforced, as the exposure of keys does not compromise all sessions. However, since transparent attested tunnel operates outside of the enclave, it is vulnerable to system-level attacks, and a dishonest or semi-honest party can abuse the revealed keys. Conversely, the integrated attested tunnel operates in the enclave alongside the TSS engine, which renders the key inaccessible to system-level attackers, eliminating any chance of key revelation.

Tampering with Trusted Tunnel (A07) – System-level and MITM attackers aim to manipulate data packets transferred through the trusted tunnel, thereby violating SR2. Such attackers attempt to disrupt communication, reveal sensitive data, or alter it for malicious purposes. In the case of the transparent attested tunnel, the implemented prototype employs the Datagram Transport Layer Security (DTLS) protocol to guarantee end-to-end confidentiality and integrity. However, it is the application's responsibility to maintain the order of message delivery. Since the trusted tunnel runs outside of an enclave, it leaves the communication vulnerable to tampering by system-level attackers. Conversely, the proposed design of the integrated attested tunnel leverages the Transport Layer Security (TLS) end-to-end secure channel and Transmission Control Protocol (TCP) reliable connection, which is protected within an enclave. This approach effectively mitigates the risk of trusted tunnel tampering and preserves the integrity and confidentiality of the trusted tunnel traffic in line with SR2.

Spoofing DNS (A08) – The attacker, system-level or MITM, may utilize DNS spoofing attack to extract sensitive data of a crypto account. This technique enables the attacker to redirect all traffic associated with the wallet to a server of their choice, facilitating the interception of all TSS messages and the extraction of key

shares (SR6). However, the proposed designs in our research address this issue and provide robust protection against such attacks. Specifically, our design incorporates a trusted tunnel that provides an end-to-end secure channel that cannot be intercepted. Furthermore, mutual authentication is implemented within the trusted tunnel, preventing any possibility of communication with malicious actors. Lastly, the system configuration files like local DNS records are protected by LibOS manifest, effectively eliminating any potential DNS spoofing and traffic redirection. We can add integrity-protected DNS protocols (DNSSEC, DoH, DoT) to the crypto wallet bundle as another layer of protection. These comprehensive measures effectively mitigate the risk of DNS spoofing and safeguard the security of the crypto wallet.

Tampering with Allowlist (A09) – In our design, Allowlist is located outside of crypto wallet bundle because putting Allowlist inside the bundle causes circular dependency between parties' SGX quotes; hence, Allowlist is not sealed and protected by LibOS. The system-level attacker can exploit this design decision and tampers Allowlist contents with its fake enclave measurments. The attacker can then proceed to communicate as dishonest and semi-honest parties with honest parties whose Allowlists have been tainted. This attack cannot compromise the whole system because an honest party just accepts connections from malicous parties, but it does not send any messages to adversaries because as mentioned in A08, DNS spoofing, all IPs and hostnames configuration associated with honest parties are tamper-proof by LibOS sealing or other techniques.

In the case of a transparent attested tunnel, the attacker must compromise t+1 systems and their Allowlists to participate in the protocol, which is equivalent to a basic attack on the TSS protocol. However, in our prototypes, all sensitive data is stored inside an enclave, and the attacker must attempt to exploit the TSS protocol to be successful (SR4-5). Similarly, in the case of an integrated attested tunnel, even after tampering with t+1 Allowlists, honest parties do not send any messages to adversaries, which is equivalent to satisfying SR4-5 security requirements. Overall, while the placement of the Allowlist presents a potential vulnerability, our system designs and protocols are structured to mitigate these risks effectively.

Exploiting TSS Protocol (A10) – As previously stated, TSSs are intricate

protocols that contain sub-protocols that lack standardization. Due to this complexity and the lack of standard protocols, many reported attacks have targeted TSS protocols, especially threshold ECDSA signatures [11, 12, 78, 45]. In most of these attacks, the attacker takes control of one party and sends unverified and corrupted protocol parameters to the victim. Over multiple rounds of message exchanges, the attacker can extract the key shares (SR4). Depending on the strength of the protocol, a semi-honest party can extract the private key by merely observing exchanged messages (SR5). A trusted tunnel is employed in the proposed designs, which utilizes SGX quotes to authenticate honest parties that wish to participate in the TSS protocol. This approach prevents malicious actors from involving themselves in the TSS protocol and potentially exploiting it. Therefore, the embedded trusted tunnel and provided mutual authentication protect wallets against attacks associated with TSS protocol exploitation.par

Exploiting Buggy Implementation (A11) – In his paper, Aumasson[9] introduces "Forget and Forgive," "The Lather, Rinse, and Repat," and "Golden Shoe" attacks, which stem from a buggy implementation of TSS protocols. In these attacks similar to A10, the attacker takes control of one party and sends unverified and corrupted parameters to the victim to exploit a bug in an enclave (SR4-5). As we mentioned in the previous attack, the proposed designs leverage a trusted tunnel and mutual authentication based on SGX quotes to prevent the attacker from sending malformed messages and exploiting the protocol.

Steering Input (A12) – Upon acquiring control over the host's software stack, a system-level attacker can control the I/O operations handled by the OS. It enables the attacker to manipulate the information displayed to the user, thereby causing the user to initiate a transaction that appears legitimate but in reality, transfers funds to the attacker's account. However, this threat can be mitigated using TSS, as the attacker must compromise t+1 parties to execute such an attack. Additionally, the system can fortify its defenses by utilizing trusted I/O devices, such as USB dongles, where the crypto wallet operating in the enclave validates the origin of the input. This defense mechanism closely resembles that of a cold wallet but benefits from the privilege segregation provided by the TSS.

Compromising Crypto Wallet (A13) – In our threat model (2.3), we assume

that the crypto wallet operating within our system's enclave is a trusted entity. Any attempt to compromise this honest party would violate our identified security requirements. Thus, we can discuss how our proposed designs can mitigate this threat. Firstly, if the crypto wallet is compromised, the attack surface reverts to a basic TSS environment in which the attacker must compromise t+1 parties to execute their attack. Secondly, we utilize a trusted tunnel to authenticate trusted communications from a broader perspective. If an attacker attempts to compromise a crypto wallet, they must communicate with the enclave and exploit a vulnerability. Leveraging the trusted tunnel, we can ensure that only trusted sources are able to communicate with the crypto wallet, thereby significantly limiting the attack surface. Finally, our system allows us to authenticate trusted sources not only based on their MRENCLAVE and MRSIGNER measurements but also by their name and versions (*ISV_PROD_ID* and *ISV_SVN*). The traffic labeling enables us to discern and ban compromised parties in our network if any exploitations are discovered related to them. Implementing these defensive measures collectively can enhance the security of our system substantially.

Chapter 4

Benchmarks and Evaluation

In this section, we begin by discussing the specifics of the prototype's implementation. Subsequently, we elaborate on the benchmark framework developed for this study, which serves as a means to evaluate the proposed solution. Finally, we present the results of both macro and micro-benchmarks to assess the implemented prototypes.

4.1 Prototype Specification

In the preceding chapter, we presented two implementations to provide an honest environment for the threshold ECDSA signing protocol and subjected them to security analysis. It is evident that the integrated attested tunnel aligns completely with the security requirements outlined in chapter 3. However, the integrated attested tunnel cannot be embedded into TSS client applications without undergoing modification, which necessitates sacrificing compatibility in favor of enhancing security measures. In the ensuing discussion, we explain the specifications of the implemented prototypes.

4.1.1 Transparent Attested Tunnel

The transparent attested tunnel was implemented in the Golang programming language, which provides a native concurrency feature that is vital for the trusted tunnel's operation. To facilitate the deployment of unmodified TSS client applications within an enclave, we utilized version 1.4 of Gramine LibOS [22]. Given the performance overhead associated with the enclave execution, we configured Gramine's manifest file to leverage the exitless feature [79], which can improve performance by up to 58% [80]. However, for the purpose of comparison, we also included the results without the exitless feature in the benchmarks. In developing the prototype of the transparent attested tunnel, we utilized Pion's DTLS 1.2 implementation in Go [81] and Go's native library of TUN/TAP interfaces [82]. The ra-tls library [83],

embedded within the Gramine LibOS, was employed for key generation and remote attestation.

To deploy an honest TSS party with a transparent attested tunnel, users must run a provided Docker image that contains several essential components, including the unmodified TSS application binary, the data file generated by Gramine after pre-processing and signing the manifest file, the transparent attested tunnel, and an Allowlist. The shell script initiates the deployment process by generating a private key and certificate, which are used as the honest party's credentials to initialize the transparent attested tunnel. Once the tunnel is established, the TSS client application is started, and TSS messages are passed through the trusted tunnel using a virtual IP of the TUN device. For authentication of honest TSS parties, the trusted tunnel uses EPID Provisioning and Remote Attestation [17] technique. After a valid attestation to verify the SGX quotes in the X.509 certificate, the trusted tunnel checks the measurements with the Allowlist provided along with the other components. Overall, This process ensures that only legitimate TSS parties are authenticated and allowed to communicate through the trusted tunnel.

4.1.2 Integrated Attested Tunnel

The integrated attested tunnel is part of our implementation of TSS client application. Specifically, we have developed a prototype of a threshold ECDSA signing client that includes a trusted tunnel for the secure transfer of TSS messages, as well as an authentication mechanism to verify the identity of honest parties. Our TSS client bundle comprises a TSS engine, session management layer, transport layer, and trusted tunnel, all implemented in Golang programming language. The TSS engine is based on the GG18 [11] protocol, specifically tss-lib [15] library, Binance's open-source implementation of GG18. Meanwhile, the transport layer uses the built-in TLS library of Go, namely crypto/tls [84]. The session management layer also coordinates sending and receiving of TSS messages between parties since the order of messages must be preserved. Similarly, key generation and remote attestation are performed using the ra-tls library in Gramine LibOS. It is worth noting that our prototype leverages Gramine LibOS to protect an honest party inside an enclave. In total, the implemented TSS client prototype contains 1904 lines of code in Golang

[85], with cgo utilized to access the ra-tls library for handling remote attestation and credential generation.

As outlined in previous sections, the prototype of TSS client is shipped via a Docker image that incorporates the TSS client, Gramine LibOS, Gramine's dependencies, and the Allowlist. The initial entry point for the Docker container executes our TSS client prototype within an enclave. Technically, the first step involves generating a private key and X.509 certificate that contains SGX quotes. Subsequently, the trusted tunnel is initialized with the generated certificate and starts listening on a designated port. Ultimately, the TSS engine sends TSS messages to other parties in the TSS protocol. All transmitted messages traverse the transport layer and are secured within the trusted tunnel. Upon receiving the TSS message, an honest TSS party conducts remote attestation, EPID in our case, and Allowlist verification to detect the identity of the communicating party and establish the secure channel. It is necessary to mention that the session management layer wraps TSS messages in session data and buffers received messages to assure that the TSS round associated with the received message is less than or equal to the TSS round of the party.

4.2 Benchmark Framework

This section provides a comprehensive view of the benchmarking process employed in this study. Firstly, we describe the specific criteria that were used to evaluate the prototypes. Secondly, we present the benchmark framework that was developed to facilitate the benchmarking procedure. Lastly, we introduce the specifications of the systems on which the benchmarks were conducted to ensure the validity and reliability of the findings.

To evaluate the effectiveness and overhead incurred by the trusted tunnel, we designed micro and macro-benchmarks. In macro-benchmarks, we assessed the impact of the trusted tunnel and enclave on network performance and TSS protocol. For the micro-benchmark, we measured the execution time of different phases of the trusted tunnel to estimate incurred overhead.

Regarding the transparent attested tunnel, our study focuses on two primary macro-benchmarks. Firstly, we assessed the download throughput for a single UDP connection. To achieve this, we utilized iperf3 [29], a network throughput test tool.

We deployed an iperf3 server on a container without SGX capabilities; however, the iperf3 client container can run in an enclave, depending on benchmarks. The throughput was measured over a 10-second interval, with bandwidth gradually increased from 100MiB to 1GiB. Additionally, we incorporated the TSS protocol macro-benchmark to calculate the duration of DKG and threshold signing with varying numbers of participants. The number of parties involved in the protocol increases incrementally from 3 to 9, with the threshold t being defined as two-thirds of the total participants minus one. For instance, for 3 parties, the threshold is 1, so we need at least 2 parties to sign a message collaboratively. To evaluate the performance of the integrated attested tunnel, we conducted macro- and micro-benchmarks. Firstly, we benchmarked the DKG and distributed signing operations. Furthermore, we designed a micro-benchmark to assess the time overhead incurred by the trusted tunnel. In order to achieve this, we inserted probes at strategic points within the TSS party prototype, enabling us to measure the duration of each phase in its lifecycle associated with the trusted tunnel. This measurement allowed us to identify the root cause of any delay in the DKG and threshold signing processes, providing us with valuable insights.

We developed a benchmark framework to conduct the benchmarks mentioned above. In the initial stage of our testing, we classify the benchmarking into four levels: native, gramine, tunnel, and exitless variation of enclave-based execution. The "native" level involves benchmarking the unmodified version of the application. The "gramine" level involves running the benchmarks on Gramine LibOS without the attested tunnel. The "tunnel" level indicates that the benchmark runs on top of Gramine LibOS and utilizes the attested tunnel for the secure channel and peer authentication. Lastly, the "exitless" variation denoted that the "Exitless Feature" of Gramine LibOS is set up. By comparing the results of these benchmarks, we are able to attribute any overhead to the enclave execution, specifically to the trusted tunnel; in better words, this comparison allows us to identify the source of any performance issues and make system improvements.

To implement the benchmark framework, we utilized shell scripts and Docker Compose to automate the benchmarking process. For each class of benchmarks, we developed a controlling shell script located at the root of its directory. This script was responsible for starting and stopping the containers and passing parameters to the

internal scripts (/scripts directory) running inside the container. Docker Compose is instrumental in setting up the benchmark environment in a reproducible manner, delivering what is known as Infrastructure as Code (IaC). It provides the necessary network, disk, and device resources to containers to communicate and store the results. The scripts directory on each category of benchmarks contains scripts executed inside a container or an enclave. Figure 4.1 accurately demonstrates the relationship between these three components. First, the controlling script runs containers based on the predefined environment in Docker Compose and passes the required parameters to internal scripts. After starting the containers, the container runs the internal scripts containing the actual commands to be benchmarked. This hierarchy allows us to simulate the desired environments and easily add new benchmarks without changing the entire structure and inventing something new.

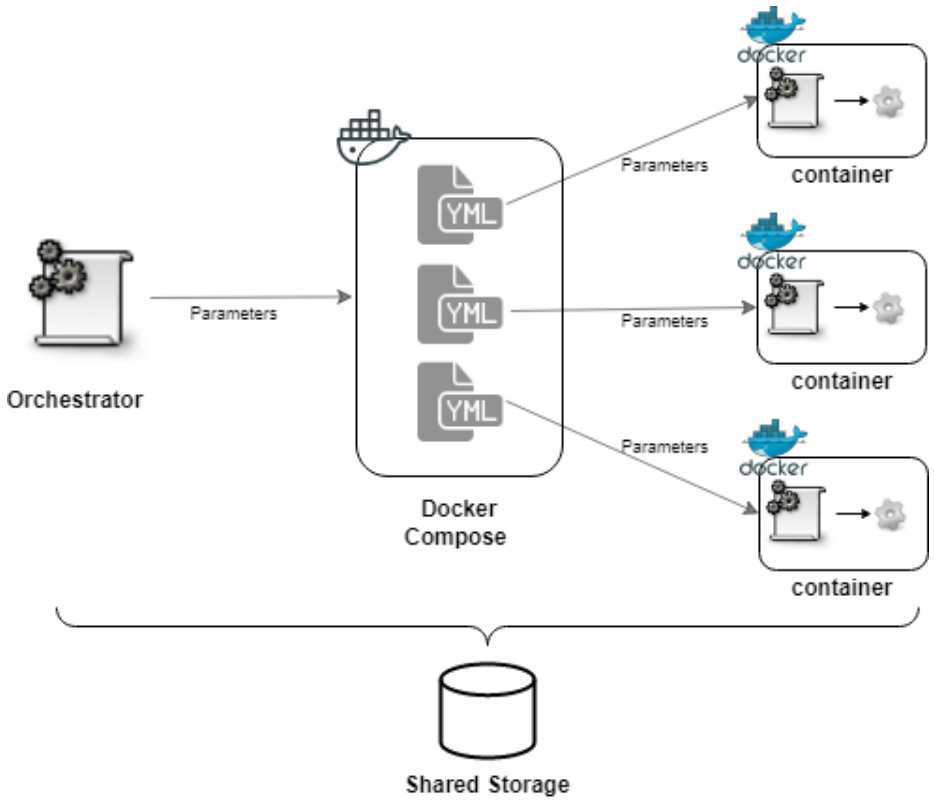


Figure 4.1: The Architecture of Benchmarking Framework

The experimental setup for this study was carefully designed to ensure reliable and accurate results. The benchmarking framework was executed on a high-performance

workstation equipped with an Intel(R) Core(TM) i9-9900KF CPU with 8 cores, 128GB of memory running at 2666MT/S, and 256GB of SSD. The workstation runs Ubuntu 20.04.5 LTS with a kernel of 5.13.5-051305-generic with SGX-related features enabled. The base docker image used in the framework is also Ubuntu 20.04 LTS, which installs the necessary packages on demand. It is noteworthy that the workstation is connected to the local network via a 1Gbps NIC, Intel I219-V, although no NICs are utilized in the benchmarks. Instead, all the containers use the default bridge driver, which enables the building of a local network in the Docker host, providing namespace isolation from the host's network namespace. In the subsequent section, we will present and analyze the results obtained from the benchmarks.

4.3 Evaluation

4.3.1 Transparent Attested Tunnel

As previously indicated, our study employed two macro-benchmarks to evaluate the overhead imposed by the transparent attested tunnel. The initial benchmark assessed the throughput of the download UDP link under three modes: native, gramine, and tunnel. Subsequently, we conducted a second benchmark to measure the duration of DKG and signing across the same three modes: native, gramine, and tunnel. It should be noted that we included a tunnel mode that featured the "Exitless Feature" to demonstrate the potential impact of utilizing enclaves without this feature.

iperf3 Macro-benchmark - The network performance metric is crucial in gauging the potential overhead of deploying a trusted tunnel within an honest environment, given that all traffic is transmitted via the trusted tunnel to reach its destination. Considering this, we conducted measurements of the maximum UDP downlink throughput, employing the iperf3 tool. The iperf3 client and server containers were set up in different modes, namely, native, gramine, and tunnel. We recorded the average downlink throughput in 10-second intervals, and the benchmark was executed for a duration of 30 seconds. To account for the impact of bandwidth on throughput, we gradually increased the bandwidth limit in increments of 100Mbit/s for every step. Our measurements covered the maximum UDP downlink throughput under bandwidth limits ranging from 100Mbit/s to 1Gbit/s. As illustrated in Figure 4.2, native

and gramine benchmarks demonstrated similar throughput, whereas tunnel exhibited an overhead of approximately 5.5% at the peak of the graph. This overhead can be attributed to the latency imposed by attestation and DTLS tunnel. It is worth noting that the Exitless Feature significantly enhances the throughput in which when the client runs inside an enclave without Exitless Feature, the throughput drops around 33%.

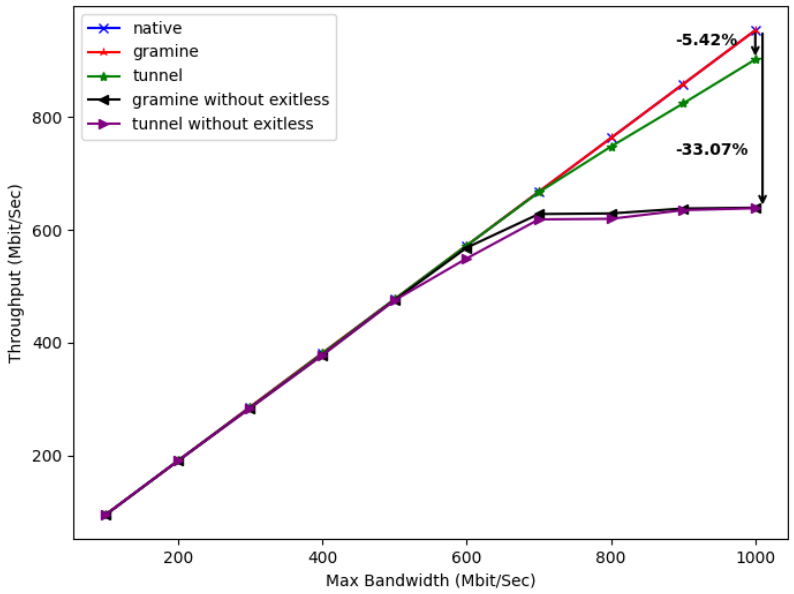
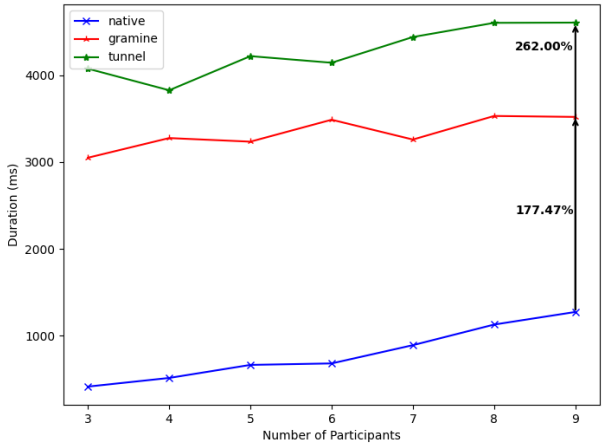


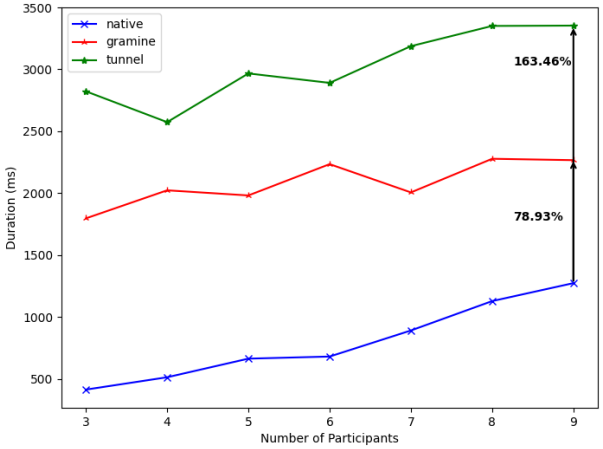
Figure 4.2: Network Performance of Transparent Attested Tunnel

Threshold ECDSA Macro-benchmarks - In our second macro-benchmark, we measured the time required for DKG and threshold ECDSA signing. We structured the benchmark such that the number of participants progressively increased from 3 to 9, with a threshold of two-thirds of the parties minus one, which translates to the requirement of at least two-thirds of the players to contribute to signing a message or transaction. To conduct this benchmark, we divided it into two distinct phases: the key generation phase and the signing phase. We recorded the results of measurements for each phase, such as duration, key shares, error logs, and signatures onto the disk. Moreover, as previously mentioned, we utilized ZenGo's Multiparty-ECDSA implementation for this benchmark, whereby we commenced by running the

coordinator node, then adding the participants for each phase. We use gg20 version of Multiparty-ECDSA which is based on [12].



(a) DKG Execution Time with Enclave Initialization



(b) DKG Execution Time without enclave Initialization

Figure 4.3: DKG Macro-Benchmark of Transparent Attested Tunnel

As illustrated in Figures 4.3, running the threshold ECDSA client inside an enclave on LibOS, gramine benchmark, results in a significant delay in comparison to the native benchmark. This delay is primarily due to the initialization of the enclave and the loading of the binary into it, which are necessary steps when a program runs inside an enclave. It is worth noting two crucial points in this regard. Firstly, the initialization time occurs only once during the lifetime of a process; thus, we can expect that the actual execution time of a process in an enclave, without initialization, would

be considerably less than the duration we observed with initialization. Secondly, the initialization time is closely tied to the details of the manifest file. Specifically, by restricting the manifest file in terms of ensuring the integrity and confidentiality of the disk content, enclave features, and so forth, we can achieve a shorter initialization time. Considering this fact, we measured the initialization of enclave in which the enclave just runs the Multiparty-ECDSA binary to print the help options. After measuring the init time, we decrease this value from the whole execution time to estimate the actual duration of TSS protocols in an enclave (Figures 4.3b, 4.4b). We will delve into this notion more when we talk about the micro-benchmarks of the integrated attested tunnel. As shown in 4.3b, we experience an overhead of 79% when we run Multiparty-ECDSA player inside an enclave, gramine benchmark. Additionally, in terms of tunnel benchmark, we experience 84% more overhead than gramine benchmark. The reasons for the observed overheads are as follows:

- First of all, The DKG phase, a critical component in the TSS protocol, requires the computation of large prime numbers for the public/private shares. This phase also involves resource-intensive tasks such as zero-knowledge proofs. When executing such computations within an enclave that necessitates memory encryption and ECALL/OCALL instructions, there is a noticeable overhead that has been quantitatively documented and illustrated in the accompanying chart. Additionally, as we discussed in Chapter 2, the key generation of TSS protocol is a probabilistic algorithm that relies on the strength of entropy, and entropy is a measure of the degree of randomness in a system, and a high level of entropy is crucial in ensuring the security of cryptographic protocols. Therefore, the strength of entropy is important in the key generation phase of a TSS protocol since it determines the quality of the generated keys. It is aligned with what is depicted in Figure 4.3. While the overall DKG execution time is increasing, there are some skewed points. This can be attributed to the indeterministic aspect of DKG and the impact of the strength of entropy in the key generation. As such, it is crucial to take into account these computational costs when analyzing the overhead associated with the trusted tunnel.

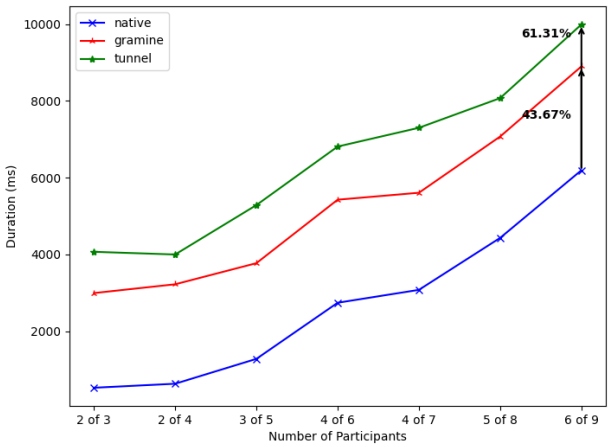
- Secondly, Multiparty-ECDSA application does not utilize any secure tunnel mechanism to exchange TSS messages between parties. Consequently, native

and gramine benchmarks just use bare TCP protocol as the transport layer. In contrast, the transparent attested tunnel leverages the Datagram Transport Layer Security (DTLS) protocol to establish a trusted tunnel between TSS parties and authenticate honest parties. According to this notion, This additional layer of security incurs an overhead compared to the native and gramine benchmarks, as depicted in the accompanying Figure 4.3.

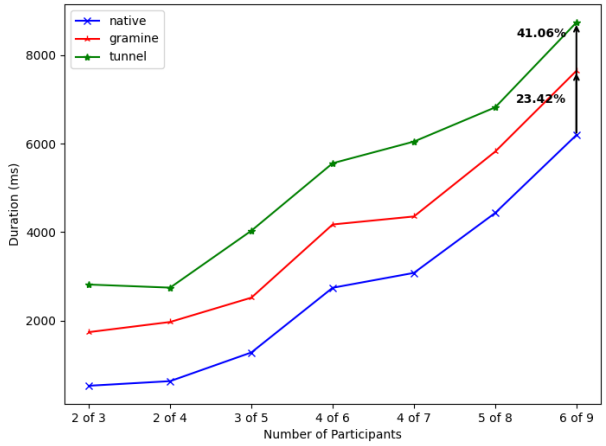
- The transparent attested tunnel leverages remote attestation to verify the provided quotes in the Transport Layer Security (TLS) certificate which ensures the authentication of honest parties involved in the TSS protocol. Remote attestation, including EPID, is not a free process and adds an overhead to the tunnel benchmark on top of what we discussed previously.

In the signing phase of the Threshold ECDSA, as depicted in Figures 4.4b, the gramine and tunnel benchmarks exhibit an overhead compared to native execution of Multiparty-ECDSA which are 23% and 41% respectively. For the tunnel benchmark, this overhead can be attributed to several factors, including protecting confidentiality and integrity provided by the DTLS layer and detecting honest parties provided by the remote attestation mechanism. However, in contrast to the DKG phase, the signing phase is a deterministic process. As such, we do not observe skewed points that were observed in the DKG phase. Instead, the execution time of the signing phase increases as the number of participants increases, which is in line with the fact that when the threshold increases, the protocol requires more TSS message exchanges to sign a message.

Furthermore, it is worth noting that the signing phase is less computing-demanding compared to the DKG phase. As a result, the overhead of using the enclave in the signing phase, depicted in gramine and tunnel benchmarks, is less than in the DKG benchmark. According to Figure 4.4b, the overhead of tunnel benchmark for DKG with nine parties was approximately 163%, while for the signing of six out of nine parties, it was 41% compared to the baseline, native execution.



(a) Signing Execution Time with Enclave Initialization



(b) Signing Execution Time without enclave Initialization

Figure 4.4: Signing Macro-Benchmark of Transparent Attested Tunnel

We can attribute this drop to two factors: 1) the number and size of messages: In [12] protocol which is used in Multiparty-ECDSA, DKG phase has 4 rounds while the signing phase has 7 rounds. It means that bigger and more messages are required to be exchanged in DKG phase which can be translated into a higher rate of encryption and decryption. 2) Remote Attestation: the remote attestation based on EPID is a costly operation compared to other operations in this system because parties need to connect to IAS to validate quotes, and it takes hundreds of milliseconds while it takes just less than milliseconds to validate a certificate using CA roots. Accordingly, in DKG phase, a party needs to complete 8 remote attestations, but it requires only 5

remote attestation operations in the signing phase. These observations highlight the fact that remote attestation and DTLS channel have essential roles in the overhead of the transparent attested tunnel compared to gramine benchmark.

Upon examination of the provided observations, we can draw the conclusion that the primary overhead associated with the transparent attested tunnel compared to the native execution stems from the enclave initialization process, which occurs only once during the execution of an honest party. Furthermore, the delay observed in tunnel mode, as opposed to gramine mode, is primarily attributable to the attestation process and DTLS tunnel. This delay can be reduced by deploying Data Center Attestation Primitives (DCAP) technique, optimizing the trusted tunnel code, and utilizing SGX directly instead of using Gramine LibOS. In the next section, we shall employ micro-benchmarks in the TSS client application with the integrated attested tunnel in specific to accurately measure the duration of each phase in an honest party.

4.3.2 Integrated Attested Tunnel

As mentioned earlier, to meet the security requirements introduced in Chpater 3, we implemented the prototype of TSS client with the integrated attested tunnel. In this section, we present the prototype's evaluation in terms of DKG and signing macro-benchmarks which measure the execution time of DKG and signing phases in GG18 [11] threshold ECDSA protocol. Moreover, we provide a micro-benchmark in which we gather data from probes implanted in the TSS client code.

TSS Client Macro-benchmarks – In order to assess the overhead associated with an integrated attested tunnel, we developed two macro-benchmarks to measure the execution time of the DKG and signing phases. As DKG is an indeterministic function, its execution time varies between each iteration, which can make it difficult to compare different benchmarks. To address this issue, we designed a separate phase called *genPreParams* mode in our prototype, in which each party calculates its required cryptographic parameters beforehand. The results of this phase are pre-calculated required parameters that are stored on disk. It is worth noting that the execution time of generating pre-parameters was not included in the DKG benchmark. Overall, *genPreParams* mode offers a way to standardize the DKG benchmark for fair comparisons.

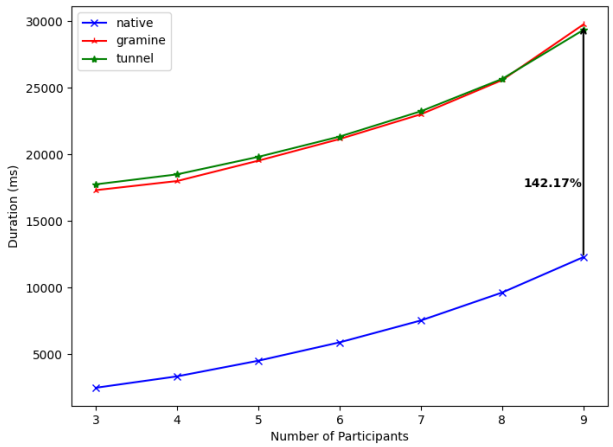
In the DKG benchmark, we instantiate N number of TSS clients in containers with the required parameters. We increase N from 3 to 9 to measure the impact of the number of participants in our benchmark. Similar to what we did in the transparent attested tunnel, we provide three categories of environments to conduct our benchmark: native, gramine, and tunnel. The specifications of these categories are as follows:

- native – TSS client runs on a container with tlsMode of *tlsWithCAs* and a provided TLS key/cert pair. It means that TSS client needs to verify the certificate of peers using the given certificate authority in the config file; additionally, it must use a pre-generated pair of TLS key and certificate provided in the command line to establish a TLS connection.

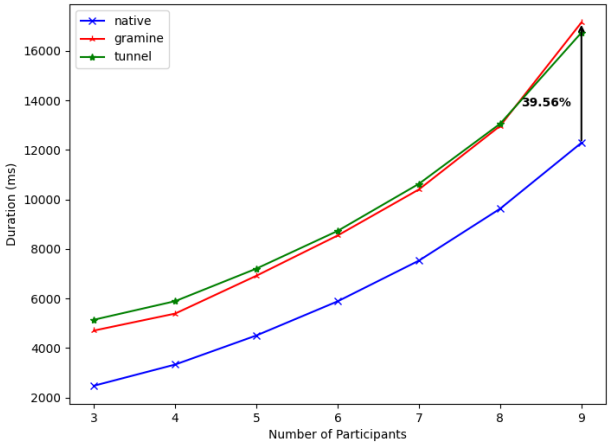
- gramine – TSS client runs inside an enclave utilizing gramine. It is also provided with *tlsWithCAs*, key, and cert parameters. Consequently, the gramine environment establishes TLS connection with the same TLS configurations as the native configuration.

- tunnel – TSS client runs inside an enclave with tlsMode of *aTls*. It means that TSS client needs to generate its own TLS key and certificate to establish a TLS connection to other peers. Additionally, this pair must be signed by SGX Provisioning Enclave to be valid.

All other parameters of TSS clients are the same in each environment. To calculate the DKG execution time, we capture the epoch time before and after the execution of DKG phase and consider their differences as an instance of DKG execution time. Figure 4.5 depicts the DKG execution time for 3 to 9 participants in 3 aforementioned environments. It is necessary to point out that we considered the average of 5 instances for each case as the final value of DKG execution time; moreover, native execution reflects the baseline in our analysis.



(a) DKG Execution Time with Enclave Initialization



(b) DKG Execution Time without enclave Initialization

Figure 4.5: DKG Macro-Benchmark of TSS Client with Integrated Attested Tunnel

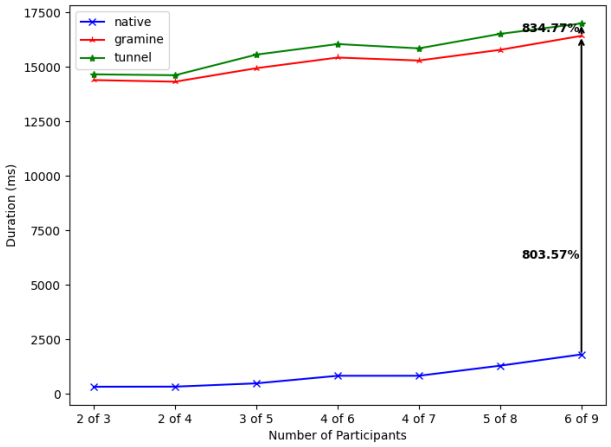
We can derive several observations from the data presented in Figure 4.5. Firstly, it is evident that the execution time of DKG increases in direct proportion to the number of participants, as anticipated. This is a result of the increased number of message exchanges required as the number of participants grows. Secondly, it can be observed from Figure 4.5a that initializing the enclave results in a substantial overhead of 142% for 9 participants. This overhead can primarily be attributed to the enclave initialization, as indicated in 4.5b, since the overhead drops to 39% after deducting the enclave initialization time. Notably, the difference between the overhead of the tunnel and gramine is around 18% for 3 participants, and this difference

decreases as the number of participants increases. This can be attributed to the fact that as the overall DKG execution time increases, the impact of latency associated with key generation and remote attestation in tunnel becomes relatively insignificant. Finally, it is observed that for 9 participants, the tunnel benchmark shows better performance than the gramine benchmark. This behavior can be explained by two possible factors: Firstly, when multiple parties verify the validity of a certificate by performing remote attestation, the IAS caches the results, reducing the latency associated with remote attestation. Secondly, although the main computations related to the DKG phase are conducted during the 'PreParams' phase, there are still some probabilistic computations that need to be performed during the DKG phase. The impact of these computations may be greater than the self-signed generated attested TLS (aTls) key/cert and remote attestation in the DKG.

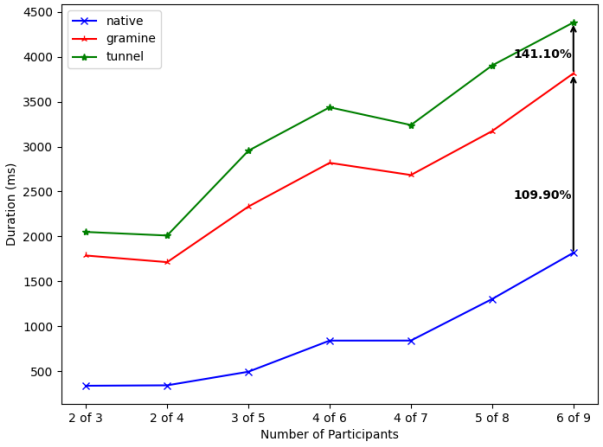
In order to benchmark the signing phase, we have considered two-thirds of the parties involved in the DKG phase. Specifically, we have set the threshold t to be equal to '$(\frac{2}{3})N-1$', where N represents the total number of participants in the DKG phase. The native, gramine, and tunnel environments were set up with the same configurations used for the DKG benchmark. To measure the execution time of the signing phase, we recorded the epoch time before and after running TSS client in Sign mode. The results of these measurements are presented in Figure 4.6, which shows the execution time of the signing phase for values of N ranging from 3 to 9, with a threshold '$(\frac{2}{3})N-1$'. We should note that we have utilized the mean of five instances for each case to get the execution time of the signing phase. Furthermore, we set the native execution as the baseline for our analysis.

Several observations can be made from Figure 4.6. Notably, the signing phase in gramine and tunnel benchmarks incurs significant overhead, as indicated by Figure 4.6a. This overhead can be attributed to the enclave initialization time, which is considerably higher than the signing duration in the native environment. As shown in Figure 4.6b, upon deducting the duration of enclave initialization in gramine, this overhead drops to 110%, eight-fold less than the value illustrated in Figure 4.6a. Additionally, in all three cases, the duration of the signing protocol increases as the number of participants increases, consistent with our expectation in terms of the number of TSS messages exchanged between participants. While the required number

of participants remains constant and the total number of parties generating key shares increases, the execution time slightly decreases, suggesting that the computation that tss-lib requires to make the signature is reduced in these cases. Moreover, another key observation in Figure 4.6b is that the difference between overhead associated with tunnel and gramine benchmarks is nearly the same, and for $(t, N) = (5, 9)$, it is approximately 31%. This difference can be attributed to the generation of a self-signed aTls key/cert pair and remote attestations performed in the tunnel benchmark.



(a) Signing Execution Time with Enclave Initialization



(b) Signing Execution Time without Enclave Initialization

Figure 4.6: Signing Macro-Benchmark of TSS Client with Integrated Attested Tunnel

TSS Client Micro-benchmarks - To determine the underlying causes of overhead incurred in the gramine and tunnel benchmark, we employed a series of probes within the TSS client code to accurately measure the execution time of specific tasks. These probes were activated by providing the *-microbench* option to the TSS client programs. The configuration of the native, gramine, and tunnel environments remained consistent what we provide in the macro-benchmarks; additionally, the benchmark consisted of three distinct phases: pre-parameter generation, key share generation, and message signing. We conducted the micro-benchmark for three parties and separately collected the results for the DKG and signing phases. Table 4.1 represents the results of the micro-benchmark for the DKG phase.

Microbenchmark	**native**	**gramine**		**tunnel**	
	Duration(ms)	Duration(ms)	Overhead	Duration(ms)	Overhead
Load TSS Client and TLS Channel Configs	0.093	4.099	4307.53%	1.802	1837.63%
Generate aTls Certificate	N/A	N/A	N/A	313.076	N/A
Load Pre-Params	0.147	1.458	891.84%	1.565	964.63%
Verify Certificate (Max)	0.413	0.903	118.64%	672.260	162674.82%
Write DKG Share to Disk	0.086	1.928	2141.86%	6.492	7448.84%
Generate DKG Share	2560	2930	14.45%	2868	12.03%
Whole Execution	2562	2951	15.18%	3190	24.51%

Table 4.1: DKG Micro-benchmarks of TSS Client with Integrated Attested Tunnel

Microbenchmark	**native**	**gramine**		**tunnel**	
	Duration(ms)	Duration(ms)	Overhead	Duration(ms)	Overhead
Load TSS Client and TLS Channel Configs	0.090	3.776	4095.56%	3.699	4010.00%
Generate aTls Certificate	N/A	N/A	N/A	263.628	N/A
Load Key Share	0.297	1.775	497.64%	2.087	602.69%
Verify Certificate (Max)	0.278	0.889	219.78%	644.61	231774.10%
Generate Signature	298.034	389.764	30.78%	716.302	140.34%
Whole Execution	298.907	400.771	34.08%	990.402	231.34%

Table 4.2: Signing Micro-benchmarks of TSS Client with Integrated Attested Tunnel

Table 4.1 presents important observations. First, the implemented TSS client prototype incurs significant overhead in I/O operations when it operates inside an enclave, as evident from the results of the "Load TSS Client and TLS Channel Configs" "Load Pre-Params" and "Write DKG Share to Disk" micro-benchmarks. This finding aligns with our expectations because Golang does not utilize glibc to issue syscalls and instead issues syscalls directly [86]. Conversely, Gramine LibOS handles syscalls by replacing them with a specific call instruction through patching glibc and musl C libraries. Therefore, when a Golang program calls a syscall, many context

switches occur between the kernel, LibOS, and the program, resulting in significant performance loss [87]. Consequently, a performance drop is expected when the TSS client conducts I/O operations. It is worth noting that similar behavior is observed in the "Load TSS Client and TLS Channel Config" and "Load Key Share" micro-benchmarks in the signing phase (Table 4.2).

Another notable observation pertains to the "Generate aTls Certificate" and "Verify Certificate (Max)" micro-benchmarks. Since a party needs to verify the TLS certificate of peer parties in the TSS protocol, it needs to do a certificate validation for each, so the log displays multiple "Verify Certificate" values. Moreover, certificate validation can be considered a concurrent operation whereby for each TLS connection, a party runs a goroutine to validate the corresponding certificate. Consequently, the "Verify Certificate (Max)" value is defined as the maximum duration of certificate verification among all TLS connections in our benchmark. On the other hand, both the native and gramine environments do not require generating a SGX self-signed certificate to establish a TLS channel, so the "Generate aTls Certificate" value is inapplicable to these environments. Considering all these factors, the tunnel environment is expected to experience more overhead because it must first generate an aTls certificate and then perform remote attestation to verify the provided TLS certificate.

Finally, based on the recorded benchmark results, the actual execution time of the TSS client in an enclave is significantly lower than what was calculated in the macro-benchmark. Since enclave initialization occurs only once in its lifetime, the effective execution time of the TSS client can be considered the "whole execution" presented in the micro-benchmarks. Accordingly, the tunnel environment incurs a 24% overhead in the DKG phase (Table 4.1), which is acceptable given the security benefits proposed in our architecture and security requirements. In the signing phase, while gramine environment experiences a 34% overhead compared to the native baseline, the integrated attested tunnel experiences a 231% overhead for two signers with a total of three players. Based on the data provided in Table 4.2, most of this overhead can be attributed to generating aTls certificate and remote attestation operations. According to the the signing macro-benchmark 4.6, the duration of the "Generate Signature" operation increases as the number of participants grows, making the impact

of the tunnel's delay minor and negligible. Overall, we can conclude that the performance of the TSS client with the integrated attested tunnel is acceptable, considering the security advantages it provides to a crypto wallet.

After considering the observations discussed above, we can propose some techniques to enhance the performance of the TSS client with the integrated attested tunnel. In order to optimize the performance of the TSS client itself, a recommended approach would be optimizing the code to eliminate bottlenecks, such as excessive I/O operations. Additionally, optimization of the manifest file of Gramine could decrease the initialization time of enclaves. To address the problem associated with issuing syscalls in Golang on Gramine, a possible solution is to patch the Go compiler to use glibc instead of direct syscalls [88]; however, this task is labor-intensive and requires continuous maintenance. An alternative and intriguing option would be to use SGX-SDK written in Go, such as EGo [89], which eliminates Gramine LibOS from the architecture and reduces the overhead caused by calling syscalls and their excessive context switches. Furthermore, it facilitates fine-tuning of the SGX environment based on specific requirements and eliminates unnecessary generic routines implemented by Gramine LibOS. However, with this new architecture, the developer is responsible for initializing, starting, and managing enclaves in their code. Since the tss-lib library is written in Golang, the TSS client also needs to be developed in Golang. Nevertheless, there are some TSS libraries written in other languages, like Rust [14], which can be utilized to write code without encountering the challenges associated with Golang. We utilized SGX1 without Enclave Dynamic Memory Management (EDMM), where all SGX parameters had to be specified in advance in Gramine's manifest file. In contrast, SGX2 has the EDMM feature, which allows for dynamic addition and removal of enclave memory during runtime. It also provides the ability to change memory permissions and types and create dynamic threads. As the initialization of the enclave is closely related to the size of the enclave and the number of threads specified in the manifest file, this capability of SGX2 helps considerably reduce initialization time since Gramine can add memory pages and thread control structure at runtime.

To enhance the performance of the integrated attested tunnel, we need to address the latency of generating aTls certificate and remote attestation. Gramine uses

mbedtls library under the hood to implement ra-tls mechanism, and most of its latency is attributed to high startup time, which is the result of default seeding of random number generator using */dev/random* and */dev/urandom*; accordingly, to reduce the latency associated with generating aTls certificate, the SGX signed certificate, we can force to employ RDRAND/RDSEED engine to reduce the setup time [90]. Finally, the remote attestation is an essential component of our design which provides mutual authentication and allows only honest parties to participate in the protocol; however, it incurs some overhead to our system. In the implemented prototypes, we used EPID remote attestation method, which needs communications with IAS infrastructure. To minimize this overhead, we can deploy the DCAP method of remote attestation in which we set up internal attestation service (AS) servers with caching capability. This technique can reduce the latency associated with remote attestation considerably.

Chapter 5

Conclusion

In this thesis, we focused on securing multi-party crypto wallets relying on threshold ECDSA signing. Organizations and exchanges widely use these wallets to benefit from their usability and enhanced security features. However, the inherent complexity of the protocol, coupled with the immaturity of their implementations, makes these wallets vulnerable to attacks and loss of millions of funds. Given the severity of potential attacks, we propose security solutions based on Intel's implementation of TEE, SGX, to mitigate attacks against multi-party crypto wallets. Our work focuses on the key generation and transaction signing domains while leaving other aspects of crypto wallets for future works. Moreover, given the widespread usage of the ECDSA signing algorithm in the blockchain, we limit our work to threshold ECDSA signature and DKG protocols, specifically the GG18 and GG20 protocols that are highly respected in the crypto industry.

As outlined in the threat model section, most attacks on TSS (Threshold Secret Sharing) protocols involve participating dishonest parties and exploiting the protocol through multiple rounds of communication. Accordingly, we proposed two solutions in which honest parties can detect dishonest players and prevent them from exploiting the protocol. In these solutions, we bound the identity of TSS players to the code and data of their corresponding processes and verified their identity using hardware, namely Intel SGX. This architecture not only protects multi-party wallet users from attackers exploiting the protocol but also prevents the revelation of their key shares when an attacker conducts a system-level attack and takes control of their systems. To provide a comprehensive security solution, we stated the security requirements of our system, based on the threat model presented in Chapter 2. Following the security requirements for our system, we proposed two designs, each with its advantages and drawbacks. The transparent attested tunnel allows unmodified multi-party crypto wallets to join the TSS protocol, authenticate honest parties, and detect dishonest

parties, but it cannot meet all the security requirements that we propose. Strictly speaking, since the tunnel runs outside an enclave, it is not protected from system-level attacks and consequent threats. In the second solution, we sacrifice flexibility for higher security and implement a TSS client prototype with an integrated attested tunnel feature that runs in an enclave. We provided a comprehensive security analysis, in which, based on the threat model we introduced, we described the attacks and attackers, then discussed the defense mechanisms against the attack and whether the proposed architecture could protect us against these attacks, and finally, how the attacks relate to the provided security requirements. The results of our security analysis show that the integrated attested tunnel meets all the security requirements and can provide a high level of assurance, which is necessary for a multi-party crypto wallet.

To assess the efficacy of the proposed designs, we executed two prototypes and ran macro and micro-benchmarks; moreover, we constructed a framework to combine different benchmarks that cater to our specific requirements. In line with the original design, every player was presented as a container bundle comprising all the necessary packages to operate a multi-party wallet in an enclave. Since the transparent attested tunnel permits the deployment of unmodified programs, we constructed a macro-benchmark leveraging iperf3 to compute the overhead incurred on network throughput when deploying the transparent tunnel. We documented the benchmark results in three environments of native, gramine, and tunnel. Data reveals that tunnel results in a 5.4% overhead compared to the native baseline, which may escalate up to 33% without the Exitless feature of Gramine. Concerning the DKG and signing macro-benchmarks, the use of the tunnel produced a significant overhead; however, most of it pertains to the initialization of an enclave. As enclave initialization transpires once in its lifetime, we can exclude this duration. We discovered that for the DKG, we encountered an overhead of approximately 163% for nine participants and around 61% when six out of nine players signed a message. Regarding the integrated attested tunnel, we executed macro-benchmarks to determine the duration of the DKG and signing phase of the TSS protocol. The results indicate that similar to the other prototype, most of the latency is connected to enclave initialization. Hence, after subtracting the initialization time, we experienced a 40% and 141% overhead,

respectively, for the DKG and signing in the tunnel environment for nine participants and six signers. To uncover the root of these overheads, we introduced a micro-benchmark to gauge the duration of different steps required to complete the DKG or signing. According to our findings in Chapter 4, most of the overhead in our prototype arises from generating SGX-signed TLS certificates and remote attestation to validate the identity of potential participants. Furthermore, due to complications in Golang syscall invocation, the performance of I/O operations of the TSS client in an enclave is notable. The most noteworthy discovery pertains to the effective execution time in tunnel mode, indicating that the actual execution of the TSS client in an enclave is much less than what we calculated in the macro-benchmark. For three players and two signers, tunnel mode leads to a 24% increase in DKG and a 231% increase in signing operations. However, we must consider that the whole execution time of signing is around 300ms in the native environment, while certificate verification takes 600ms in the tunnel environment. It implies that this difference is mostly due to remote attestation, which can be reduced with DCAP and caching techniques. Moreover, we should bear in mind that since the duration of signing increases with an increasing number of signers and participants, the impact of key generation and remote attestation on the overhead would be less significant. Considering these facts, we can conclude that TSS client prototype with the integrated attested tunnel satisfies all the security requirements, and its performance is acceptable, which makes it usable in operational settings.

5.1 Limitations

We have presented two distinct prototypes to mitigate attacks on TSS clients, each with advantages and drawbacks. While the transparent attested tunnel can authenticate honest parties within the TSS protocol without needing application modification, it fails to meet our proposed security requirements. Conversely, the TSS client with the integrated attested tunnel satisfies all security requirements but necessitates modification to the TSS client to be integrated. Therefore, a possible improvement is providing a transparent attested tunnel between clients that meets security requirements. The SENG [69] runtime, which operates on the client side of

its architecture, delivers a transparent shield capable of authenticating network traffic. Another potential improvement is related to the I/O operations in our prototype; strictly speaking, our TSS client prototype is subject to poor I/O performance due to Golang syscall complications. On the other hand, Gramine LibOS lacks support for all the syscalls implemented in the Linux kernel, such as fork() and exec(), which can incur additional overhead under certain conditions. Therefore, using SGX-SDK or other LibOSes [26] may alleviate this issue.

5.2 Future Work

This section will discuss domains and directions we can address to improve our prototypes' security, performance, and useability.

Attestation and Verification - The modification of any components of the TSS client bundle within the provided architecture results in a change in the measurement, report, and verification identity of a TSS party. While updating the Allowlist of all other participants can pose a challenge, this design decision allows for the honest parties' identity to be tightly coupled to hardware and protected from numerous attacks against a software-based identity management system. Striking a balance between the security and usability of prototypes is crucial, and one possible solution is the Panoply approach [91] in which tss-lib is divided into multiple enclaves, including shared libraries and services, and the core tss-client receives services from them through attested Remote Procedure Calls (RPC). Another critical factor in this domain is the security of remote attestation infrastructure. The entire software stack of IAS should strictly be added to the TCB of our architecture since, in both EPID and DCAP remote attestation, we rely on AS to validate the aTls. One possible solution is to distribute this infrastructure among all participants, secured by a TEE. However, this is not a trivial problem since, in the case of IAS, we have the public key of IAS in SGX Platform Software (PSW), but in the case of distributed AS, adding the identity of the counterpart causes a circular dependency that is similar to a deadlock situation. For the same reason, an enclave does not include Allowlist in our architecture. MAGE [92] proposes a solution in which an enclave is divided into two separate parts. The specific part contains the enclave's functionality, while the common part has the information to derive the identity of counterparties.

Multi-party Crypto wallet - Our research proposes a prototype of a multi-party crypto wallet that protects TSS parties from attacking the protocol and revealing key shares. To address the importance and widespread usage of the ECDSA algorithm, we focused on DKG and signing of a message using GG20 and GG18 threshold ECDSA protocols. In practical application, the multi-party crypto wallet must integrate into a blockchain to understand its transaction format. Moreover, it must consider other factors such as refreshing, public verifiable backups, and supporting Hierarchical Deterministic (HD) multi-party wallets. Since the Schnorr signature [93] provides properties such as aggregation that are well-suited for multi-party wallets, it is necessary to consider key generation and signing for EdDSA.

SGX Security - Even though SGX provides hardware isolation and protection for the execution of security-critical code from system-level attacks, it can be attacked [60]. Although many of these attacks require specific conditions to succeed, they are still feasible. Intel typically provides a microcode patch to resolve each attack, but the time between discovery, delivery, and deployment of these patches can provide an opportunity window for attackers to exploit them. Potential solutions to mitigate the risks of such attacks include deploying memory obfuscation techniques such as Oblivious RAM (ORAM) [94] and utilizing co-processors to offload these computations from SGX [95]. In the context of TSS clients with an attested tunnel, if the security of SGX becomes compromised, the whole system's security reverts to the security of the TSS protocol.

TEE Platforms - It is important to note that crypto wallets aspire to replace our traditional wallets, so it is not sufficient to provide security solutions solely for multi-party crypto wallets on workstations. Considering smartphones as a platform to run a TSS client is imperative. For this purpose, it is necessary to port the proposed architecture to ARM TrustZone, and other platforms and frameworks supporting TEE, such as AMD Secure Encrypted Virtualization (SEV), Keystone [96], and Penglai [97].

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[2] "Bitcoin open source implementation of p2p currency." `http://p2pfoundation.ning.com/forum/topics/bitcoin-open-source`. Accessed: 2022-11-23.

[3] "Ethereum whitepaper." `https://ethereum.org/en/whitepaper/`. Accessed: 2022-11-23.

[4] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "Sok: Decentralized finance (defi)," *arXiv preprint arXiv:2101.08778*, 2021.

[5] "Non-fungible tokens (NFT)." `https://ethereum.org/en/nft/`. Accessed: 2022-11-23.

[6] "Decentralized autonomous organizations (DAO)." `https://ethereum.org/en/dao/`. Accessed: 2022-11-23.

[7] C. Polizu, M. Mata, S. Liebowitz, G. Baldassarri, L. Guadagnuolo, A. Birry, , and A. O'Neill, "A deep dive into crypto valuation." `https://www.spglobal.com/en/research-insights/featured/special-editorial/understanding-crypto-valuation`. Accessed: 2022-11-23.

[8] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *International conference on the theory and application of cryptology and information security*, pp. 514–532, Springer, 2001.

[9] J.-P. Aumasson and O. Shlomovits, "Attacking threshold wallets," *Cryptology ePrint Archive*, 2020.

[10] Cryptopedia Staff, "Crypto wallets: Hot vs. cold wallets." `https://www.gemini.com/cryptopedia/crypto-wallets-hot-cold`. Accessed: 2022-11-28.

[11] R. Gennaro and S. Goldfeder, "Fast multiparty threshold ECDSA with fast trustless setup," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1179–1194, 2018.

[12] R. Gennaro and S. Goldfeder, "One round threshold ECDSA with identifiable abort," *Cryptology ePrint Archive*, 2020.

[13] ING Bank, "threshold-signatures: Multiparty threshold ecdsa scheme." `https://github.com/ing-bank/threshold-signatures`. Accessed: 2022-11-28.

[14] ZenGo-X, "multi-party-ecdsa: Multi-party ecdsa." `https://github.com/ZenGo-X/multi-party-ecdsa`. Accessed: 2022-11-28.

[15] BNB Chain, "tss-lib: Multi-party threshold signature scheme." `https://github.com/bnb-chain/tss-lib`. Accessed: 2022-11-28.

[16] D. Tymokhanov and O. Shlomovits, "Alpha-rays: Key extraction attacks on threshold ecdsa implementations," *Cryptology ePrint Archive*, 2021.

[17] V. Costan and S. Devadas, "Intel SGX explained," *Cryptology ePrint Archive*, 2016.

[18] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "Trustjs: Trusted client-side execution of javascript," in *Proceedings of the 10th European Workshop on Systems Security*, pp. 1–6, 2017.

[19] S. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing security and privacy of tor's ecosystem by using trusted execution environments," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 145–161, 2017.

[20] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "Securekeeper: Confidential zookeeper using intel sgx," in *Proceedings of the 17th International Middleware Conference*, pp. 1–13, 2016.

[21] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "SafeBricks: Shielding Network Functions in the Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 201–216, 2018.

[22] "Gramine Library OS with Intel SGX Support." `https://github.com/gramineproject/gramine`. Accessed: 2022-11-28.

[23] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 645–658, 2017.

[24] T. Barabosch and E. Gerhards-Padilla, "Host-based code injection attacks: A popular technique used by malware," in *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pp. 8–17, IEEE, 2014.

[25] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, and others, "SCONE: Secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 689–703, 2016.

[26] "SCONE confidential computing." `https://sconedocs.github.io/`. Accessed: 2022-12-22.

[27] "occlum: Occlum is a memory-safe, multi-process library OS for intel SGX." `https://github.com/occlum/occlum`. Accessed: 2022-12-22.

[28] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij, "Integrating remote attestation with transport layer security," *arXiv preprint arXiv:1801.05863*, 2018.

[29] "iperf3." `https://iperf.fr`. Accessed: 2022-12-22.

[30] Y. Desmedt, "Society and group oriented cryptography: A new concept," in *Advances in Cryptology—CRYPTO'87: Proceedings 7*, pp. 120–127, Springer, 1988.

[31] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*, pp. 207–220, Springer, 2000.

[32] I. Damgård and M. Koprowski, "Practical threshold rsa signatures without a trusted dealer," in *Advances in Cryptology—EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques Innsbruck, Austria, May 6–10, 2001 Proceedings 20*, pp. 152–165, Springer, 2001.

[33] P. MacKenzie and M. K. Reiter, "Two-party generation of dsa signatures," in *Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings 21*, pp. 137–154, Springer, 2001.

[34] A. Lysyanskaya and C. Peikert, "Adaptive security in the threshold setting: From cryptosystems to signature schemes," in *Asiacrypt*, vol. 1, pp. 331–350, Springer, 2001.

[35] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 435–464, Springer, 2018.

[36] J. Doerner, Y. Kondi, E. Lee, and A. Shelat, "Secure two-party threshold ecdsa from ecdsa assumptions," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 980–997, IEEE, 2018.

[37] J. Doerner, Y. Kondi, E. Lee, and A. Shelat, "Threshold ecdsa from ecdsa assumptions: The multiparty case," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1051–1066, IEEE, 2019.

[38] A. Dalskov, C. Orlandi, M. Keller, K. Shrishak, and H. Shulman, "Securing dnssec keys via threshold ecdsa from generic mpc," in *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part II 25*, pp. 654–673, Springer, 2020.

[39] S. Goldberg, L. Reyzin, O. Sagga, and F. Baldimtsi, "Efficient noninteractive certification of rsa moduli and beyond," in *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part III*, pp. 700–727, Springer, 2019.

[40] D. Cozzo and N. P. Smart, "Sharing the luov: threshold post-quantum signatures," in *Cryptography and Coding: 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16–18, 2019, Proceedings 17*, pp. 128–153, Springer, 2019.

[41] N. P. Smart and Y. Talibi Alaoui, "Distributing any elliptic curve based protocol," in *Cryptography and Coding: 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16–18, 2019, Proceedings*, pp. 342–366, Springer, 2019.

[42] G. Castagnos, D. Catalano, F. Laguillaumie, F. Savasta, and I. Tucker, "Two-party ecdsa from hash proof systems and efficient instantiations," in *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pp. 191–221, Springer, 2019.

[43] G. Castagnos, D. Catalano, F. Laguillaumie, F. Savasta, and I. Tucker, "Bandwidth-efficient threshold ec-dsa." Cryptology ePrint Archive, Paper 2020/084, 2020. `https://eprint.iacr.org/2020/084`.

[44] C. Komlo and I. Goldberg, "Frost: flexible round-optimized schnorr threshold signatures," in *Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers 27*, pp. 34–65, Springer, 2021.

[45] R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled, "Uc non-interactive, proactive, threshold ecdsa with identifiable aborts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1769–1787, 2020.

[46] D. Boneh and V. Shoup, "A graduate course in applied cryptography," *Draft 0.5*, 2020.

[47] J.-P. Aumasson, A. Hamelink, and O. Shlomovits, "A survey of ecdsa threshold signing," *Cryptology ePrint Archive*, 2020.

[48] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[49] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pp. 427–438, IEEE, 1987.

[50] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with an honest majority," *Proc. the Nineteenth Annual ACM STOC'87*, pp. 218–229, 1987.

[51] N. Gilboa, "Two party rsa key generation," in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings*, pp. 116–129, Springer, 1999.

[52] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10*, pp. 522–526, Springer, 1991.

[53] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology—CRYPTO'91: Proceedings*, pp. 129–140, Springer, 2001.

[54] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*, pp. 295–310, Springer, 1999.

[55] Coinbase, "Kryptology: Coinbase's advanced cryptography library." `https://github.com/coinbase/kryptology`. Accessed: 2022-4-9.

[56] I. Allison, "Liquid exchange attack: Can a crypto wallet ever be 100 safe from hacks?." `https://www.coindesk.com/tech/2021/08/20/liquid-exchange-attack-can-a-crypto-wallet-ever-be-100-safe-from-hacks/`, Aug. 2021. Accessed: 2022-2-9.

[57] Systems Software and Security Lab, Georgia Institute of Technology, "Sgx 101." `https://sys.cs.fau.de/extern/lehre/ws22/akss/material/gramine-scone.pdf`. Accessed: 2022-4-9.

[58] T. Konopik, "Securing whole applications with sgx," 2022.

[59] A. Gabizon, "On the security of the bctv pinocchio zk-snark variant," *Cryptology ePrint Archive*, 2019.

[60] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of sgx and countermeasures: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–36, 2021.

[61] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs.," in *NDSS*, 2017.

[62] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx.," in *NDSS*, 2018.

[63] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting sgx enclaves from practical side-channel attacks," in *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pp. 227–240, 2018.

[64] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx," *Cryptology ePrint Archive*, 2017.

[65] "Intel® SGX SDK for linux OS." `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html`. Accessed: 2022-8-10.

[66] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[67] "gsc – gramine shielded containers – GSC contributions documentation." `https://gramine.readthedocs.io/projects/gsc/en/stable/index.html`. Accessed: 2022-7-15.

[68] "Manifest syntax – gramine documentation." `https://gramine.readthedocs.io/en/stable/manifest-syntax.html`. Accessed: 2022-7-20.

[69] F. Schwarz and C. Rossow, "Seng, the sgx-enforcing network gateway: Authorizing communication from shielded clients," in *Proceedings of the 29th USENIX Conference on Security Symposium*, pp. 753–770, 2020.

[70] Inc. Verbatim, "lwIP - a lightweight TCP/IP stack - summary." `https://savannah.nongnu.org/projects/lwip/`. Accessed: 2022-8-11.

[71] "Snabb: Simple and fast packet networking." `https://github.com/snabbco/snabb`. Accessed: 2022-8-11.

[72] "DPDK networking." `https://www.dpdk.org/`. Accessed: 2022-8-15.

[73] "Universal TUN/TAP device driver – the linux kernel documentation." `https://docs.kernel.org/networking/tuntap.html`. Accessed: 2022-9-3.

[74] "Attestation and secret provisioning – gramine documentation." `https://gramine.readthedocs.io/en/stable/attestation.html#mid-level-ra-tls-interface`. Accessed: 2022-9-23.

[75] E. Rescorla and N. Modadugu, "RFC 6347: Datagram transport layer security version 1.2." `https://datatracker.ietf.org/doc/rfc6347/`, Jan. 2012. Accessed: 2022-9-8.

[76] "Site to site VPN routing explained in detail." https://openvpn.net/vpn-server-resources/site-to-site-routing-explained-in-detail/. Accessed: 2023-3-10.

[77] "U.S. treasury sanctions notorious virtual currency mixer tornado cash." https://home.treasury.gov/news/press-releases/jy0916. Accessed: 2023-3-20.

[78] T. K. Frederiksen, Y. Lindell, V. Osheter, and B. Pinkas, "Fast distributed rsa key generation for semi-honest and malicious adversaries," in *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II 38*, pp. 331–361, Springer, 2018.

[79] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 238–253, 2017.

[80] "Performance tuning and analysis – gramine documentation." https://gramine.readthedocs.io/en/latest/performance.html#exitless-feature. Accessed: 2023-3-10.

[81] "dtls: DTLS 1.2 Server/Client implementation for go." https://github.com/pion/dtls. Accessed: 2022-10-8.

[82] "water: A simple TUN/TAP library written in native go." github.com/songgao/water. Accessed: 2022-10-25.

[83] "sgx-ra-tls." https://github.com/cloud-security-research/sgx-ra-tls. Accessed: 2022-10-30.

[84] "crypto/tls: Go packages." https://pkg.go.dev/crypto/tls. Accessed: 2023-3-10.

[85] "VS code counter - visual studio marketplace." https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter. Accessed: 2023-3-10.

[86] C. Siebenmann, "Chris siebenmann's blog." https://utcc.utoronto.ca/~cks/space/blog/programming/GoCLibraryAPIIssues?showcomments. Accessed: 2023-3-11.

[87] "Issues preventing running go applications in gramine." https://github.com/gramineproject/gramine/issues/702. Accessed: 2023-3-10.

[88] "Graphenesgx golang support and enhancement." https://github.com/intel/GrapheneSGX-Golang-Support-and-Enhancement/tree/20190715-golang/LibOS/libs/golang. Accessed: 2023-4-9.

[89] "Ego: Building confidential apps in go." `https://github.com/edgelesssys/ego`. Accessed: 2023-3-12.

[90] "Sanitize cpuid flags used by cryptography libraries." `https://github.com/gramineproject/gramine/issues/1014`. Accessed: 2023-4-10.

[91] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves.," in *NDSS*, 2017.

[92] G. Chen and Y. Zhang, "{MAGE}: Mutual attestation for a group of enclaves without trusted third parties," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 4095–4110, 2022.

[93] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, pp. 161–174, 1991.

[94] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine on intel sgx," in *Network and Distributed System Security Symposium*, 2019.

[95] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek, "Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1903–1918, 2020.

[96] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[97] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the penglai enclave.," in *OSDI*, pp. 275–294, 2021.