ROAST in Rust

Implementing and Benchmarking Schnorr Threshold Signature Schemes

Master Thesis

Lukas Leo Schacher

Faculty of Science
at the University of Bern

September 2024

Prof. Dr. Christian Cachin
Mariarosaria Barbaraci

Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Threshold signatures are increasingly important for modern applications like blockchain and secure multi-party computation, with Schnorr signatures being favored for their simplicity and efficiency. FROST was one of the first major works in threshold Schnorr schemes, but it lacks robustness against malicious participants. The purpose of developing ROAST was to improve resilience by guaranteeing the consistent creation of signatures in adversarial environments. This thesis implements ROAST within the Thetacrypt codebase, creating the first robust threshold signature scheme in this framework. We evaluate ROAST's performance in a real-world setting, focusing on latency, reliability and number of signing sessions required to produce a signature. Our benchmarks show that while ROAST increases latency compared to FROST, it significantly enhances robustness without substantial overhead, offering a strong trade-off between security and performance. In light of the National Institute of Standards and Technology's (NIST) current efforts to standardize threshold versions of the EdDSA and Schnorr signatures, we hope this thesis inspires further work, especially in testing these protocols in a real-world setting.

Contents

1 Introduction **1**

2 Background **3**
2.1 Notation . . . 3
2.2 Threshold cryptography . . . 3
2.2.1 Secret sharing . . . 4
2.2.2 Verifiable secret sharing . . . 4
2.2.3 Distributed key generation . . . 5
2.3 Threshold signatures schemes . . . 5
2.3.1 Digital signatures . . . 5
2.3.2 Threshold signatures . . . 6
2.3.3 Interactive vs. non-interactive threshold signing . . . 7
2.3.4 Some security properties . . . 7
2.4 Thetacrypt codebase . . . 8

3 Schnorr threshold signatures **10**
3.1 Schnorr signatures . . . 10
3.2 FROST . . . 11
3.2.1 FROST rounds . . . 11
3.2.2 Practical considerations . . . 13
3.2.3 Limitations . . . 14
3.3 ROAST . . . 14
3.3.1 Key differences to FROST . . . 14
3.3.2 Core functionalities . . . 15
3.3.3 Detecting malicious signers . . . 16
3.3.4 Security properties and complexity . . . 17
3.3.5 Considerations for ROAST in a decentralized deployment . . . 18
3.4 Related work . . . 18
3.4.1 SPRINT . . . 19
3.4.2 The many faces of Schnorr . . . 19
3.4.3 HARTS . . . 19
3.4.4 Arctic . . . 20

4 Implementation **21**
4.1 Overview . . . 21
4.2 Deviations from original ROAST . . . 21
4.2.1 Start signal . . . 22
4.2.2 Marking malicious participants . . . 22
4.2.3 Verification and sending the final signature to participants . . . 22
4.2.4 Performance optimization . . . 22

4.3 Implementation details . . . 23
4.3.1 Coordinator . . . 23
4.3.2 Rounds . . . 23
4.3.3 Message content . . . 26
4.4 Setup for benchmarking experiments . . . 27
4.4.1 Choosing the coordinator . . . 27
4.4.2 Simulating malicious parties . . . 27

5 Benchmarking **30**
5.1 Methods . . . 30
5.1.1 High-level overview . . . 30
5.1.2 Benchmarking parameters . . . 30
5.1.3 Benchmarking client setup . . . 33
5.1.4 Infrastructure configuration . . . 33
5.2 Results . . . 35
5.2.1 Instance completion rates . . . 35
5.2.2 Latency at the server side . . . 37
5.2.3 Number of signing sessions for ROAST signature . . . 40

6 Conclusion **43**
6.1 Contributions . . . 43
6.2 Future work . . . 43

A ROAST pseudocode **49**

B Monitoring **52**

C Extended results **53**
C.1 Server-side latency . . . 53
C.2 Instance completion rates . . . 54

1 Introduction

Threshold cryptography has become increasingly important in modern applications, such as blockchain technology and secure multi-party computation. Schnorr [Sch89] signatures are particularly interesting because they do not require pairings, unlike, e.g., BLS signatures [BLS04], which leads to improved efficiency and simplified implementation. This simplicity, combined with the adoption of Schnorr signatures in blockchain technologies like Bitcoin [Nak09, MPSW19], motivated significant research into threshold Schnorr signatures [KG20, CKM21, RRJ+22, BTZ22, BHK+24, Sho23, CKM23, KG24, BLSW24]. The ongoing efforts [BP23] of the National Institute of Standards and Technology (NIST) attest to the growing interest in standardizing threshold signatures and multi-party threshold methods. With a target release date of summer 2024, this involves work on threshold versions of the EdDSA [BDL+11, JL17] and Schnorr signatures.

Among the existing threshold Schnorr schemes, FROST [KG20] stands out as one of the main contributions to this field. FROST is recognized for its efficiency and adaptability in environments where participants are predominantly honest. However, it lacks robustness, particularly in scenarios involving malicious participants or system crashes, and may require a complete restart of the protocol. A direct follow-up work is ROAST [RRJ+22], a protocol designed to address the robustness limitations of FROST. ROAST is a wrapper protocol that incorporates several innovative features. It enhances robustness by providing fault tolerance to handle malicious actors and system crashes. The presence of a semi-trusted coordinator and pipelining features efficiently manage signing sessions and allow for parallel processing. Additionally, its semi-interactive and asynchronous operation reduces the need for strict synchrony assumptions, making the protocol more resilient in real-world conditions.

This thesis aims to implement the ROAST protocol in Thetacrypt [Cry24], thereby providing the first robust threshold signature scheme within the codebase. Thetacrypt is a distributed service for threshold cryptographic schemes developed by the Cryptology and Data Security (CRYPTO) research group at the University of Bern, Switzerland. For the implementation, we assume a setting where the protocol must tolerate up to $f = \frac{n-1}{3}$ malicious participants, with a threshold of $t = n - f$ required for successful signing. This choice reflects a practical consideration of robustness in asynchronous environments, ensuring that ROAST can operate effectively even under adverse conditions. We will evaluate the performance of the ROAST implementation, focusing on its performance in terms of latency, reliability, and the number of signing sessions required to produce a valid signature. We will compare our results with the existing FROST

implementation, showing that although ROAST increases latency, it significantly enhances robustness without imposing substantial overhead. This evaluation underscores the trade-offs between security and performance, providing valuable insights into the practical deployment of robust threshold signature schemes in real-world scenarios.

Chapter 2 provides an overview of the background and notation used throughout this thesis and introduces the core concepts of threshold cryptography. In Chapter 3, we present existing Schnorr threshold signature schemes, including the FROST protocol and its limitations. We introduce the ROAST protocol, highlighting its unique features and improvements over existing schemes. Additionally, we briefly touch on related work that is not extensively covered in this thesis. Chapter 4 details our implementation of the ROAST protocol in Thetacrypt. We discuss the design choices, the integration of the protocol into the existing codebase, and the challenges encountered during the implementation process. Chapter 5 presents the benchmarking results of the ROAST implementation. We evaluate the performance of the protocol and compare the results with the existing FROST protocol. Finally, Chapter 6 concludes the thesis, summarizing the key findings and contributions. We discuss the implications of the ROAST protocol and its potential applications in real-world scenarios, and outline possible future research directions.

2 Background

This chapter outlines the key concepts in threshold cryptography that are essential for understanding the ROAST protocol. It covers the core techniques like secret sharing, distributed key generation, and threshold signatures. Additionally, the chapter introduces the Thetacrypt codebase, which serves as the foundation for implementing and evaluating ROAST.

2.1 Notation

Here, we introduce mathematical notations to be consistently utilized throughout this thesis. Unless explicitly stated otherwise, the ensuing content is applied to all parameters and notations introduced herein.

Let $\mathbb{G}$ represent a cyclic group of prime order q, with the generator denoted as g. Within this context, we assume that the discrete logarithm problem in this group is hard. Let $H : \{0,1\}^* \rightarrow \mathbb{Z}_q^*$ be a cryptographic hash function. We use the notation $s \leftarrow\$ S$ to show that s is chosen uniformly at random from the set S.

Let n denote the total number of participants in the signature scheme, and t represent the threshold of the secret-sharing scheme. Participant i is referred to as P_i, where $1 \leq i \leq n$. Let s_i denote the secret share for P_i. The public key that all participants in the threshold signature scheme share is denoted by Y, with $Y_i = g^{s_i}$ representing the public key share for P_i. Lastly, m denotes the message intended for signing.

2.2 Threshold cryptography

Threshold cryptography [Des94] is intended to provide security to cryptographic keys through two properties: collaboration and fault tolerance. Threshold schemes rely on the homomorphism existing between algebraic groups, which acts like a one-way function safely to distribute cryptographic operations across several parties, thereby allowing shared secrets to be reconstructed without their ever being exposed to any single party.

More formally, cryptographic protocols called (t, n)-threshold schemes, or t-out-of-n, allow a set composed of n parties to share a secret s. This distribution is designed such that the collaboration of any t

participants from the set of n is required to recover s, and subsets of size less than t cannot extract any meaningful information about shared secret s.

2.2.1 Secret sharing

One of the core components of threshold cryptography is secret sharing. In its general version, it allows to transform a secret value into n shares such that $t \leq n$ shares is the minimum number of shares required to rebuild it, while fewer than t shares disclose nothing about the secret. Threshold schemes leverage the core principles of Shamir's secret sharing [Sha79], which constitutes a (t, n)-threshold scheme. Grounded in polynomial interpolation, it leverages the mathematical property that a polynomial of degree $t-1$ requires t points for unique reconstruction. In a t-out-of-n scheme, a secret s, element of a finite field $\mathbb{F}_q$ with generator g is divided among n parties, $\{P_1, ..., P_n\}$, demanding the collaboration of at least t parties for reconstruction.

Algorithmically, a trusted dealer D uniformly selects a random polynomial $f(X) \in \mathbb{F}_q[X]$ as

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{t-1} x^{t-1}$$

of degree $t-1$, where $f(0) = s$. The polynomial is then divided into shares, $s_i = f(i)$, and sent to P_i for $i = 1, ..., n$, so that each participant receives $(i, f(i))$. In order to rebuild s among the indices that form a set $S \subseteq \{1, ..., n\}$, each member in a group of t parties initially broadcasts its share. Upon receiving t shares, the secret s can be reconstructed by computing

$$s = f(0) = \sum_{i \in S} \lambda_0^S s_i$$

where λ_i^S denotes the Lagrange coefficient for i and S defined as

$$\lambda_i^S = \prod_{j \in S \setminus \{i\}} \frac{j}{j-i}$$

Note that Shamir's secret sharing scheme allows arbitrary subsets of parties of size $t < n$ to reconstruct the secret s. This technique forms the basis for secure multi-party computation and plays a significant role in threshold cryptography.

2.2.2 Verifiable secret sharing

Verifiable secret sharing (VSS) extends secret sharing schemes to be fault-tolerant against a possibly faulty dealer and to ensure that the participants receive identical information, which can be verified. Thus, agreement is achieved on the resulting reconstruction of the secret even though the secret is not shared in plaintext. It simultaneously achieves two very important goals, to maintain the consistency of share distribution and to terminate with a guaranteed condition. VSS is a fundamental building block for various distributed cryptographic protocols. VSS introduces fault tolerance so that malicious actors, including the dealer, cannot compromise the integrity of the shared secret.

Feldman's VSS Feldman's VSS [Fel87], an extension of Shamir's secret sharing, incorporates a verification step to establish the consistency of a participant's share with a publicly committed value assumed to be accurately visible to all involved parties. This additional step ensures the integrity of each participant's share by validating it against the publicly committed value. In the event of a validation failure, participants possess the capability to file a complaint against the dealer, taking subsequent actions such as broadcasting the complaint to all other participants.

Pedersen's VSS Pedersen [Ped91a] expands upon Feldman's VSS by providing participants with two secret shares and corresponding public commitments. Verification involves checking share consistency against the committed values. The rest of Pedersen's VSS protocol is identical to Feldman's VSS, notably, complaints can be filed and acted upon in the same manner.

2.2.3 Distributed key generation

Distributed key generation (DKG) is used to prevent single points of failure by setting up protocols without relying on a central authority. The main aim is the collaborative generation of a public key and the sharing of the corresponding secret key between users to achieve high security and resistance to adversaries. In DKG, every participant plays an equal role in the creation of the shared secret. Upon completion of the protocol, all participants possess a shared public key Y. However, each participant retains only a share s_i of the corresponding secret s, ensuring that no subset of participants smaller than the threshold possesses complete knowledge of s.

Pedersen [Ped91b] introduced the initial DKG protocol, leveraging Feldmann's VSS executed concurrently by each participant. This is a protocol of two succinct rounds: the first round involves the broadcasting of polynomial commitments, and the second round distributes secret shares to corresponding parties.

However, Gennaro et al. [GJKR07] have demonstrated vulnerabilities in Pedersen's DKG. Their focus was on biased distributions due to participant complaints upon receipt of shares. To address this, Gennaro et al. proposed a more elaborate three-round approach, introducing a preliminary commitment round to exchange Pedersen's commitments before biased distributions occur. Additional research [GJKR03] showed that, while there is limited bias in Pedersen's DKG, the protocol remains secure in specific applications.

DKG protocols play a deciding role in generating public and private key pairs while preventing manipulation by corrupted parties. These versatile protocols are specifically designed to support a range of public-key cryptosystems, including ones that rely on discrete logarithms. They are capable of adapting to diverse network circumstances. Some have been adapted to synchronous networks [GJKR07], while others have weaker assumptions, like models based on asynchrony with proactive security [CKLS02, KG09].

2.3 Threshold signatures schemes

2.3.1 Digital signatures

In a traditional digital signature scheme, a single entity holds the secret key, and the signing process involves the use of this key to generate a signature. We adhere to the following syntax for digital signature schemes:

Definition 1 (Digital signature scheme)**.** A *digital signature scheme* [Sma16] is a triple of polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ defined as follows:

- $\mathsf{KeyGen}(1^\lambda) \to (Y, s)$ is a key generation algorithm that takes a security parameter 1^λ and produces a pair of keys: the public key Y and secret key s.
- $\mathsf{Sign}(s, m) \to \sigma$ is a signing algorithm that takes as input a secret key s and a message m, and outputs a signature σ.
- $\mathsf{Verify}(Y, m, \sigma) \to 0/1$ is a verification algorithm that takes as input a public key Y, a message m, and a signature σ, and outputs a boolean value. Accept (1) the signature is valid; otherwise, reject (0).

Next, we formally define the completeness and security properties that the digital signature scheme must satisfy.

Completeness A digital signature scheme is *complete* if, for any message m and any key pair (Y, s) generated by $\mathsf{KeyGen}(1^\lambda)$, the signature $\sigma = \mathsf{Sign}(s, m)$ is always accepted by the verification algorithm, in particular $\mathsf{Verify}(Y, m, \sigma) = 1$.

Security Consider an adversary given the public key Y generated by KeyGen and oracle access to the signing algorithm $\mathsf{Sign}(s, m)$ for adaptively chosen messages. A scheme $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is said to be *existentially unforgeable against chosen message attacks* (EU-CMA), as defined by Goldwasser et al. [GMR88], if the adversary cannot produce a valid signature for any new message m that was not queried to the oracle, except with negligible probability in the security parameter 1^λ.

2.3.2 Threshold signatures

In contrast, *threshold signatures* [Sma16] take a joint approach where no single participant is given the complete secret key. Such schemes are characterized by the parameters (t, n), where t refers to the minimum number of participants that must collaborate, and n is the total number of participants in the scheme.

In such schemes, the secret key is shared among the participants, while a common public key represents the entire group. In the concept of Shamir's secret sharing, a secret key is divided into a set of shares using a polynomial-based approach. Most importantly, t participants $P_1, ..., P_t$ can collaboratively produce a signature over a message using just their respective secret shares $s_1, ..., s_t$, without ever reconstructing the original key. This requires secure methods for generating and distributing the secret key shares $s_1, ..., s_n$ as no participant or group of fewer than t should learn the secret key. DKG protocols are one approach to achieve this while in some cases a trusted dealer may be acceptable.

The threshold signature schemes are inherently different from traditional signing processes; here, a joint effort is required instead of an individual entity applying the secret key to produce a signature. More precisely, participants jointly contribute their shares towards the computation of the signature in a secure and distributed manner. The verification algorithm remains exactly the same, so the resulting signature is verified as if it was generated by only one entity. Following the definition of Gennaro et al. [GRJK07, GG18], a *threshold signature scheme* consists of two main components:

- In the key generation process, denoted as $\mathsf{Tresh\text{-}KeyGen}$, a group of participants collaboratively generates a common public key Y and a set of secret key shares $\{s_i\}_{i=1}^n$. Each participant P_i receives their own secret share s_i, which represents a portion of the overall secret key s. The key shares are distributed in such a way that a minimum of t participants is required to reconstruct the secret or to perform a signing operation, forming a t-out-of-n threshold scheme.

- The signing process, denoted as $\mathsf{Tresh\text{-}Sign}$, is also distributed. It allows the participants, using only their secret shares s_i, to collaboratively produce a single aggregated signature σ on a given message m. This aggregated signature can then be verified using the standard verification algorithm $\mathsf{Verify}(Y, m, \sigma)$, as defined in the digital signature scheme, ensuring that the signature is valid without revealing the individual secret shares or requiring all participants to be present.

Completeness A threshold signature scheme is *complete* if, for any message m and any set of t participants' secret shares $\{s_i\}_{i=1}^t$ generated by $\mathsf{Tresh\text{-}KeyGen}(1^\lambda)$, the aggregated signature $\sigma = \mathsf{Tresh\text{-}Sign}(\{s_i\}_{i=1}^t, m)$ is always accepted by the verification algorithm $\mathsf{Verify}(Y, m, \sigma) = 1$.

Security A threshold signature scheme is said to be *unforgeable* if no adversary, who corrupts at most $t-1$ participants, can produce a valid signature on any new message m that has not been previously signed by the protocol. This holds even if the adversary has full knowledge of the public key Y, the shares of the corrupted participants, and the views of the protocols Tresh-KeyGen and Tresh-Sign on any set of adaptively chosen messages $m_1, \ldots, m_k$. The adversary should only be able to produce a valid signature on a previously unsigned message with negligible probability in the security 1^λ,

Threshold implementations have been developed for widely recognized signature schemes, such as RSA [Sho00], BLS [BLS04], ECDSA [GG18, CGG+20, GS22], and Schnorr [KG20, RRJ+22, BHK+24, CGRS23, BLSW24, KG24].

2.3.3 Interactive vs. non-interactive threshold signing

Threshold signature schemes can be categorized based on the level of interaction required among participants during the signing process.

Non-interactive In a *non-interactive* threshold signature scheme, each participant independently computes and broadcasts a partial signature in a single round. After receiving enough partial shares, each part can assemble the signature locally. This deterministic process eliminates the need for multiple communication rounds among participants. Examples of a non-interactive threshold signature scheme are threshold BLS [BLS04] and threshold RSA [Sho00].

Semi-interactive *Semi-interactive*, sometimes called *partially-interactive* [BTZ22], threshold signature schemes contain distinct rounds: preprocessing and signing [CGRS23]. The preprocessing round involves participants performing operations without knowledge of the message or the involved parties' subset. Subsequently, in the message-dependent signing round, each participant shares their local output with the other participants. That way interaction is allowed, with the preprocessing round ensuring message-independent operations. While direct communication between parties is absent during signature generation, the preprocessing round introduces a limited form of interaction. An example of a semi-interactive threshold signature scheme is FROST [KG20].

Fully interactive A *fully interactive* threshold signature scheme involves multiple rounds of communication and interaction among parties to generate a threshold signature. Unlike non-interactive or semi-interactive schemes where the signature can be generated with limited or no real-time communication, in a fully interactive scheme, active communication and coordination among the parties are necessary throughout the whole process of signature generation. This approach is less common due to the increased communication overhead and complexity. Examples of fully interactive threshold signature schemes include multiple variants of ECDSA [GG18, CGG+20, GS22].

2.3.4 Some security properties

In this section, we informally outline some of the security properties that threshold signature schemes aim to achieve.

Robustness *Robustness* [Sho00] refers to the protocol's ability to guarantee successful signature generation despite disruptions caused by malicious actors. A protocol is considered robust if it allows t honest signers to reliably produce a valid signature even in the presence of up to f malicious signers who attempt to sabotage the process, provided that $f \leq n - t$.

Unforgeability *Unforgeability* guarantees that $t-1$ malicious signers cannot produce a valid signature. This property addresses a scenario where an adversary lacks knowledge of the secret key s but can acquire signatures for a finite number of chosen messages from a signing oracle [GMR88]. A threshold signature scheme achieves EU-CMA when no adversary can create a new valid signature for a message that has not been signed before.

Identifiable aborts *Identifiable aborts* [CGG+20] refer to a security property that ensures if a signing session fails to complete due to disruptions or other issues, the protocol can reliably identify the specific signers who caused the abort. This property is crucial for accountability and fault tolerance, as it allows the protocol to identify and exclude malicious actors from future sessions.

2.4 Thetacrypt codebase

Thetacrypt [ABC+23] is a Rust[1] codebase developed and maintained by the CRYPTO research group at the University of Bern, Switzerland. Its primary focus is on providing threshold cryptography as a service by implementing various threshold primitives such as ciphers, signatures, distributed key generation, coin schemes, and randomness beacons. The repository is open-source and available on GitHub [Cry24].

Thetacrypt operates through three essential layers depicted in Figure 2.1: the Service Layer, the Core Layer, and the Network Layer. These layers collectively contribute to the functionality and robustness of the system. Thetacrypt employs a modular architecture to enhance the accessibility and deployability of threshold cryptography.

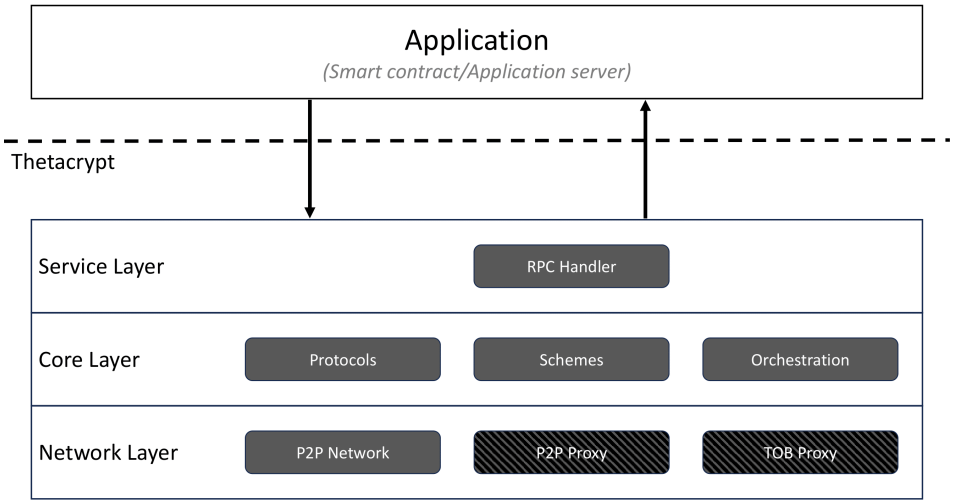


Figure 2.1: Abstracted architecture of Thetacrypt

Service layer At the highest level, the Service layer should handle API requests and present management code to deal with the service itself. This is what a user or application will use as an entry point when trying to interact with Thetacrypt. The Service Layer includes an RPC handler that manages API requests and other user interactions.

[1] `https://www.rust-lang.org/`

Core layer Sitting at the bottom, next to the Service layer, the Core layer forms the core of Thetacrypt. It coordinates the concurrent execution of protocols and schedules them to ultimately control cryptographic operations. This is where the cryptographic primitives are implemented to effect threshold schemes for the secure execution of cryptographic protocols. It has been designed to tackle complex, multi-round schemes in a distributed setting in a modular and flexible way for deployment and integration of distributed cryptosystems. Most of the work done during this thesis was at this core layer level, based on the modularity of the application.

Network layer The Network layer is at the base of Thetacrypt and allows nodes to communicate through a peer-to-peer (P2P) network or gossip protocol. In addition to the inter-node coordination required for the cryptographic protocols, this layer also provides two proxy modules to the upper stage. These modules facilitate integration with existing replicated services and their corresponding network layers, such as blockchains. This allows the instances to exchange information and coordinate with each other, thereby enabling the execution of distributed cryptographic protocols.

Cryptography Internally, Thetacrypt supports several elliptic curves, such as `Bls12381` [BLS02], `Bn254` [BN05], `Ed25519` [BDL+12] and various `RSA` variants, accommodating various cryptographic schemes. It handles concurrent instances of threshold protocols, employs a keychain for different schemes and groups, and provides API endpoints for client interactions. Thetacrypt provides a variety of cryptographic schemes, such as ciphers [SG02, BZ03], signatures [Sho00, BLS04, KG20], and a coin scheme [CKS05]. The sole interactive threshold scheme before this thesis was FROST [KG20]. As part of this thesis, we implement the second interactive scheme, ROAST [RRJ+22]. The implementation will be discussed in Chapter 4 and benchmarked in Chapter 5.

3
Schnorr threshold signatures

In this chapter, we provide an overview of existing Schnorr threshold signature schemes, introducing the FROST protocol and its limitations. We then focus on the novel ROAST protocol, highlighting its unique features and improvements over existing schemes.

3.1 Schnorr signatures

A Schnorr signature [Sch89] is a cryptographic digital signature scheme that provides a secure method for verifying the authenticity of a message or transaction. It is based on the discrete logarithm problem in a finite cyclic group and the Fiat-Shamir transform [FS86] enables binding the signature σ to the message m in a non-interactive way. Building on the definitions provided by Schnorr [Sch89] and Smart [Sma16], we define the concept more formally as follows:

Definition 2 (Schnorr signature scheme)**.** The *Schnorr signature scheme* is a triple of polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ defined as follows:

- $\mathsf{KeyGen}(1^\lambda) \to (Y, s)$: Input a security parameter 1^λ. Randomly select a secret key $s \leftarrow\!\$\ \mathbb{Z}_q^*$ and compute the corresponding public key as $Y = g^s$. Output the key pair (Y, s).
- $\mathsf{Sign}(s, m) \to \sigma$: Input the secret key s and a message m. Randomly select a nonce $k \leftarrow\!\$\ \mathbb{Z}_q^*$ and compute the commitment $R = g^k$. Compute the challenge $c = H(R, Y, m)$ and the response $z = k + s \cdot c$. Output the signature $\sigma = (R, z)$.
- $\mathsf{Verify}(Y, m, \sigma) \to 0/1$: Input the public key Y, the message m, and the corresponding signature $\sigma = (R, z)$. Compute the challenge $c = H(R, Y, m)$ and the commitment $R' = g^z \cdot Y^{-c}$. Accept (1) the signature if $R = R'$; otherwise, reject (0).

We refer to the original paper [Sch89] for the security proofs of Schnorr signatures. The scheme is provably secure under the discrete logarithm assumption in the random oracle model [PS00].

3.2 FROST

FROST [KG20] (Flexible Round-Optimized Schnorr Threshold signatures) is a pioneering cryptographic protocol that sparked significant interest in the area of Schnorr threshold signatures. As the first major work in this domain, it serves as the primary point of reference and comparison for subsequent developments in the field. The protocol aims at minimizing network overhead while maintaining security and flexibility. FROST is semi-interactive, featuring a message-independent pre-processing round followed by a message-dependent signing round. This allows it to be utilized as either a two-round protocol or optimized to a single-round protocol with a preprocessing step. Notably, FROST doesn't place any limitations on how the threshold parameter t relates to the overall number of participants n, meaning it guarantees unforgeability even in the case of a dishonest majority ($t - 1 \geq n/2$).

Efficiency vs. robustness FROST is optimized for *optimistic* settings, in which the participants presumably are honest. In comparison to robust threshold signatures, it merely aborts the protocol and requires every participant to proceed with the entire protocol afresh without the malicious party if a party misbehaves. This in turn directly translates to far less communication overhead in the typical successful case. Furthermore, since FROST aborts upon detecting misbehavior, it inherently possesses the property of identifiable aborts.

Signature Aggregator role FROST utilizes an optional semi-trusted entity called the Signature Aggregator ($\mathcal{SA}$) to streamline communication among participants. While the $\mathcal{SA}$ is not essential, it can be any participant or even an external party, responsible for reporting misbehavior and publishing the final signature. The $\mathcal{SA}$ is trusted to correctly report any misbehavior it detects among the participants during the signature generation process and to aggregate and publish the final signature without alteration. Although semi-trusted, the $\mathcal{SA}$ cannot forge signatures, compromise the private key, or learn any sensitive information that would enable it to do so. The protocol remains secure against chosen-message attacks even if the $\mathcal{SA}$ is malicious. In the setting without the presence of an $\mathcal{SA}$, participants broadcast their messages directly to each other through a shared broadcast channel. In the following, we present the process utilizing an $\mathcal{SA}$.

Commitment server The commitment server, a centralized entity, is responsible for storing and distributing these commitment values to ensure all participants have consistent access during the signing process.

3.2.1 FROST rounds

In this section, we provide an overview of the steps involved in the FROST protocol, including the key generation, preprocessing, and signing rounds.

Key generation FROST can use a traditional trusted dealer or employ DKG to generate and share keys among participants. In FROST [KG20], the protocol uses the latter and builds upon the Pedersen DKG protocol [Ped91b] with a key modification to address rogue-key attacks. Participants are required to present zero-knowledge proofs about their secret value commitments, ensuring bad actors cannot claim incorrect shares. This would confirm that all generated keys are valid and necessary for threshold signature security in the following steps. Regardless of the method of key generation, at the completion of this step, each participant has:

- A secret share: a unique portion of the private key known only to them.

- A public key share: this share enables other participants to verify the signature produced by the participant.
- A group verifying key: a public key corresponding to the private key divided into shares, that can be used to verify the final FROST signature.

Preprocessing The preprocessing round in FROST serves several purposes, like generating and sharing values of commitments to be used in the following signing round. More than that, it allows for an efficient way of handling a number of signing sessions by participants who pre-compute and store the commitment values in advance. The batching approach thus minimizes the computational and communication overheads of each signing operation. Further, such a preprocessing round provides the foundation to guarantee integrity and authenticity in commitment values. Figure 3.1 illustrates the steps of the preprocessing round.

1. Each participant P_i generates a pair of nonces $(d_i, e_i) \in \mathbb{Z}_q^* \times \mathbb{Z}_q^*$.
2. Next, each participant computes the corresponding public commitments $D_i = g^{d_i}$ and $E_i = g^{e_i}$. This can be done in batches to prepare for multiple signing sessions, hence the second index j.
3. The commitments are then sent to the commitment server, where they are stored. Note that in the two-round variant of FROST, this step is performed immediately before signing with only a single commitment.

FROST Preprocess

	Participant i		**Commitment Server**
1 :	$((d_{ij}, e_{ij}), ...) \leftarrow\$ \mathbb{Z}_q^* \times \mathbb{Z}_q^*$		
2 :	$(D_{ij}, E_{ij}) = (g^{d_{ij}}, g^{e_{ij}})$		
3 :	Store $((d_{ij}, D_{ij}), (e_{ij}, E_{ij}), ...)$		
4 :		$((D_{ij}, E_{ij}), ...) \longrightarrow$	
5 :			Store $((D_{ij}, E_{ij}), ...)$

Figure 3.1: FROST preprocessing round [Kom20]

Signing The signing phase in FROST involves a series of steps where each participant computes their signature share, which is then aggregated by the signature aggregator $\mathcal{SA}$ to produce the final signature. This phase follows the preprocessing round and requires the commitments of every participant involved in the signing operation, along with their identifiers. The steps involved in the signing round are explained in the following and algorithmically depicted in Figure 3.2.

1. To start the signing round we require the commitments of each participant taking part in the signing operation, along with their identifiers. Those are stored in the value B for parties $P_1, ..., P_t$.
2. Each signer receives the message to be signed m and the value B.
3. Initially, each signer computes the set of binding factors ρ_ℓ, which binds each participant's signature to the identifier ℓ, the specific message m and commitments B.

4. Next, the joint group commitment R is computed by multiplying each signer's commitments D_ℓ and E_ℓ in combination with the binding factor ρ_ℓ.

5. The challenge c is computed by hashing the joint commitment R, the public key Y, and the message m. These three steps are performed by each signer and they obtain the same values for ρ_ℓ, R, and c.

6. The response z_i is then computed by each participant P_i using the two nonces d_i and e_i in combination with the binding factor of the participant ρ_i and the challenge c. The Lagrange coefficient λ_i enables us to use additive secret sharing with the secret s_i. Note that the computation of z_i on line 6 cannot be inverted by any adversary that does not know (d_i, e_i), which protects against removing the binding factor and ties the signature share to the message m.

7. Once each signer has produced their signature share z_i and sent it to the signature aggregator $\mathcal{SA}$, the final signature σ is computed by summing up all responses z_i and published together with the joint group commitment R.

FROST Sign

	Signer i		**Signature Aggregator $\mathcal{SA}$**
1 :			$B = ((1, D_1, E_1), ..., (t, D_t, E_t))$
2 :		$\xleftarrow{(m, B)}$	
3 :	$\rho_\ell = H_1(\ell, m, B), \ell \in S$		
4 :	$R = \prod_{\ell \in S} D_\ell \cdot (E_\ell)^{\rho_\ell}$		
5 :	$c = H_2(R, Y, m)$		
6 :	$z_i = d_i + (e_i \cdot \rho_i) + \lambda_i \cdot s_i \cdot c$		
7 :		$\xrightarrow{z_i}$	
8 :			Publish $\sigma = (R, z = \sum_{i \in S} z_i)$

Figure 3.2: FROST signing round [Kom20]

It is noteworthy that FROST produces a regular Schnorr signature, meaning the format and verification procedure are identical to single-party Schnorr signatures. This allows for straightforward integration into existing systems and applications that support Schnorr signatures.

3.2.2 Practical considerations

There are many FROST implementations provided by third parties, differing widely in terms of complexity and functionality. In the following, we will mention some of the most prominent ones. Zcash Foundation [Fou24] provides a Rust implementation of the two-round variant of FROST supporting many ciphersuites. There, the implementation allows the user to choose between a trusted dealer and DGK for key generation and if a $\mathcal{SA}$ is used or not. The Applied Research Team of the Bank of Italy published an implementation of FROST written in C as an experimental module [ART23] of the `secp256k1` [sec24] Bitcoin library. This implementation follows the design choices of the original paper closely with two rounds and a $\mathcal{SA}$. It is designed to be used in the context of Bitcoin and is used to provide threshold

signatures for the Bitcoin network. Taurus group [Gro21] used Go to implement FROST in a two-round variant with DGK following the original paper, albeit without an $\mathcal{SA}$. Instead, the signers broadcast their signature shares to each other and locally aggregate the full signature. Similarly, the CRYPTO research group at the University of Bern, Switzerland, implemented FROST as part of the Thetacrypt project [Cry24], which provides a Rust implementation of FROST as part of a larger framework for threshold cryptography. This two-round implementation does not include a $\mathcal{SA}$ and uses a trusted dealer for key generation. A total order broadcast channel is used to ensure that all participants receive the same messages in the same order, which is crucial for the correctness of the protocol.

These implementations have two things in common: First, they all implement the two-round variant of FROST due to the complexity of the single-round pre-processing step and the assumption of a trusted server to store the commitments. Second, they leave the handling of misbehaving participants and the eventual restart of the protocol to the application layer. Their design is such that they are meant to detect misbehavior and bring the protocol to a halt, thus giving the property of identifiable aborts and requiring the application to decide how to proceed. This makes sense, as the protocol itself does not specify how to handle misbehaving participants, so it will be left to the application layer based on the requirements of the application.

3.2.3 Limitations

Practical implementations highlight the protocol's limitations. A single point of failure can be introduced through the use of an $\mathcal{SA}$, as it is in charge of aggregating the signature shares and publishing the final signature. If an attacker compromises an $\mathcal{SA}$, the whole protocol is in danger. This can be avoided by removing the $\mathcal{SA}$, and having participants broadcast their signature shares to one another; this introduces extra state and communication complexity. Also, as mentioned above, while FROST is very efficient and flexible it is not particularly robust; for example, misbehaving participants can easily force it to abort. In contrast to robust threshold signature schemes, which operate securely even when participants misbehave, FROST takes a more optimistic approach. In case of misbehavior, FROST aborts the protocol and needs to be re-run with the full protocol without the malicious party, which adds communication overhead and possible delays. This lack of robustness in FROST motivates the development of ROAST, a new way to threshold Schnorr signatures designed to address these limitations.

3.3 ROAST

ROAST [RRJ$^+$22], an acronym for *RObust ASynchronous Threshold signatures*, introduces a novel approach to threshold Schnorr signatures, leveraging a semi-interactive threshold signature scheme as a building block. Ruffing et al. designed ROAST to be robust, ensuring that a malicious participant cannot halt the protocol. The protocol is asynchronous, meaning it does not rely on any synchrony assumptions, and is the first of its kind to provide such security guarantees. The protocol is asynchronous, meaning it does not rely on any synchrony assumptions, and it is the first of its kind to provide such security guarantees.

3.3.1 Key differences to FROST

ROAST is a wrapper protocol for an underlying semi-interactive threshold signature scheme Σ that provides identifiable aborts and unforgeability, such as FROST. In the following, we describe ROAST with Σ = FROST, as in the original paper. In other words, unlike FROST, ROAST is not a digital signature scheme but rather a protocol that enhances the security of an underlying scheme. The individual signing sessions within ROAST are executed using the underlying scheme Σ. While ROAST leverages the underlying cryptographic foundation of FROST, it introduces several central enhancements:

Robustness ROAST detects malicious parties submitting invalid shares or conflicting messages to the coordinator and eliminates them from the process. Upon detecting malicious participants, ROAST simply ignores their messages and continues execution—even in contrast to FROST. That is, multiple signing attempts are made to ensure the generation of a robust signature in the presence of malicious signers.

Coordinator ROAST introduces a coordinator, a semi-trusted entity responsible for initiating signing sessions and managing the signing process. The coordinator replaces the signing aggregator $\mathcal{SA}$ from FROST and is trusted to ensure the protocol's robustness, meaning it ensures the signing process continues without halting due to disruptive or malicious behavior. However, it does not have the ability to generate or alter valid signatures. This allows for an optimistic selection of the coordinator, as it can be replaced if found to be unreliable or malicious.

Pipelining ROAST, unlike FROST, allows multiple attempts to generate a signature by initiating multiple concurrent (FROST) singing sessions within a single run of the protocol. This is achieved by pipelining (FROST) signing sessions, where the coordinator initiates subsequent signing sessions even while waiting for responses from the current session. Signers prepare for future sessions by providing the coordinator with fresh pre-signature shares in addition to their signature shares. As soon as the coordinator receives enough signature shares for a new session, it initiates the next session while waiting for the remaining shares of the current session. This improves efficiency and reduces the time required to initiate additional (FROST) signing sessions.

Asynchrony The coordinator is connected to the signers through authenticated and reliable P2P channels. ROAST eliminates the need for timeouts by assuming the eventual delivery of messages among honest parties, enabling progress even in asynchronous networks. In contrast, FROST requires synchronous communication.

In addition to those enhancements, there are other substantial differences. Notable changes are as follows:

Key generation Both FROST and ROAST allow for flexibility in the choice of key generation methods. While FROST uses a customized version of the Pedersen DKG protocol [Ped91b] in its paper, any other DKG protocol that satisfies the necessary security properties may also be used. Similarly, ROAST does not specify a particular DKG protocol and can accommodate any DKG protocol or trusted dealer, provided that the resulting keys meet the basic correctness condition, allowing for the aggregation of public keys via Shamir secret sharing interpolation in the exponent.

Minor changes to FROST as subprotocol ROAST introduced a few minor changes to the FROST protocol used as Σ compared to Komlo and Goldberg [KG20]. The changes are of a practical nature and do not affect the security of the protocol. They build upon improvements presented in FROST2 [CKM21] and subsequent work [BTZ22] and include the use of a different hash function for the challenge computation and the ability to aggregate presignature shares before broadcasting them to signers. The authors name their variant FROST3, but we will refer to it as FROST for simplicity. Consider the functions of FROST3 in Figure A.1 in the appendix.

3.3.2 Core functionalities

First, we briefly introduce some supplementary definitions. A *responsive signer* is a signer who has actively participated in recent signing rounds and provided valid responses. A *pending signer* is a signer

who has yet to submit a valid reply in the ongoing session. A *malicious signer* either sends an invalid response or a message inconsistent with the protocol, see also Section 3.3.3.

ROAST operates through the coordinated efforts of a central coordinator $\mathcal{C}$ and individual signers $S_i, ..., S_n$: $\mathcal{C}$ keeps track of the signers, their respective state and the signing sessions. For this purpose, $\mathcal{C}$ uses the set R for responsive signers and set M for malicious ones. Each signer is limited to be pending in at most one signing session at a time. Pending signers are excluded from R. Honest signers are guaranteed eventual inclusion in set R, while unreliable or malicious signers are effectively blacklisted and added to the set M, ensuring their exclusion from future sessions. In contrast, reliable signers are consistently incorporated into future signing sessions. As soon as $|R|= t$, $\mathcal{C}$ initiates a new signing session. For the full pseudocode of ROAST, we refer to Figure A.2 in Appendix A.

In the worst-case scenario, ROAST anticipates $f + 1$ signing sessions, where f of those $f + 1$ sessions involve exactly one unreliable party. Importantly, among the $f + 1$ sessions, at least one session will contain no malicious signers, meaning at least one session will eventually successfully terminate.

Coordinator

- Oversees responsive and malicious signers:
  - Maintains a list R of responsive signers who have actively participated in recent rounds and provided valid responses. Initially, R is empty. Once a signer P_i sends their initial pre-signature share ρ_i, they are added to R.
  - Maintains a separate list M of signers identified as uncooperative or malicious. Messages from these signers are ignored. Initially, M is empty.
- Coordinates signing sessions:
  - Initiates new sessions once at least t responsive signers are in R, sending them:
    - The aggregated pre-signature ρ of this specific signing session.
    - A list of participating signers R, $|R|= t$.
  - Receives and tracks:
    - Signature shares σ_i from signers for ongoing and future sessions, representing individual contributions to the final signature.
    - Fresh pre-signature shares ρ'_i from signers, used to prepare for potential future signing sessions and improve efficiency.
  - Validates the received shares σ_i. Invalid shares are ignored, and the corresponding signers are marked as malicious and thus added to M. Valid shares are aggregated to generate the final signature σ.
  - Manages multiple signing sessions concurrently by using the pipelining mechanism.

Signer

- Generates a pre-signature share ρ_i in the initial round and sends it to the coordinator.
- Upon receiving a session pre-signature ρ and a list of participating signers R for a specific session from $\mathcal{C}$, generates signature share σ_i for that session.
- Send σ_i and a new pre-signature share ρ'_i back to $\mathcal{C}$. The latter is used to enable pipelining of signing sessions.

3.3.3 Detecting malicious signers

The mechanism to detect a malicious signer is twofold: First, ROAST uses the concept of identifiable aborts in FROST. This means that a malicious signer can be identified by the coordinator if they send an invalid

signature share. Second, a participant is also marked as malicious if they send a message to the coordinator that is not consistent with the protocol. More specifically, a participant is marked as "responsive" if they already sent a valid message to the coordinator. Responsive participants are not expected to send any further messages to the coordinator until the coordinator sends a new message to them, which removes them from the set of responsive participants. Once a participant is marked as malicious, the coordinator will no longer consider any messages from that participant.

3.3.4 Security properties and complexity

Unforgeability ROAST inherits FROST's support for arbitrary choices of t and n, allowing it to guarantee unforgeability against a dishonest majority ($t - 1 \geq n/2$). It is important to note that due to ROAST's concurrent execution of multiple FROST signing sessions, ROAST achieves a weaker notion of unforgeability, see Section 4.1 in the original paper. This means that an adversary might be able to generate multiple valid signatures for the same message. For instance, a malicious coordinator could collect the final signatures from multiple completed signing sessions of Σ. This is however often sufficient for applications where the final goal is precisely to prevent attackers from producing signatures.

Robustness Here we discuss the formal proof of ROAST's robustness presented in the original paper. Note that ROAST provides a generalized notion of robustness compared to the standard definition: Robustness ensures that a signing session will produce a valid signature as long as there are at least t honest signers, even if others are trying to sabotage the process. This holds up to $t - 1$ malicious signers, provided there are enough honest participants (i.e., $f \leq n - t$). However, when dealing with a dishonest majority (where $t - 1 \geq n/2$), the protocol may still create a valid signature with t honest signers, but it can't guarantee the signing process will always finish (liveness). This separates the threshold for unforgeability from the threshold for liveness, as ROAST only guarantees liveness if $f < \frac{n}{3}$.

With that in mind, Theorem 4.3 [RRJ+22] establishes that ROAST(Σ) is robust, with the coordinator guaranteed to terminate after initiating a maximum of $n - t + 1$ signing sessions of the underlying scheme Σ. The proof relies on several key concepts in addition to responsive and pending signers: a *disruptive signer* is a signer who deliberately avoids sending valid messages in any session. A *valid reply* is a message sent by a signer within a session that passes validation checks. Finally, a session *terminates* when all participants sent valid replies to the coordinator.

Honest signers, by design, are never disruptive to the protocols and consistently send valid replies to the coordinator $\mathcal{C}$. They are never falsely identified as malicious due to their compliant behavior. The proof assumes the opposite scenario: no session ever terminates. Subsequently, it highlights a contradiction by emphasizing that honest signers will ultimately provide legitimate responses, which will cause them to be included in the responsive list R. When the list hits the threshold t, a new session starts since there are at least t honest signers. This means that eventually, a session containing only non-disruptive pending signers is bound to occur, ensuring its termination. This contradicts the initial assumption, proving that at least one session successfully finishes.

Furthermore, the proof shows that the protocol initiates at most $n - t + 1$ sessions. If more sessions are initiated without termination, the number of pending signers in the responsive list falls below the threshold t, preventing further session initiation. This reinforces the robustness of ROAST by confirming that it ends after a limited number of attempts.

Complexity The asymptotic complexity analysis focuses on three aspects: asynchronous rounds, communication, and computation.

Asynchronous Rounds ROAST adopts a standard definition of asynchronous rounds from Canetti et al. [CR93]. After an initial preprocessing step, each signing session requires two additional asynchronous

rounds, with the protocol initiating at most $n - t + 1$ signing sessions to deliver a valid signature. Consequently, the coordinator produces a signature after at most

$$1 + 2(n - t + 1) = 2(n - t) + 3$$

asynchronous rounds.

Communication The underlying signature scheme Σ controls the communication complexity of ROAST, namely the sizes of presignature shares, presignatures, and signature shares in Σ. Assuming these sizes are $\mathcal{O}(\lambda)$, where λ represents security parameters. As ROAST initiates at most $n - t + 1$ signing sessions with t signers each, the total bits transmitted during a ROAST run is bounded by

$$t(n - t + 1)(n + \mathcal{O}(\lambda)) = \mathcal{O}(tn^2 + tn\lambda).$$

Computation Each signer engages in one PreRound and one SignRound call per session, while the coordinator performs one PreAgg call and up to t calls to ShareVal per session. The overall computational effort for a ROAST run is at most

$$(n - t + 1)(\tau_{\text{PreRound}} + \tau_{\text{SignRound}}) + \tau_{\text{PreRound}}$$

for signers and

$$(n - t + 1)(\tau_{\text{PreAgg}} + t \cdot \tau_{\text{ShareVal}}) + \tau_{\text{SignAgg}}$$

for the coordinator, excluding state maintenance time.

3.3.5 Considerations for ROAST in a decentralized deployment

In the centralized variant of ROAST, there is no requirement for a secure broadcast channel. It involves a semi-trusted coordinator who communicates with signers through P2P channels. None of the signers communicate with one another. Specifically, while the coordinator drives the process of signing and sends out messages, it plays no role in the security (unforgeability) of the final signature. This allows for optimistic selection: if the elected coordinator ends up being unreliable, it can be replaced without undermining the protocol.

This might not be very desirable in many cases. Thus, ROAST provides a decentralized alternative. In that version, instead of one coordinator, all n signers execute $n - t + 1$ concurrent instances of the protocol. In each instance, a different signer acts as the coordinator while also participating as a regular signer. This redundancy ensures that at least one coordinator is honest so long as t signers are honest. Since the honest signers are talking to each other directly, message delivery is now guaranteed. Of course, this comes at a cost: execution time and communication complexity both increase. Furthermore, the protocol may produce multiple valid signatures if the multiple instances succeed independently.

3.4 Related work

While this thesis does not explore them in detail, there are other current and interesting schemes worth mentioning for their contributions to the field.

3.4.1 SPRINT

The SPRINT [BHK+24] protocol, a permuted acronym for "Robust Threshold Schnorr with Super-INvertible Packing", addresses the challenge of high-throughput Schnorr signature generation in large-scale, asynchronous settings, particularly designed for public blockchains. SPRINT introduces a novel approach to non-interactively generate Schnorr-type signatures with packed secret sharing [FY92] using a super-invertible matrix [HN06], enabling SIMD (Single Instruction, Multiple Data) computations for efficient signing. SPRINT combines this with an early-termination agreement, so that the protocol might terminate early. Those techniques come with a trade-off of reduced resilience to a specified threshold. The protocol distinguishes itself by achieving superior throughput compared to FROST, and inherently to ROAST, while providing robustness. It also is adaptable in dynamic settings through the integration of batch randomness extraction techniques, which allow efficient presignature generation. Additionally, SPRINT works with diverse committees for each run and introduces a randomness beacon for sub-sampling committees, thereby enhancing scalability.

However, as discussed by Shoup [Sho23] (see Section 3.4.2), SPRINT's security is confined to specific operation modes. The security theorem in SPRINT applies only when a restricted number of presignatures is generated before the requests to sign. The criticism by Shoup underlines that due to the too restrictive security context, the purpose of presignatures essentially fails in SPRINT. The security analysis of the SPRINT paper focuses its consideration on a very restrictive setting known as a chosen message attack where a fixed-size batch of presignatures signs a corresponding batch of messages. Namely, the restriction is seen when several batches of presignatures are generated in advance; under this condition, the protocol is vulnerable to subexponential attacks and thus becomes ineffective.

3.4.2 The many faces of Schnorr

Shoup [Sho23] enhances SPRINT to be concurrently secure in a black-box way. Shoup introduces a novel approach for robust threshold signing protocols for Schnorr signatures that combines elements of FROST and SPRINT. This approach offers security and robustness without relying on synchrony assumptions, ensures security even with an unlimited number of presignatures, enables concurrent signing requests with minimal latency, achieves high throughput, and demonstrates optimal resilience against corrupt parties (up to $f < \frac{n}{3}$).

The contributions extend to a unified framework for protocol analysis, abstracting core ideas for versatile implementation. The paper provides security proofs in the generic group model and random oracle model for various enhanced attack modes associated with both SPRINT and FROST. Shoup shows that their proposed threshold Schnorr signing protocol can be reduced to the security proofs established for SPRINT and FROST. Furthermore, the paper gives an additive key-derivation model to deal with threshold key maintenance challenges. This way, the signing committee will maintain only a single master key and derive subkeys as needed, allowing for easier distributed computations.

Although the paper offers a detailed analysis of the individual components, it lacks a clear and comprehensive description of the complete protocol and its steps. This makes the protocol difficult to understand and apply in practice within the scope of this master thesis.

3.4.3 HARTS

The HARTS [BLSW24] protocol, short for "High-threshold, Adaptively Robust Threshold Schnorr," was recently introduced to address the limitations of existing threshold Schnorr signature schemes like FROST, ROAST, SPRINT, and the enhancements by Shoup [Sho23]. Detailed by Bacho et al., HARTS aims to provide adaptive security, high efficiency, and robustness in asynchronous environments.

HARTS uses a novel packed asynchronous distributed key generation (ADKG) protocol that leverages a high-threshold asynchronous verifiable secret sharing (AVSS) scheme. This innovation allows for the

secure and efficient distribution of keys among n parties, maintaining robustness with a communication cost of $\mathcal{O}(\lambda n^2 \log n)$ bits with λ being the signature size and round complexity of $\mathcal{O}(1)$. The use of super-invertible matrices helps with efficient multi-nonce generation, which reduces communication overhead significantly compared to traditional methods.

The paper highlights several improvements of HARTS over existing protocols. Unlike FROST and ROAST, which do not support adaptive security, HARTS ensures security even with dynamic adversaries. Compared to ROAST, which suffers from high communication costs and multiple rounds for signature generation, HARTS achieves near-optimal communication efficiency and minimal round complexity. While SPRINT introduces efficient signing via packed secret sharing, its security is limited to specific operation modes, and it faces challenges with multiple presignature batches.

These issues are overcome by HARTS, which provides a unified security model that handles dynamic adversaries and multiple signing requests effectively. Compared to the improvements of Shoup [Sho23], which provide a unified framework of robust threshold signing with concurrent security and high throughput, HARTS provides strong robustness and throughput advantages with a focus on adaptive security key generation and management. While HARTS offers a promising development in Schnorr threshold schemes, its real-world deployment and testing have yet to be seen.

3.4.4 Arctic

Most recently, the Arctic protocol [KG24], introduces a deterministic, stateless two-round threshold Schnorr signature scheme. The main innovation is that signers do not need to maintain state between signing rounds. The authors formally define a new building block called *Verifiable Pseudorandom Secret Sharing* (VPSS) and introduce the efficient VPSS *Shine* for nonce generation and verification.

Arctic's key generation process provides participants with secret signing shares and public signing shares. These shares are computed by the Shine key generation algorithm. The resulting keys will be used in subsequent signing sessions for nonce and commitment generation. The signing procedure in Arctic consists of two rounds. In the first round, participants deterministically generate nonces and commitments. This is the main difference to related schemes like FROST and its follow-up works, where participants randomly generate their nonces. These nonces are re-derived in a second round, after which the participants verify each other's commitments. If the verifications are successful, a group commitment is computed along with signature shares. In the final step, the group commitment and the individual signature shares are aggregated to form the final signature.

The security of Arctic, under the discrete logarithm problem and inside the random oracle model, assumes the availability of authenticated channels over the exchange of messages. However, Arctic and Shine, in their current forms, are not robust against misbehaving participants. To make them robust, Shine can be extended by requiring a minimum number of participants, $\mu \geq 3t - 2$, ensuring that any inconsistencies in commitments can be detected and the misbehaving parties identified. This approach allows Arctic to also function robustly by validating commitments and signature shares, while also requiring secure communication channels to prevent message replay and ensure liveness.

Contrary to some of the previously discussed schemes, Arctic (and Shine) have a practical deployment written in Rust, which is publicly available on GitHub [Gol24]. Their performance benchmarks show substantial improvements in computational overhead per signer compared to similar protocols like MuSig-DN [NRSW20], highlighting their efficiency in moderate group sizes.

4 Implementation

In this chapter, we present the implementation of the ROAST protocol in the Thetacrypt framework. We cover the integration with existing structures, deviations from the original ROAST protocol, and performance optimizations. We also outline the setup required for benchmarking to evaluate the performance of our implementation under realistic conditions.

4.1 Overview

Our implementation of ROAST produces Schnorr signatures over the `Ed25519` [JL17] elliptic curve. The protocol is implemented in Rust and integrated into the Thetacrypt framework [Cry24] to utilize its cryptographic operations and network communication. For the rest of this thesis, we refer to our implementation as Θ-ROAST. Θ-ROAST mostly follows the pseudocode outlined in ROAST [RRJ+22], with Σ = FROST and the deviations listed in Section 4.2.

We use the FROST implementation provided by the Thetacrypt framework, which is based on the FROST paper [KG20]. This implementation is a two-round variant of FROST with a trusted dealer for key generation without the use of the $\mathcal{SA}$ role. Therefore, the signing shares are broadcast to all signers and locally aggregated by each signer. Θ-ROAST cannot directly use FROST as a complete subprotocol due to differences in their structures, with Θ-ROAST being centralized and FROST being decentralized. However, we use the same cryptographic functions, code, and operations from FROST within Θ-ROAST. By employing the identical cryptographic components, we ensure a consistent basis for a fair comparison between the two schemes, despite their differing architectures.

4.2 Deviations from original ROAST

In Θ-ROAST with Σ = FROST, we closely followed the original pseudocode outlined, maintaining a centralized architecture featuring a single coordinator node and a group of participants. The distributed setting discussed in Section 3.3.5 was not implemented. However, we introduced several deviations from the original protocol to improve its efficiency and adapt it to the Thetacrypt framework. These deviations are discussed in the following.

4.2.1 Start signal

ROAST does not state how the nodes are initialized in a practical setting. We added an initialization round at the beginning of the protocol, where the coordinator broadcasts a start signal to all participants, indicating the readiness to receive messages. Only after receiving this start signal the signers are allowed to send messages to the coordinator. This handles the case of a slow-starting coordinator and ensures that the coordinator is initialized and ready to receive messages before the signers start sending their messages.

4.2.2 Marking malicious participants

The original approach to marking malicious participants, illustrated in Figure 4.1, allows for a specific scenario: If a signer provides a commitment and subsequently sends another message to the coordinator before receiving a new message, they will be marked as malicious. However, they still remain part of the set of responsive participants. This design may introduce inefficiencies in the following sense: the coordinator might start a signing session that includes a participant marked as malicious because this signer is still labeled as responsive. Although the signing session started in this way would never actually complete, it could needlessly trigger the performance of signing rounds by the other participants, only for the coordinator to ignore all messages from the participant it had marked malicious.

In our modified version of this protocol, we tackle this inefficiency by instantly excluding any participant identified as malicious from the group of participants that respond. This guarantees that the coordinator will never commence a signing session that includes a party that has been identified as malicious. The updated procedure is illustrated in Figure 4.2, with the particular alteration emphasized in line 2.

ROAST original

proc MarkMalicious(i)

1 : $M \leftarrow M \cup \{i\}$

2 : if $|M| > n - t$

3 : **fail**

Figure 4.1: ROAST MarkMalicious

Θ-ROAST deviation

proc MarkMalicious(i)

1 : $M \leftarrow M \cup \{i\}$

2 : $R \leftarrow R \setminus \{i\}$

3 : if $|M| > n - t$

4 : **fail**

Figure 4.2: Θ-ROAST MarkMalicious

4.2.3 Verification and sending the final signature to participants

In Θ-ROAST, we need an extra step beyond what is given in the pseudocode of the original paper. After the coordinator aggregates signature shares and computes a valid signature, it needs to send the final signature to the participants. This guarantees that the latter will have access to it and be able to verify it before utilization. The pseudocode in the original paper does not include this step, but it is necessary in practice. The final signature is already valid and cannot be exploited to forge new signatures when it is broadcast to all participants, including those marked malicious, using a gossip protocol—a very simple and efficient way of sending a P2P network message for broadcasting.

4.2.4 Performance optimization

We deviate from the pseudocode and implement an optimization mentioned in the experimental implementation of the original paper. We enable the coordinator to cache the group commitment $R = DE^b$ for

each FROST signing session. This optimization reduces the computational overhead for $\mathcal{C}$, allowing it to compute the joint group commitment R only once during the validation of each signature share in a signing session. Consequently, the overall protocol efficiency increases, the effect of cryptographic computations in the benchmarks is alleviated, and $\mathcal{C}$ can just reuse the cached value.

4.3 Implementation details

Θ-ROAST is not only the first robust threshold protocol implemented within Thetacrypt but also the first to introduce a central coordinator. This addition presented integration challenges because the existing Thetacrypt codebase was not designed to handle protocol instances where one party, $\mathcal{C}$, manages significantly more responsibilities than the other participants.

4.3.1 Coordinator

This subsection explains the role of the central coordinator in Θ-ROAST, highlighting its unique responsibilities and the modifications needed for its integration. To manage the coordinator's state, we implemented a structure called `RoastCoordinator`, as shown in Listing 1. This structure follows much of the ROAST protocol pseudocode but includes additional elements specific to our implementation. Two key aspects of this implementation are noteworthy.

First, we had to accommodate the dual role of a signer who also functions as the coordinator, as discussed in Section 4.4.1. Until now, existing protocols in Thetacrypt, like FROST, relied on each participant maintaining its own local state, with responsibilities evenly distributed among all signers. In Θ-ROAST, however, the coordinator's role is significantly more complex. To manage the additional responsibilities of the coordinator without compromising the structure of a protocol instance, we introduced a flag called `act_as_signer`. This flag is activated whenever $\mathcal{C}$ needs to perform tasks both as a coordinator and as a regular signer. For example, the flag allows $\mathcal{C}$ to smoothly switch between coordinator tasks (like aggregating shares or managing sessions) and signer tasks (like generating signature shares). This design maintains compatibility with the existing architecture while effectively managing the unique dual role within the protocol.

Second, the coordinator is responsible for maintaining detailed state information throughout the protocol. Unlike previous implementations, where all participants had identical roles and shared the same responsibilities, the dedicated coordinator role introduces additional complexity in state management. As shown in Listing 1, the coordinator tracks responsive and malicious signers, pre-signature shares, session details, signature shares, and group commitments. This required designing `RoastCoordinator` as a substructure within the protocol's main structure, extending a regular signer's capabilities to handle additional state and logic. This approach adheres to Thetacrypt's structural requirements while seamlessly integrating the coordinator role for Θ-ROAST.

4.3.2 Rounds

This subsection discusses the round structure in Θ-ROAST and how it differs from FROST. Within Thetacrypt, FROST is a two-round protocol, where each protocol execution consists of exactly two rounds. In contrast, Θ-ROAST can require more than two rounds, which posed a challenge for integrating our implementation into the existing structure of Thetacrypt. To achieve this integration, it was essential to divide the protocol into two phases that loop until a signature is successfully created: `do_round()` and `update()`. The protocol begins with the `do_round()` function to allow each participant to take their initial actions, followed by `update()` to process incoming messages and update the local state. While we will not delve into too much detail about these functions, it is important to understand their roles in the protocol.

```rust
pub struct RoastCoordinator {
    // Vec<signer_id> R: S_i is responsive if i in R
    responsive_list: Vec<u16>,
    // Vec<signer_id> M: S_i is known to be malicious if i in M
    malicious_list: Vec<u16>,
    // HashMap<(signer_id, signer_public_commitment)> P:
    // P[i] is the latest presignature share of S_i
    presignature_list: HashMap<u16, PublicCommitment>,
    // Session counter
    sid_ctr: u8,
    // HashMap<(signer_id, sid)> SID: SID[i] is the session that includes S_i
    session_list: HashMap<u16, u8>,
    // HashMap<(sid, Vec<signer_id>)> T:
    // T[sid] is the set of signer indices of session sid
    session_signer_list: HashMap<u8, Vec<u16>>,
    // HashMap<(sid, session_aggregated_presignature)> N:
    // N[sid] is the presignature of session sid
    session_presignature_list: HashMap<u8, PublicCommitment>,
    // HashMap<(sid, Vec<signer_signature_share>)> S:
    // S[sid] is the set of signature shares for session sid
    session_signature_share_list: HashMap<u8, Vec<RoastSignatureShare>>,
    // HashMap<(sid, group_commitment)> optimization:
    // G[sid] is the group commitment for session sid
    group_commitment_list: HashMap<u8, GroupElement>,
    // Used for Thetacrypt internal instance handling
    abort: bool,
    // Signals that we have t signature shares ready to assemble
    ready_for_signature: bool,
    // Signals that we have t fresh presignature shares
    ready_for_new_session: bool,
    // Coordinator is also a signer
    act_as_signer: bool,
    // Signals that the coordinator is in the initial round
    is_init_round: bool,
}
```

Listing 1: Definition of the `RoastCoordinator` structure, illustrating the data fields used to manage the state and responsibilities of the coordinator in the Θ-ROAST protocol. The comments are partially taken from the ROAST pseudocode to provide reference points for understanding the implementation.

In the first phase, starting with `do_round()`, each party determines whether they need to perform an action, such as sending a message to other nodes. This is followed by the `update()` function, where incoming messages are processed, and the local state is updated. For example, in FROST, each signer checks the incoming messages and stores the received presignature shares from other signers locally.

Next, each party determines whether the current state requires them to perform an action in the current iteration of the loop, specifically whether they need to run `do_round()`. Executing `do_round()` always results in sending a message to other node(s). In FROST, this means that the signer checks if it has received at least t presignature shares, $\rho_1, \ldots, \rho_t$, and if so, it aggregates them into a group presignature, ρ, and subsequently computes and outputs its signature share, σ_i. These examples have been simplified to provide a basic understanding of the process. In practice, these two phases include additional steps, such as verifying the validity of the shares.

For FROST, this process is relatively straightforward, as there are only two rounds of the protocol, and thus, two executions of `do_round()`. In the initial round, where each signer S_i is initialized, `do_round()` generates their local presignature share, ρ_i, and sends it to the other signers. Since FROST is implemented in a decentralized manner, all signers receive this message. The second execution of `do_round()`, as described earlier, involves aggregating the presignature shares from t signers and producing the final signature share.

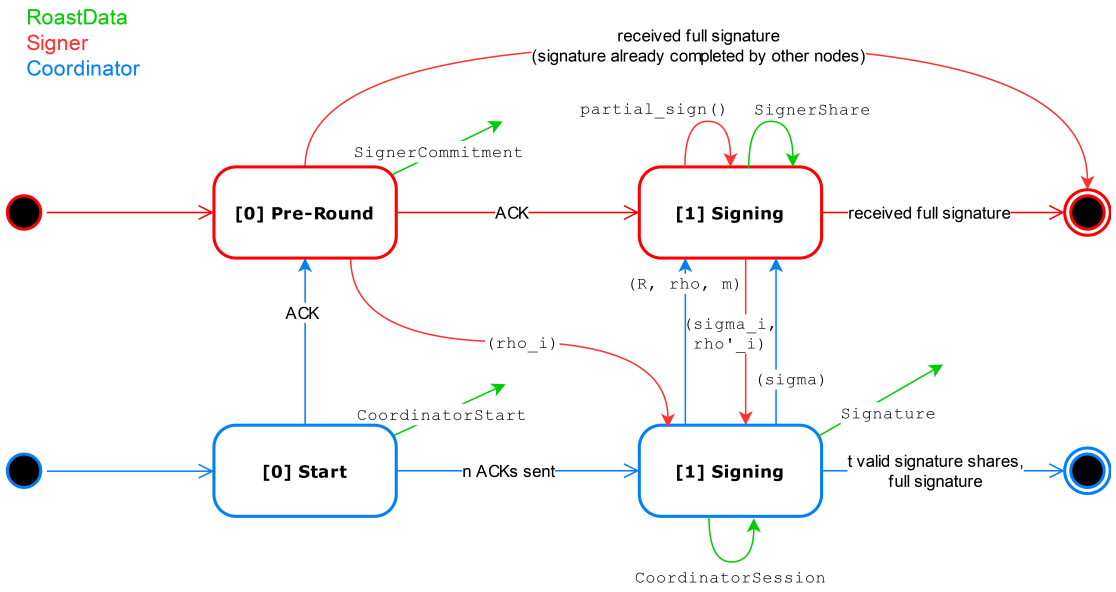


Figure 4.3: Overview of the Θ-ROAST rounds. The figure shows the flow of messages and highlights the dynamic communication patterns between the coordinator and the signers.

In Figure 4.3 we visualize the round procedure for Θ-ROAST, encompassing the two roles and the associated state transition between rounds. The figure highlights the different message types, namely `RoastData` (marked in green), which are used to manage communication between the coordinator and signers. These message types (`SignerCommitment`, `SignerShare`, `CoordinatorStart`, `CoordinatorSession`, and `Signature`) will be explained in more detail in the following section (see Section 4.3.3). Additionally, the figure shows the state transitions for a party acting as a signer. The second round, in this case *[1] Signing*, involves producing a signature share, σ_i, along with a fresh presignature share, ρ'_i, can be repeated multiple times for an honest node until the coordinator returns the final signature. In the worst-case scenario, where an honest node is involved in every unsuccessful signing

session (i.e., when the required number of valid shares is not reached), this second round can be repeated up to $f + 1$ times (see Section 3.3.2).

On the other hand, the coordinator in Θ-ROAST has a different set of responsibilities compared to regular signers, leading to different tasks in the `do_round()` function. Unlike FROST, where all signers perform the same actions, Θ-ROAST requires the coordinator to manage multiple rounds and handle messages differently. The coordinator's tasks include indicating readiness to begin the protocol by sending a starting signal (`CoordinatorStart`), initiating new signing sessions and distributing necessary data to selected signers (`CoordinatorSession`), and finally, aggregating and distributing the final signature (`Signature`). Additionally, the fact that one node simultaneously serves as both the coordinator and a regular signer further complicates the implementation. This was managed by using the `act_as_signer` flag, which allows the node to perform tasks appropriate to its dual roles.

4.3.3 Message content

Here, we discuss the content of messages exchanged in Θ-ROAST and how these messages are structured to handle the different roles of the coordinator and the signers, ensuring compatibility with Thetacrypt. The output of `do_round()` in both protocols results in a message sent to other parties: in FROST, the message is broadcast to all other signers, while in Θ-ROAST, it is sent either from a single signer to $\mathcal{C}$ or from $\mathcal{C}$ to selected signers. This reflects the centralized communication pattern in Θ-ROAST, compared to the decentralized approach of FROST.

```
// Outgoing message content in FROST
pub enum FrostData {
    Commitment(PublicCommitment), // Presignature share rho_i
    Share(FrostSignatureShare), // Presignature share rho_i
}

// Outgoing message content in ROAST, separate for signers and coordinator
pub enum RoastData {
    SignerCommitment(PublicCommitment), // rho_i
    SignerShare(RoastSignatureShare, PublicCommitment), // sigma_i, fresh rho'_i
    CoordinatorStart(),
    CoordinatorSession(
        Vec<u16>,         // Set of participating signers R
        PublicCommitment, // Presignature of a specific signing session rho
    ),
    Signature(FrostSignature), // Final signature sigma
}
```

Listing 2: Comparison of message enums between FROST and Θ-ROAST

Listing 2 builds upon Figure 4.3 and shows how Θ-ROAST separates message content to handle the different roles and responsibilities of $\mathcal{C}$ and the signers. This separation is a prerequisite for ensuring that Θ-ROAST integrates smoothly within Thetacrypt. Thetacrypt requires a single enumeration (`enum`) for messages while accommodating different types of message data for both the coordinator and the signers. Each message type in Θ-ROAST serves a specific function:

- The `SignerCommitment` message allows each signer to send their presignature share, ρ_i, to the coordinator.

- The `SignerShare` message enables signers to send their signature shares, σ_i, along with fresh presignature shares, ρ'_i, for future rounds.
- The `CoordinatorStart` message is used by $\mathcal{C}$ to signal that it is ready to begin the protocol, addressing the need discussed in Section 4.2.1.
- The `CoordinatorSession` message allows $\mathcal{C}$ to initiate a new session and distribute necessary data to selected signers.
- The `Signature` message communicates the final aggregated signature, σ, to all participants, as outlined in Section 4.2.3.

This message structure enables Θ-ROAST to handle the diverse communication needs of both the coordinator and the signers while remaining compatible with the existing setup in Thetacrypt.

4.4 Setup for benchmarking experiments

In this section, we outline the specific implementation details required to enable the benchmarking experiments described in Chapter 5.

4.4.1 Choosing the coordinator

In the original ROAST protocol, the coordinator can be any participant in the group or an additional, separate node. It is not specified *how* the coordinator is chosen. In Θ-ROAST, we appoint a specific participant as the coordinator while initializing the nodes. The coordinator is randomly selected at the beginning of each protocol execution based on its identifier. This is a strategy that would allow us to spread the extra computational load incurred by being a coordinator for benchmarking purposes. We came up with a technique similar in style to that presented in Mir-BFT [SDV19], but we didn't know about their work when we devised our method.

4.4.2 Simulating malicious parties

Θ-ROAST can run either in an honest mode without malicious parties (ROAST-HON) or in a malicious mode with parties behaving dishonestly (ROAST-MAL). We use ROAST-MAL to highlight the main feature of ROAST: robustness. We assume that an adversary can corrupt nodes, but it cannot partition the network in a way that prevents honest nodes from communicating with the coordinator. Additionally, we do not want those corrupt nodes to outright refuse participation in the protocol, because this would not hinder ROAST in any way. The remaining, honest nodes would be the only ones responding to the coordinator and immediately form a signing session and provide a valid signature. Similarly, having dishonest nodes failing to reply after sending their initial commitments would not increase the signing session attempts substantially, because the coordinator only puts responsive nodes in a new signing session. All honest nodes will eventually send their signature share to the coordinator, and they will eventually be put in a signing session together.

In our setup, it was not feasible to implement an adaptive adversary that could dynamically adjust its strategy during protocol execution. Achieving this would require communication between malicious nodes to coordinate their actions, allowing them to strategically fail one node per signing session. This would have brought in significant overhead, which can bias our benchmarking results by artificially increasing the network load and, hence, impacting performance metrics. For this reason, we simulate malicious signers by limiting the number of rounds in which they behave honestly. We use the variable `rounds_until_malicious`, which is `None` for honest nodes or a value of type `u16` for eventually

dishonest parties. In total, we want $n-t = f$ dishonest signers, with the node carrying the dual role of both coordinator and signer always being honest. The setup works in a way such that the first f non-coordinator signers are configured to be disruptive for the protocol.

The first malicious signer only behaves honestly for a single round. Concretely, this signer sends the initial commitments to the coordinator so that it is put into a signing session. This malicious signer then does nothing for the remainder of the protocol, effectively sabotaging the signing session it is in. The second malicious signer behaves honestly for two rounds so that it gets placed in a further signing session after participating honestly in the first one. Afterward, this node stops contributing to the protocol, thereby sabotaging another signing session. The third malicious node behaves honestly for three rounds, and so on. With this setup logic, we aim to maximize the possibility of requiring many signing sessions for ROAST to complete. In the worst-case scenario (from a protocol point-of-view), the protocol would need $n - t + 1$ signing sessions to complete, with each malicious node causing one session to fail. The exact code is displayed in Listing 3. Note that we use the constant `ATTACKER_LEVEL` to differentiate between ROAST-HON and ROAST-MAL. In practice, the response time and delay between the coordinator and signers are essential for selecting the initial t signers of the initial signing session.

```rust
const ATTACKER_LEVEL: u8 = 0; // 0 = No malicious behavior, _ = Malicious behavior

...

// Used for testing malicious signers: Computes how many rounds a malicious
// signer should behave honestly. The severity parameter can be used to adjust
// the behavior of the malicious signer.
// Returns rounds_until_malicious
fn setup_malicious_signer(
    signer_id: u16,
    total_signers: usize,
    threshold: usize,
    coord_id: u16,
) -> Option<u16> {
    let severity = ATTACKER_LEVEL;
    if severity == 0 || signer_id == coord_id {
        return None; // Coordinator is always honest
    }

    match severity {
        _ => {
            // In total we get n - t malicious signers.
            // The first n - t non coordinator signers are malicious
            // The first malicious signer only does one round,
            // the second malicious signer only does two rounds, etc.

            // We start with signer 1 unless the coordinator is signer 1
            let malicious_start = if coord_id == 1 { 2 } else { 1 };

            if signer_id >= malicious_start
                && signer_id < malicious_start + total_signers as u16
                    - threshold as u16
            {
                Some(signer_id - malicious_start + 1)
            } else {
                None
            }
        }
    }
}

```

Listing 3: Configuration of malicious signer behavior in ROAST-MAL, showing how the number of rounds a malicious signer behaves honestly is determined based on its identifier and the overall protocol setup. The `ATTACKER_LEVEL` constant differentiates between honest and malicious modes.

5 Benchmarking

Here, we present the benchmarking experiments conducted to evaluate the performance of Θ-ROAST in the Thetacrypt framework. We outline the benchmarking parameters, the benchmarked schemes, and the methods used to conduct the experiments. Finally, we discuss the results of the experiments.

5.1 Methods

5.1.1 High-level overview

The benchmarking experiments were conducted on the DigitalOcean[1] cloud platform, utilizing virtual machines (VMs) to deploy the Thetacrypt server and the benchmarking client. Refer to Figure 5.1 for a high-level overview of the benchmarking setup. Each VM carries a Thetacrypt server instance to function as a signer, while one node per run additionally acts as coordinator. The benchmarking client runs on a different VM, making the performance of the benchmarking client independent of the performance of the Thetacrypt servers. More important, we ran the Thetacrypt server and the benchmarking client on their corresponding VMs inside Docker containers[2].

5.1.2 Benchmarking parameters

Geographical distribution The experiments were conducted in two geographical distributions and with different numbers of participants to evaluate the protocol's performance under various conditions. In the *regional* setting all nodes were deployed in the same geographical region, while in the *global* setting, they were scattered across the available DigitalOcean datacenters all over the world. See section 5.1.4 for more details on the DigitalOcean regions used in the experiments.

Benchmarked schemes The decision was made to benchmark the existing FROST implementation in Thetacrypt as a baseline for comparison. The FROST implementation is a two-round variant of FROST

[1] `https://www.digitalocean.com/`
[2] `https://www.docker.com/`

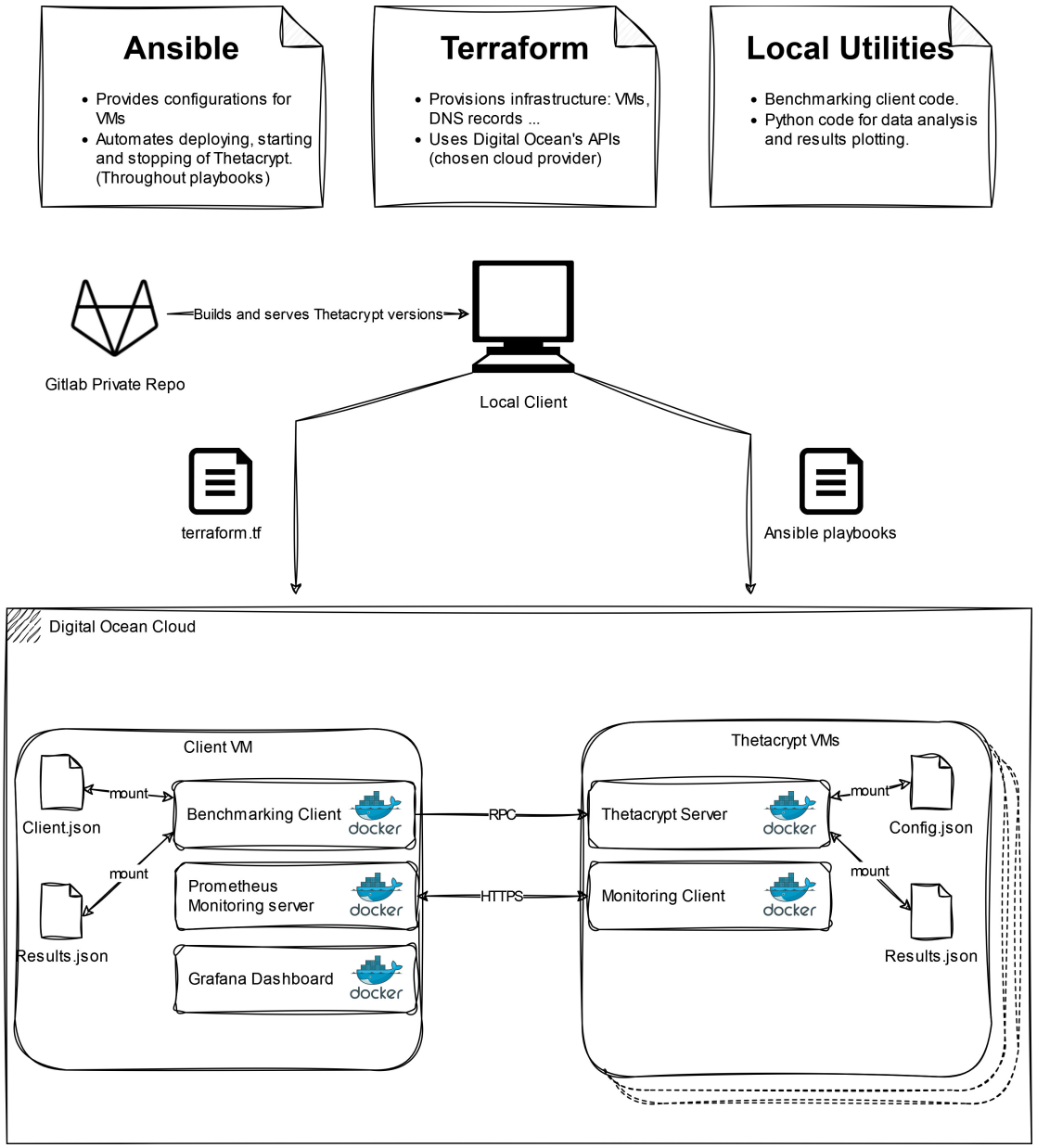


Figure 5.1: Benchmarking big picture

with a trusted dealer for key generation and without the use of the $\mathcal{SA}$ role. For Θ-ROAST we benchmarked the protocol without malicious participants, as well as with $f = \frac{n-1}{3}$ malicious participants, given that liveness is only guaranteed for $f < \frac{n}{3}$. With this setup, we aim to evaluate the performance of Θ-ROAST in scenarios with varying numbers of participants and thresholds, as well as different geographical distributions. Table 5.1 provides an overview of the benchmarked schemes, each of which ran in each of the scenarios described in Table 5.2.

Scheme	Malicious parties f
FROST	0
ROAST-HON	0
ROAST-MAL	$\frac{n-1}{3}$

Table 5.1: Benchmarked schemes

Number of nodes For the choice of the number of participants, we followed the first call for multi-party threshold schemes by NIST [BP23], which specifies six ranges: "two" for $n = 2$; "three" for $n = 3$; "small" for $4 \leq n \leq 8$; "medium" for $9 \leq n \leq 64$; "large" for $65 \leq n \leq 1024$; and "enormous" for $n > 1024$. The threshold parameter was chosen as $t = n - f$ to match the performance evaluation in the original paper [RRJ+22], where they were inspired by real-world examples, e.g., the federated Bitcoin sidechain Liquid [NPS20].

We conducted the experiments with $n = 7$ for the "small" range, $n = 34$ for the "medium" range, and $n = 127$ for the "large" range. The corresponding threshold values were $t = 5$, $t = 23$, and $t = 85$, respectively. Table 5.2 summarizes the deployment parameters used for benchmarking. Consequently, $f = 2$, $f = 11$, and $f = 42$ malicious participants were included in the experiments for the respective participant counts.

Number of parties n	Threshold t	Geographical distribution
7	5	Regional Global
34	23	Regional Global
127	85	Regional Global

Table 5.2: Deployment parameters for the benchmarking experiments

Message size We decided on a static message size of 256 bytes for all experiments. Previous tests conducted by the CRYPTO group with Thetacrypt, unrelated to this thesis, have shown that the message size affects the performance primarily during the serialization and deserialization processes. The benchmarked schemes use the same functions for both operations and thus the message size should not impact the results. The objective of our benchmarks is to evaluate the performance of the protocols, rather than the internal mechanisms of Thetacrypt.

Signatures We evaluate the threshold ciphers FROST and ROAST, with ROAST being benchmarked in both variants ROAST-HON and ROAST-MAL. The current version of Thetacrypt supports only the

`Ed25519` elliptic curve for both FROST and Θ-ROAST and thus this curve was utilized for all experiments.

Metrics We measure three primary metrics: *protocol completion rates*, *server-side latency*, and exclusively for Θ-ROAST, the *number of signing sessions*. The first two metrics are assessed using the protocol events `StartedInstance` and `FinishedInstance` on each node, capturing the time from protocol initiation on an instance to the successful termination of the protocol. Note that `StartedInstance` is only triggered when the signing process starts, meaning that preceding events like key generation are not included in the latency measurement.

For FROST, `FinishedInstance` is only triggered, when the signer was part of the t signers that produced a valid signature, leading to a maximum completion rate of $\frac{t}{n}$. This is not a limitation of the protocol itself, but rather of the implementation in Thetacrypt. For Θ-ROAST, `FinishedInstance` is triggered after the signer receives the final signature from the coordinator, see Section 4.2.3. This means that there is a communication overhead compared to FROST, as the participants need to receive the final signature from the coordinator. In the case of many sessions being needed, this can lead to a higher server-side latency. This is the trade-off to achieve robustness and in our case, a possible completion rate of 1.

Signing sessions are only measured for Θ-ROAST since, by definition, there is one and only one signing session per instance in FROST, whereas in Θ-ROAST, there might be multiple signing sessions per instance. The number of signing sessions is equal to the number of times the coordinator starts a new signing session and hence is a very good measure of the efficiency of the protocol. This is measured with the help of a counter in the coordinator node.

5.1.3 Benchmarking client setup

The benchmarking client is responsible for initiating benchmarking runs, monitoring the Thetacrypt server's performance, and collecting benchmarking data. It is written in Rust and uses a specific commit of the Thetacrypt library to instantiate the Thetacrypt server instances and create the signing requests. The benchmarking client is designed to run on a separate VM to avoid interference with the performance of the Thetacrypt servers.

Each Thetacrypt server instance produces a local CSV file with the logged events as described above. Similarly, the benchmarking client produces a CSV file with the instance IDs, allowing us to match the events with the logs of the server instances. Local timings (internal clock of VM synced with NTP) are used to determine the timestamps of those protocol events. We do not compare one VM to another; instead, we measure the relative difference of events within the same VM to ensure reliable measurements.

After the experiments, we gather this data using utility scripts. We then use `pandas`[3] to analyze the data and `matplotlib`[4] to visualize the results in graphs.

5.1.4 Infrastructure configuration

DigitalOcean VM specifications For our deployment, we utilized DigitalOcean VMs with varying specifications: The server nodes were configured with 2 GB of RAM and a single vCPU to efficiently handle their tasks without incurring unnecessary costs. Each node had a 50 GB SSD and 2 TB of data transfer capacity, which was sufficient for our needs.

In contrast, the monitoring node required a higher configuration due to the additional tasks it performed, including monitoring, benchmarking, and handling concurrent activities. Hence, it was configured with 16 GB of RAM, 4 vCPUs, and a larger 200 GB SSD. The increased transfer capacity of 8 TB would

[3] `https://pandas.pydata.org/`
[4] `https://matplotlib.org`

allow for a more than adequate bandwidth for intensive monitoring tasks. With this setup, in the maximal configuration that we used with $n = 127$ nodes.

DigitalOcean regions DigitalOcean's infrastructure provisioning spanned across 14 data centers in 9 geographic regions. However, due to resource limitations, the availability of specific data centers was restricted. Additionally, the Bangalore location was excluded based on previous tests that identified connectivity problems with other regions. Table 5.3 lists the 10 DigitalOcean regions used in the experiments.

Geographical distribution	Thetacrypt server VMs	Benchmarking client VM
Regional	Frankfurt 1	Frankfurt 1
Global	Frankfurt 1, New York City 1, New York City 3, Amsterdam 3, San Francisco 2, San Francisco 3, Singapore 1, London 1, Toronto 1, Sydney 1	Frankfurt 1

Table 5.3: DigitalOcean regions used in the benchmarking experiments

Deployment We leveraged existing automated deployment practices established by other projects within the CRYPTO group. The deployment process involved the following steps:

- **Compilation & packaging:** The application is compiled and distributed as Docker containers to ensure consistency and ease of distribution.
- **Infrastructure automation:** Terraform[5] is used for infrastructure provisioning, translating YAML-defined specifications into commands executed against the cloud provider's API.
- **Configuration management:** Ansible is used for configuration management, ensuring that the deployed VMs are correctly configured and ready for use.
- **Benchmarking:** The benchmarking process involves starting Thetacrypt server containers and the benchmarking client, collecting and aggregating benchmarking events, and automating tasks using Ansible and shell scripts.
- **Cleanup:** Terraform's built-in support for infrastructure cleanup simplifies the removal of provisioned resources after benchmark completion.
- **Helper scripts:** Utility scripts, written in bash, fetch data from the deployed nodes and assist in various tasks related to monitoring and management.
- **System monitoring:** The monitoring stack consisting of node_exporter[6] to collect metrics on each node, Prometheus[7] to collect and aggregate those metrics, and Grafana[8] to visualize that data in the form of dashboards. The monitoring stack is deployed on a separate VM to avoid interference with the performance of the Thetacrypt servers. As each experiment consisted of many runs, the monitoring of the performance of each VM was used to ensure that the nodes had enough resources

[5] https://www.terraform.io/
[6] https://github.com/prometheus/node_exporter
[7] https://prometheus.io/
[8] https://grafana.com/

available between each execution. Specifically, at least 90% of the CPU had to be idle and 50% of RAM before a new run was started by the benchmarking client. The dashboard can be seen in Figure B.1.

These practices ensure a consistent and reliable deployment process, enabling repeatable and comparable benchmarking experiments. Figure 5.1 provides an overview of the benchmarking setup, illustrating the deployment of the benchmarking infrastructure.

5.2 Results

5.2.1 Instance completion rates

In this Section, we present the results of the instance completion rates for the benchmarked schemes. For each scheme and deployment setting, we display only the results of the successful run with the lowest and highest message rates. Additionally, we list only those runs where the completed invocation rate is above 0.33 to ensure that the results are meaningful. Tables 5.4a and 5.4b show the invocation rate and the completed invocation rate. The invocation rate is defined as the number of messages sent per second. We experimented with message rates of 1, 2, 3, 5, 10, 15, 20, 25, 50, and 75, and stopped the experiments when the completion rate dropped below 0.33. Each benchmark run had a total duration of 60 seconds. The completed invocation rate is the ratio of completed instances to the total number of instances, ideally being 1.000 for Θ-ROAST and $\frac{t}{n}$ for FROST. Remember that the way we implemented Θ-ROAST, malicious nodes also receive the final signature from the coordinator and thus, can also terminate properly. Note that $\frac{5}{7} = 0.714$, $\frac{23}{34} = 0.676$ and $\frac{85}{127} = 0.669$. For the full results including intermediate steps, see Appendix C.

Before discussing the results, we note that the bottleneck in our experiments was often CPU usage, as the Docker container had access to the full CPU bandwidth of the VM. The nodes sometimes outright crashed, leading to the entire VM being down and requiring a manual power cycle for recovery. This prevents the signer instance running on said VM from participating in the current protocol execution. Thanks to our monitoring, we can pinpoint the crashes to full CPU utilization. Our preventive measure was to monitor the CPU and RAM usage of the nodes and only start a new run if the CPU usage was below 10% and the RAM usage below 50%, ensuring that each run had the same starting conditions. The RAM usage at most reached around 40%, highlighting that Thetacrypt is not memory-bound.

Moving on to the results, we observe that the higher the node count, the lower the invocation rate that the protocols tolerate. Intuitively, this is expected because the number of nodes in the network determines the number of sent and received messages. A higher invocation rate increases the load on the VM. Once this combined load exceeds the VM's capabilities, the nodes crash, and the completion rate drops. No scheme was able to handle an invocation rate of 75 messages per second.

Our experiments show that the Thetacrypt implementation of FROST handles rates up to 25 messages per second for $n = 7$ in both geographical settings. With an invocation rate of 1, FROST produced the maximum completion rate for both $n = 7$ and $n = 34$. For the latter, we were unable to get reliable results with an invocation rate higher than 1. FROST is barely runnable on $n = 127$. In fact, in the regional distribution, we were not able to produce a reliable result at all. We can only speculate, but perhaps the increased latency between the nodes in the global setting is beneficial for FROST, as the nodes have more time to process the messages and do not get overloaded as quickly. It's important to note that FROST is implemented in a decentralized variant within Thetacrypt, which inherently results in significantly higher communication overhead. This decentralized design, while beneficial for certain security properties, likely contributes to the difficulties in scaling to higher participant counts, as the increased communication requirements can overwhelm the network and processing capabilities, especially in larger configurations.

Scheme	n	Invocation rate	Completed rate
FROST	7	1	0.714
		25	0.710
	34	1	0.676
		-	-
	127	1	0.132
		-	-
ROAST-HON	7	1	1.000
		50	1.000
	34	1	1.000
		5	0.907
	127	1	1.000
		2	0.823
ROAST-MAL	7	1	1.000
		50	0.876
	34	1	1.000
		5	0.933
	127	1	0.983
		2	0.783

(a) Regional distribution

Scheme	n	Invocation rate	Completed rate
FROST	7	1	0.714
		25	0.669
	34	1	0.676
		-	-
	127	1	0.561
		-	-
ROAST-HON	7	1	1.000
		50	1.000
	34	1	1.000
		10	0.449
	127	1	1.000
		2	0.588
ROAST-MAL	7	1	1.000
		50	0.556
	34	1	1.000
		10	0.566
	127	1	0.861
		2	0.629

(b) Global distribution

Table 5.4: Reliability results showing the lowest and highest successful message rates for each scheme and deployment setting. FROST in the regional setting with $n = 127$ is included for comparison, despite a completion rate below 0.33. Note that FROST's maximum completion rates are $\frac{5}{7} = 0.714$, $\frac{23}{34} = 0.676$, and $\frac{85}{127} = 0.669$.

Overall, we assume these results are tied to the implementation of FROST in Thetacrypt and not to the protocol itself.

Generally, both Θ-ROAST variants produced very similar results. Increasing message rates yields a decrease in completion rate. We explain this by noting that, from time to time, nodes started to crash due to high CPU usage with a higher invocation rate. The handling of these unintended disruptions attests to the robustness of ROAST and the functionality of Θ-ROAST. Interestingly, in the global distribution, the saturation point is reached later for $n = 34$ at 15 messages per second, whereas the last reliable result in the regional setting was at 5 messages per second. There is no clear explanation for this; similarly to the latency results, this might be due to the distribution of VMs in the regions.

For $n = 127$ and an invocation rate of 1, only ROAST-HON achieved the maximum completion rate. This means that for all other schemes, some nodes crashed and did not finish properly. The saturation point for Θ-ROAST in $n = 127$ was at 3 messages per second for both variants in both geographical distributions, underlying the observation that ROAST-HON and ROAST-MAL performed similarly. The completion rate for both Θ-ROAST variants is lower at $n = 127$ with 2 messages per second in the global setting compared to the regional setting.

5.2.2 Latency at the server side

The server-side decryption latencies for the regional and global experiments are depicted in Figures 5.2 and 5.3, respectively. Note that the y-axis, where the latency is shown, is presented in logarithmic scale; for results in table form, see Appendix C. Naturally, the lower the latency, the better. For this portion of the results, we only considered the server-side latency for experiments conducted with an invocation rate of 1 message per second.

Overall, as expected, FROST exhibits the lowest latencies, followed by ROAST-HON and ROAST-MAL. This result is consistent across all schemes and scenarios and can be attributed to the additional communication rounds and cryptographic operations required by Θ-ROAST compared to FROST. The fact that ROAST-MAL has the highest latencies is also expected, as the malicious participants disrupt the signing sessions, leading to more rounds and thus longer protocol execution times.

The results indicate that server-side latency increases with the number of participants, which is also expected. We suspect the reason for this is twofold. First, the increased number of participants leads to more communication rounds, which in turn increases the overall time required to produce a valid signature, regardless of the protocol. Second, particularly for Θ-ROAST, the increased number of participants leads to more signing sessions, which leads to more rounds and thus, more messages. All of that increases the time spent waiting for the coordinator to start a new signing session. This also means that the signers need to wait longer for the coordinator to send the final signature, thereby increasing the server-side latency.

Similarly, we observe that the higher the number of participants, the more spread out the latency results become. The spread is very contained for $n = 7$ and $n = 34$, but different for $n = 127$, where a large spread is especially visible through long whiskers and a substantial number of outliers. This trend is consistent across all schemes and scenarios, indicating that the increased number of participants leads to a higher variance in server-side latency. Note that the outliers high on the y-axis appear clustered in the same spot due to the logarithmic scale's compression of larger values, which minimizes the visual differences between them. In contrast, lower outliers are more spread out because the logarithmic scale expands the smaller values, making even slight variations more distinct.

The general trend of slightly higher latencies in the global setting is consistent across all schemes and scenarios and can be attributed to the increased network latency between the nodes. The increased network latency leads to longer communication times between the coordinator and the signers, which in turn increases the server-side latency. However, $n = 7$ in the regional setting has much lower latencies than in the global setting for all schemes, whereas $n = 34$ and $n = 127$ have similar latencies in both settings. This substantial difference remains unexplained and would require further investigation or potential reruns

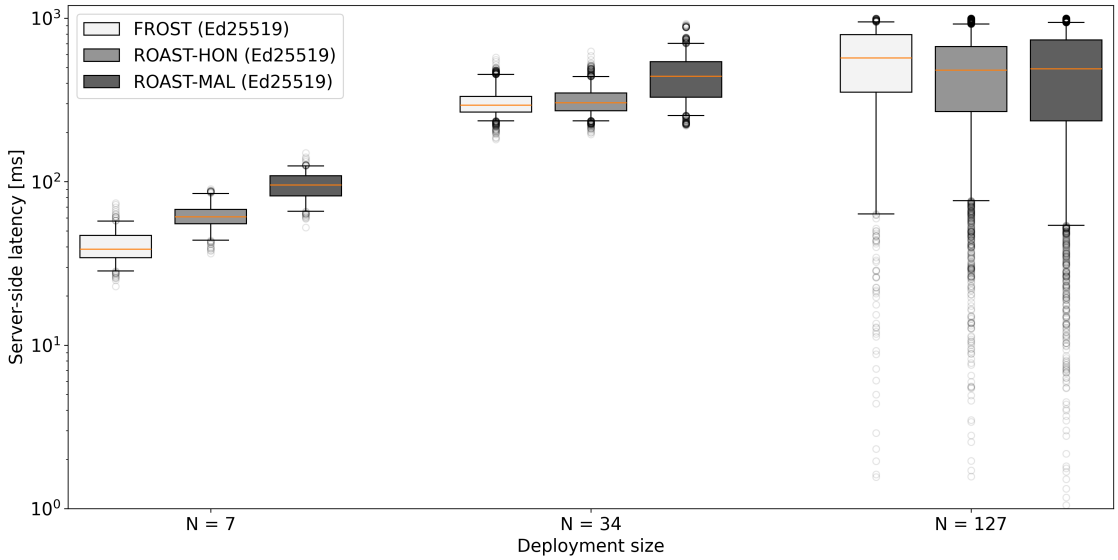


Figure 5.2: Server-side decryption latencies for regional deployment with invocation rate 1. The median (Q2) latency is represented by the horizontal orange line. The box plot extends from the first (Q1) to the third (Q3) quartile, while the whiskers span from the 5th to the 95th percentile, encompassing 90% of the data. Outliers are depicted as small circles.

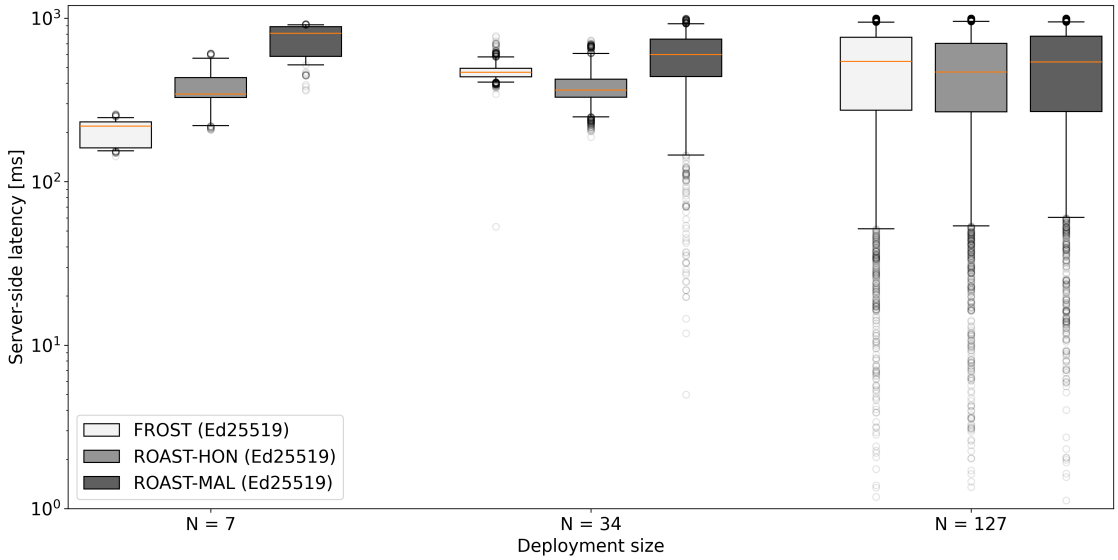


Figure 5.3: Server-side decryption latencies for global deployment with invocation rate 1. The median (Q2) latency is represented by the horizontal orange line. The box plot extends from the first (Q1) to the third (Q3) quartile, while the whiskers span from the 5th to the 95th percentile, encompassing 90% of the data. Outliers are depicted as small circles.

of the experiment. We suspect that this might be due to the exact location of the VMs in this specific experiment run, but they were equally distributed across the regions, the same as in the larger deployments, and thus, this is not a likely explanation.

Finally, there tends to be a wider spread in the server-side latency results for the global setting compared to the regional setting. We assume this is due to the increased difference in network latency between signer nodes and the coordinator in the global setting, leading to more variance in the server-side latency results. Signers located in Frankfurt are located in the same region as the coordinator and might experience lower latencies compared to signers located in other regions. Interestingly, in the regional setting, the 5th percentile of $n = 127$ deployments is in the same ballpark as the median of $n = 7$ deployments. This highlights the variance in the results with more participants and also the difference in performance between the two settings, where $n = 7$ performed substantially better. In the global setting, where $n = 7$ performed substantially worse, this is not the case.

Comparing Θ-ROAST results to ROAST [RRJ$^+$22] We briefly compare some of our results with the "static/non-coordinating strategy" evaluation in the original paper. In this scenario, f malicious signers are randomly chosen at the beginning of each run, and these malicious signers consistently fail to respond to any signing request. Intuitively, this matches the less interesting scenario we discussed in Section 4.4.2, and we would use their "static/coordinating" strategy to compare to ROAST-MAL. There, a single adversary coordinates the malicious participants: f malicious signers are again chosen randomly at the beginning of the run. In each FROST session containing malicious signers, they coordinate to ensure that only one of them disrupts the session by ignoring the signing request, leading to the maximal session count of $f + 1$. However, considering the achieved session count in our experiments averages around 2 (see Section 5.2.3), this would not be a fair comparison. Our approach did not simulate the worst-case as effectively, and thus, we argue that the comparison to the "static/non-coordinating strategy" is more appropriate.

The naive Python[9] implementation of FROST and ROAST used by Ruffing et al. [RRJ$^+$22] is publicly available [RJ22]. In their experiments, the average running time was measured by conducting 10 trials for each configuration and computing the average time taken to complete the signing process across those trials. Similarly to our experiments, they measured the running time locally on the signing nodes. Their coordinator was hosted on a VM in San Francisco, while the signers ran on a single VM in Frankfurt. Hence, we use our global results where ROAST-HON can be compared with their $f = 0$ variant and ROAST-MAL with $f = n - t$. The results for the smallest and largest n are shown in Table 5.5. Note that we compare averages and not medians, as the results in ROAST are averages as well.

Scheme	n	Average server-side latency
ROAST-HON	7	360 ms
	127	484 ms
ROAST-MAL	7	762 ms
	127	530 ms

(a) Average server-side decryption latencies

Scheme	n	Average running time
ROAST-HON	5	319 ms
	100	379 ms
ROAST-MAL	5	496 ms
	100	711 ms

(b) Average running times in ROAST [RRJ$^+$22]

Table 5.5: Selected average running times for Θ-ROAST with invocation rate 1 in the global setting in comparison with ROAST [RRJ$^+$22] for the static/non-coordinating strategy.

Our experiments produced similar results to the ones in Θ-ROAST for both variants. The average server-side latency for the honest variant of Θ-ROAST is similar to the results in the original paper, while

[9] `https://www.python.org/`

the malicious variant shows a larger discrepancy. The average server-side latency for the malicious variant is lower in our experiments than in Θ-ROAST. ROAST-MAL in our global deployment with $n = 7$ is rather slow, while with $n = 127$ it is faster than in the original paper. We believe that the discrepancy in the results may be due to the different network conditions and the use of different VMs for signers in our deployment.

Since the global benchmark with $n = 7$ seems to be an outlier, we add the regional results for additional context. In the regional setting, the average server-side latency for ROAST-HON is 62 ms with $n = 7$, and 96 ms with $n = 127$, whereas for ROAST-MAL, the latency is 96 ms for $n = 7$ and 445 ms for $n = 127$. These results suggest a much larger performance gap, favoring Θ-ROAST over the original setup. This further supports the idea that network conditions and the use of different VMs for signers in our deployment significantly influenced the observed latencies. For completeness, the actual worst-case scenario in the original paper averaged 6.428s in the 67-out-of-100 setup, which highlights the difference more sessions can make. For the small session count in our experiments, the difference is less pronounced, and the different implementations and the use of Rust instead of Python are not as impactful.

5.2.3 Number of signing sessions for ROAST signature

To investigate the effectiveness of ROAST-MAL in the presence of simulated malicious actors, we analyze the number of signing sessions required until it produces a valid signature. For comparison, we also examine the number of signing sessions for ROAST-HON, where nodes might be unresponsive even in an honest scenario, leading to potentially higher session counts. For clarity, we define a signing session as an attempt by the coordinator to collect signature shares from a group of participants to produce a valid Θ-ROAST signature. The results are based on experiments with an invocation rate of 1 per second and a total runtime of 60 seconds, resulting in 60 signature generation attempts per experiment.

The results are summarized in Tables 5.6 and 5.7. The tables provide the average and maximum number of signing sessions initiated to produce a valid Θ-ROAST signature for different numbers of parties and geographical distributions. Additionally, the tables list the number of distinct coordinators involved in the signing sessions. Note that only completed protocol runs are considered in the analysis because those are the only runs that provide meaningful data for the number of signing sessions. Consider Section 5.2.1 for more information on the completion rates of the protocols. Intuitively, we would expect the number of signing sessions to be close to 1 for ROAST-HON and higher in the presence of malicious actors, as they disrupt the signing process and necessitate additional attempts to produce a valid signature. If our simulation of malicious actors is effective, we would expect a number close to $f + 1$ for ROAST-MAL.

Number of parties n	Geographical distribution	Average session count	Maximum session count	Distinct coordinators
7	Regional	1	1	7
	Global	1	1	7
34	Regional	1.133	3	27
	Global	1	1	27
127	Regional	1	1	50
	Global	1.017	2	44

Table 5.6: Number of signing sessions initiated to produce a valid Θ-ROAST signature using scheme variant ROAST-HON. The results are gathered from a run with an invocation rate of 1 per second, leading to a total of 60 signature generation attempts per experiment.

Number of parties n	Geographical distribution	Average session count	Maximum session count	Distinct coordinators
7	Regional	2.050	3	7
	Global	2.483	3	7
34	Regional	2.050	6	30
	Global	2.533	4	28
127	Regional	2.017	5	52
	Global	1.774	6	42

Table 5.7: Number of signing sessions initiated to produce a valid Θ-ROAST signature using scheme variant ROAST-MAL. The results are gathered from a run with an invocation rate of 1 per second, leading to a total of 60 signature generation attempts per experiment.

Comparison and analysis The comparison of the results for ROAST-HON and ROAST-MAL reveals several insights:

- In the honest scenario, the average number of sessions required to generate a signature is consistently 1 or very close to 1 across all configurations of the number of parties and geographical distributions. This indicates that the protocol can efficiently produce a valid signature with minimal disruption when all participants behave honestly.
- In some instances, it is possible to observe more than one session, occasionally even three, in honest scenarios. This can occur due to various reasons, such as nodes crashing or experiencing delays in communication. Consequently, the first session may not always be the one to complete. The existence of concurrent sessions underscores the possibility of multiple honest sessions occurring simultaneously.
- The presence of malicious actors substantially increases the average number of sessions required to generate a signature compared to the honest scenario. This is evident across all configurations.
- The average of around 2 and maximum session count up to 6 in the malicious scenarios are higher than in the honest scenario, indicating that malicious actors cause more disruption, necessitating additional signing attempts. We would have liked to see even more disruption, but the results are still consistent with the expected behavior of Θ-ROAST in the presence of malicious actors. The way we set up the malicious actors (see Section 4.4.2) is limited, an adversary is not able to coordinate their actions, and the disruption is limited to the number of rounds a malicious actor behaves honestly.
- In the small deployment with $n = 7$, the number of distinct coordinators is equal to the number of parties, suggesting that each participant takes on the coordinator role at least once. For $n = 34$ we see a similar pattern, with the number of distinct coordinators close to the number of parties. In the large deployment with $n = 127$, the number of distinct coordinators is slightly below the session count of 60, indicating that some nodes might have taken on the coordinator role more than once. Overall, the results show that the coordinator selection process is effective in distributing the signing sessions across the available nodes.

Those observations are consistent with the expected behavior of Θ-ROAST in the presence of malicious actors. The protocol's robustness is demonstrated by its ability to handle disruptions caused by malicious participants, at the cost of required signing sessions and therefore at the cost of computational load. In contrast, FROST would not be able to complete and would need to rerun the whole protocol every time more than one session is required. The fact that Θ-ROAST generates a valid signature in the presence of

active adversaries and possible VM crashes, captured by maximum session count, proves its robustness and thus also the soundness of Θ-ROAST. Therefore, this provides very strong evidence that benchmarking a protocol's performance under adversarial conditions is important in establishing the resilience and effectiveness of a protocol in the real world.

6 Conclusion

6.1 Contributions

In this thesis, we implemented the ROAST protocol in the Thetacrypt codebase and extended the benchmarking setup. This implementation was used to benchmark both ROAST and FROST protocols in real-life scenarios with $n \in \{7, 34, 127\}$ nodes in a different manner than the original paper.

Benchmarking showed FROST had the lowest server-side latency, followed by ROAST-HON and ROAST-MAL. Latency increased with participants, with global settings showing slower signature generation due to increased network latency. An anomaly for $n = 7$ showed significantly lower latencies in regional settings compared to global ones. Increased numbers of participants also resulted in a greater variation in latency outcomes. Higher message rates resulted in lower completion rates due to high CPU usage, causing nodes to crash. FROST managed up to 25 messages per second for $n = 7$, but struggled with more participants. Θ-ROAST handled up to 50 messages per second for $n = 7$, but only 2 messages per second for $n = 127$. In honest scenarios, Θ-ROAST required around 1 signing session, increasing to an average of over 2 and a maximum of 6 with malicious actors. The coordinator role was well-distributed among participants.

While Θ-ROAST is slower than FROST, it offers greater robustness, especially with malicious actors. Both protocols are suffering from scalability issues; in particular, with global deployments, latencies are higher. Θ-ROAST's resilience to malicious actors and unintended VM failures allows it to be suitable for real-world applications needing robust threshold signature schemes.

Furthermore, the performance analysis in Arctic [KG24], discussed in Section 3.4.4, aligns with our findings by demonstrating efficient signature generation in groups of size $n \leq 25$ and significant reductions in computational overhead per signer compared to similar protocols. This supports our conclusions regarding the trade-offs between performance and robustness in threshold signature schemes.

6.2 Future work

Future work could explore several directions. Firstly, more advanced malicious actor simulations are required to see how Θ-ROAST would handle the actual worst-case with $n - t + 1$ signing sessions

compared to the original paper [RRJ+22]. This requires an adversary capable of coordinating the actions of malicious nodes in each round, which is not feasible with our current setup. To truly simulate the worst-case scenario, we would need an adversary that also controls the network, enabling them to delay the messages of honest parties, something we are currently unable to trigger. Secondly, in our experiments, it would be useful to understand the reason for the discrepancy between the regional and global setting for the $n = 7$ configuration properly and rerun the experiment for this configuration with nodes located in the same data centers. It would also be interesting to consider how different network conditions, various specifications of VMs, and different cloud providers impact the results. Running with higher granularity message rates and scaling up the number of nodes even further would bring more insight into this matter. Thirdly, it would be interesting to see ROAST's performance with a subprotocol Σ other than FROST, although currently only FROST meets all the requirements. Future work might be done by implementing and benchmarking protocols like HARTS [BLSW24] or Arctic, or any other follow-up work in Thetacrypt. The future for threshold cryptography, and in particular for Schnorr signatures, looks pretty exciting; benchmarking upcoming protocols will bring valuable insight.

Bibliography

[ABC+23] Orestis Alpos, Mariarosaria Barbaraci, Christian Cachin, Noah Schmid, and Michael Senn. Thetacrypt: A distributed service for threshold cryptography on-demand: Demo abstract. In *Proceedings of the 24th International Middleware Conference Demos, Posters and Doctoral Symposium, Bologna, Italy, December 11-15, 2023*, pages 33–34. ACM, 2023.

[ART23] Bank of Italy Applied Research Team. Optimized c library for ec operations on curve secp256k1. GitHub `https://github.com/bancaditalia/secp256k1-frost`, 2023. [Online; accessed 2024-04-18].

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptogr. Eng.*, 2(2):77–89, 2012.

[BHK+24] Fabrice Benhamouda, Shai Halevi, Hugo Krawczyk, Yiping Ma, and Tal Rabin. SPRINT: high-throughput robust distributed schnorr signatures. In *EUROCRYPT (5)*, volume 14655 of *Lecture Notes in Computer Science*, pages 62–91. Springer, 2024.

[BLS02] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *SCN*, volume 2576 of *Lecture Notes in Computer Science*, pages 257–267. Springer, 2002.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

[BLSW24] Renas Bacho, Julian Loss, Gilad Stern, and Benedikt Wagner. HARTS: high-threshold, adaptively secure, and robust threshold schnorr signatures. *IACR Cryptol. ePrint Arch.*, page 280, 2024.

[BN05] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 2005.

[BP23] Luís T. A. N. Brandão and Rene Peralta. NIST IR 8214C: NIST First Call for Multi-Party Threshold Schemes (Initial Public Draft). NIST Interagency/Internal Report (NISTIR) 8214C, National Institute of Standards and Technology, 2023.

[BTZ22] Mihir Bellare, Stefano Tessaro, and Chenzhi Zhu. Stronger security for non-interactive threshold signatures: BLS and FROST. *IACR Cryptol. ePrint Arch.*, page 833, 2022.

[BZ03] Joonsang Baek and Yuliang Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. In *GLOBECOM*, pages 1491–1495. IEEE, 2003.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *CCS*, pages 1769–1787. ACM, 2020.

[CGRS23] Hien Chu, Paul Gerhart, Tim Ruffing, and Dominique Schröder. Practical schnorr threshold signatures without the algebraic group model. In *CRYPTO (1)*, volume 14081 of *Lecture Notes in Computer Science*, pages 743–773. Springer, 2023.

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS*, pages 88–97. ACM, 2002.

[CKM21] Elizabeth C. Crites, Chelsea Komlo, and Mary Maller. How to prove schnorr assuming schnorr: Security of multi- and threshold signatures. *IACR Cryptol. ePrint Arch.*, page 1375, 2021.

[CKM23] Elizabeth C. Crites, Chelsea Komlo, and Mary Maller. Fully adaptive schnorr threshold signatures. In *CRYPTO (1)*, volume 14081 of *Lecture Notes in Computer Science*, pages 678–709. Springer, 2023.

[CKS05] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, pages 42–51. ACM, 1993.

[Cry24] Cryptology and Data Security Research Group CRYPTO, University of Bern. Thetacrypt repository. `https://github.com/cryptobern/thetacrypt`, 2024". [Online; accessed 2024-01-05].

[Des94] Yvo Desmedt. Threshold cryptography. *Eur. Trans. Telecommun.*, 5(4):449–458, 1994.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–437. IEEE Computer Society, 1987.

[Fou24] Zcash Foundation. Rust implementation of frost (flexible round-optimised schnorr threshold signatures). GitHub `https://github.com/ZcashFoundation/frost`, 2024. [Online; accessed 2024-04-18].

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *STOC*, pages 699–710. ACM, 1992.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *CCS*, pages 1179–1194. ACM, 2018.

[GJKR03] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure applications of pedersen's distributed key generation protocol. In *CT-RSA*, volume 2612 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2003.

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, 2007.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[Gol24] Ian Goldberg. Rust implementation of arctic and shine. GitHub `https://git-crysp.uwaterloo.ca/iang/arctic/src/main/src`, 2024. [Online; accessed 2024-08-06].

[GRJK07] Rosario Gennaro, Tal Rabin, Stanislaw Jarecki, and Hugo Krawczyk. Robust and efficient sharing of RSA functions. *J. Cryptol.*, 20(3):393, 2007.

[Gro21] Taurus Group. Implementation of the frost protocol for threshold ed25519 signing. GitHub `https://github.com/taurusgroup/frost-ed25519`, 2021. [Online; accessed 2024-04-18].

[GS22] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In *EUROCRYPT (1)*, volume 13275 of *Lecture Notes in Computer Science*, pages 365–396. Springer, 2022.

[HN06] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482. Springer, 2006.

[JL17] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[KG09] Aniket Kate and Ian Goldberg. Distributed key generation for the internet. In *ICDCS*, pages 119–128. IEEE Computer Society, 2009.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: flexible round-optimized schnorr threshold signatures. In *SAC*, volume 12804 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 2020.

[KG24] Chelsea Komlo and Ian Goldberg. Arctic: Lightweight and stateless threshold schnorr signatures. *IACR Cryptol. ePrint Arch.*, page 466, 2024.

[Kom20] Chelsea Komlo. FROST: flexible round-optimized schnorr threshold signatures. UCL Information Security Research Seminar `https://github.com/chelseakomlo/talks/blob/master/2020-12-10-ucl/ucl-presentation.pdf`, 2020. [Online; accessed 2024-03-14].

[MPSW19] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Des. Codes Cryptogr.*, 87(9):2139–2164, 2019.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2009.

[NPS20] Jonas Nick, Andrew Poelstra, and Gregory Sanders. Liquid: A bitcoin sidechain. Whitepaper, Blockstream Corporation, 2020.

[NRSW20] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. Musig-dn: Schnorr multi-signatures with verifiably deterministic nonces. In *CCS*, pages 1717–1731. ACM, 2020.

[Ped91a] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

[Ped91b] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In *EUROCRYPT*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer, 1991.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *J. Cryptol.*, 13(3):361–396, 2000.

[RJ22] Tim Ruffing and Elliott Jin. Naive implementation of roast protocol for robust threshold signatures. GitHub `https://github.com/robot-dreams/roast`, 2022. [Online; accessed 2024-08-04].

[RRJ+22] Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, and Dominique Schröder. ROAST: robust asynchronous schnorr threshold signatures. In *CCS*, pages 2551–2564. ACM, 2022.

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 1989.

[SDV19] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput BFT for blockchains. *CoRR*, abs/1906.05552, 2019.

[sec24] Frost signature scheme over secp256k1. GitHub `https://github.com/bitcoin-core/secp256k1`, 2024. [Online; accessed 2024-04-18].

[SG02] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *J. Cryptol.*, 15(2):75–96, 2002.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Sho00] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000.

[Sho23] Victor Shoup. The many faces of schnorr. *IACR Cryptol. ePrint Arch.*, page 1019, 2023.

[Sma16] Nigel P. Smart. *Cryptography Made Simple*. Information Security and Cryptography. Springer, 2016.

A
ROAST pseudocode

Figure A.1 presents the main algorithms of the ROAST protocol, with the signing functions displayed on top and the share validation and verification functions on the bottom. Figure A.2 shows the full ROAST protocol, including the main body of the algorithm and the event loop. The figures and conventions of the pseudocode are taken directly from the original paper [RRJ+22]. They use an event-based programming paradigm to account for the asynchronous network:

> After executing the code in the main body of the algorithm, the execution enters an infinite event loop that processes a queue of incoming network messages. Each message in the queue triggers the execution of an "upon receive" block. Further incoming network messages in the queue cannot be processed until after the "upon receive" block has finished executing (i.e., until the end of the block or a "break" instruction is reached). Multiple "upon receive" blocks (of the same algorithm) never run concurrently. If the queue is empty, the execution waits until a new message arrives over the network. The "send" keyword is used to send outgoing messages. The "return" keyword breaks the execution of the entire algorithm (i.e., not only the current block) and returns the indicated value. The "proc" keyword is used to define a subprocedure.

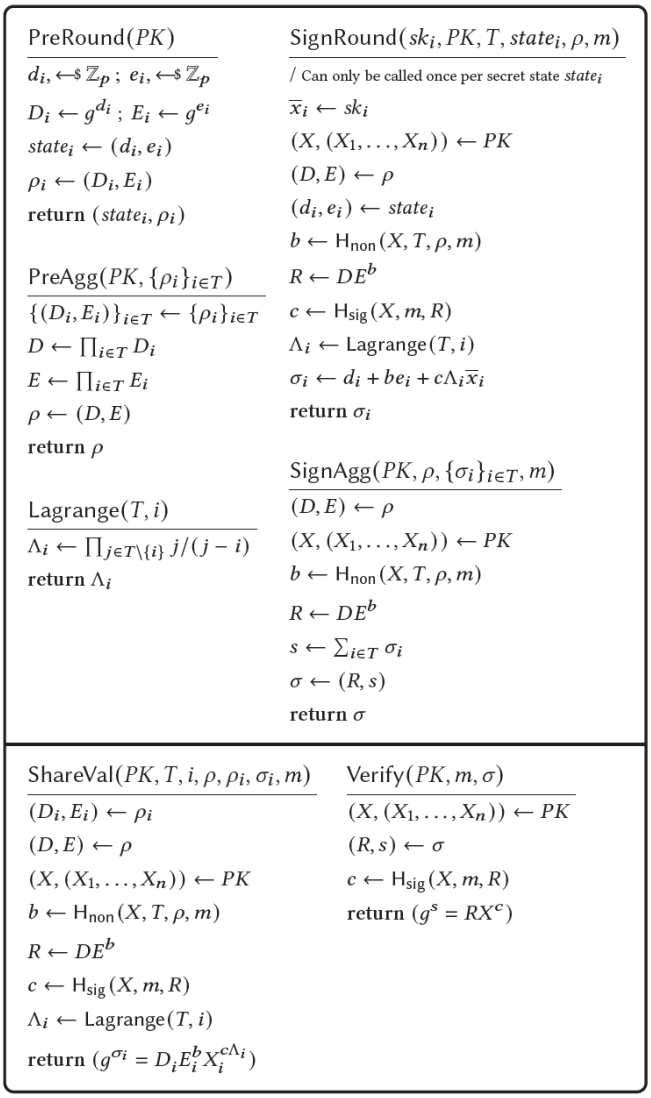
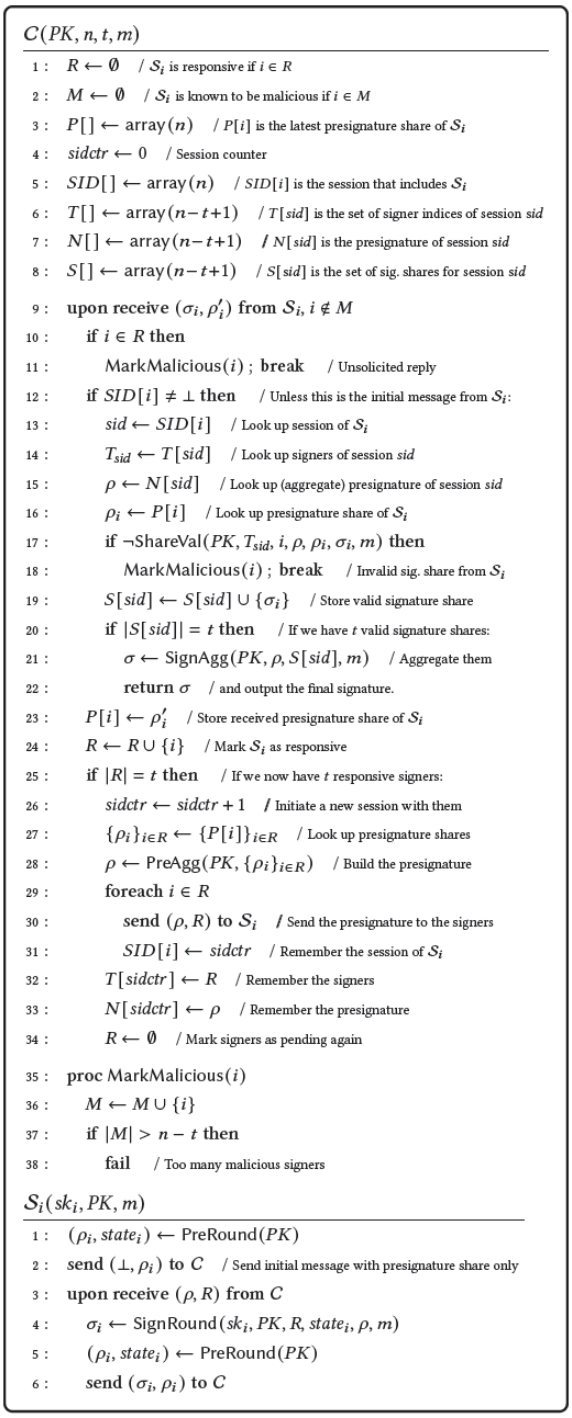


Figure A.1: Main algorithms of FROST = FROST3 [RRJ+22].



$C(PK, n, t, m)$

$R \leftarrow \emptyset$ / $\mathcal{S}_i$ is responsive if $i \in R$
$M \leftarrow \emptyset$ / $\mathcal{S}_i$ is known to be malicious if $i \in M$
$P[\,] \leftarrow \mathsf{array}(n)$ / $P[i]$ is the latest presignature share of $\mathcal{S}_i$
$sidctr \leftarrow 0$ / Session counter
$SID[\,] \leftarrow \mathsf{array}(n)$ / $SID[i]$ is the session that includes $\mathcal{S}_i$
$T[\,] \leftarrow \mathsf{array}(n-t+1)$ / $T[sid]$ is the set of signer indices of session sid
$N[\,] \leftarrow \mathsf{array}(n-t+1)$ / $N[sid]$ is the presignature of session sid
$S[\,] \leftarrow \mathsf{array}(n-t+1)$ / $S[sid]$ is the set of sig. shares for session sid
upon receive (σ_i, ρ_i') **from** $\mathcal{S}_i$, $i \notin M$
  if $i \in R$ **then**
    $\mathsf{MarkMalicious}(i)$; **break** / Unsolicited reply
  if $SID[i] \neq \bot$ **then** / Unless this is the initial message from $\mathcal{S}_i$:
    $sid \leftarrow SID[i]$ / Look up session of $\mathcal{S}_i$
    $T_{sid} \leftarrow T[sid]$ / Look up signers of session sid
    $\rho \leftarrow N[sid]$ / Look up (aggregate) presignature of session sid
    $\rho_i \leftarrow P[i]$ / Look up presignature share of $\mathcal{S}_i$
    if $\neg\mathsf{ShareVal}(PK, T_{sid}, i, \rho, \rho_i, \sigma_i, m)$ **then**
      $\mathsf{MarkMalicious}(i)$; **break** / Invalid sig. share from $\mathcal{S}_i$
    $S[sid] \leftarrow S[sid] \cup \{\sigma_i\}$ / Store valid signature share
    if $|S[sid]| = t$ **then** / If we have t valid signature shares:
      $\sigma \leftarrow \mathsf{SignAgg}(PK, \rho, S[sid], m)$ / Aggregate them
      return σ / and output the final signature.
  $P[i] \leftarrow \rho_i'$ / Store received presignature share of $\mathcal{S}_i$
  $R \leftarrow R \cup \{i\}$ / Mark $\mathcal{S}_i$ as responsive
  if $|R| = t$ **then** / If we now have t responsive signers:
    $sidctr \leftarrow sidctr + 1$ / Initiate a new session with them
    $\{\rho_i\}_{i \in R} \leftarrow \{P[i]\}_{i \in R}$ / Look up presignature shares
    $\rho \leftarrow \mathsf{PreAgg}(PK, \{\rho_i\}_{i \in R})$ / Build the presignature
    foreach $i \in R$
      send (ρ, R) **to** $\mathcal{S}_i$ / Send the presignature to the signers
      $SID[i] \leftarrow sidctr$ / Remember the session of $\mathcal{S}_i$
    $T[sidctr] \leftarrow R$ / Remember the signers
    $N[sidctr] \leftarrow \rho$ / Remember the presignature
    $R \leftarrow \emptyset$ / Mark signers as pending again
proc $\mathsf{MarkMalicious}(i)$
  $M \leftarrow M \cup \{i\}$
  if $|M| > n - t$ **then**
    fail / Too many malicious signers

$\mathcal{S}_i(sk_i, PK, m)$

$(\rho_i, state_i) \leftarrow \mathsf{PreRound}(PK)$
send $(\bot, \rho_i)$ **to** C / Send initial message with presignature share only
upon receive (ρ, R) **from** C
  $\sigma_i \leftarrow \mathsf{SignRound}(sk_i, PK, R, state_i, \rho, m)$
  $(\rho_i, state_i) \leftarrow \mathsf{PreRound}(PK)$
  send (σ_i, ρ_i) **to** C

Figure A.2: Full ROAST protocol [RRJ+22]

B
Monitoring

Figure B.1: Grafana dashboard showing the monitoring of the Thetacrypt servers during benchmarking experiments.

C Extended results

Here, we show extended tables of the results from the benchmarking experiments.

C.1 Server-side latency

Scheme	n	Lower 5th Percentile	First Quartile (Q1)	Median (Q2)	Third Quartile (Q3)	Upper 95th Percentile
FROST	7	28	36	39	43	57
	34	236	275	294	314	454
	127	64	435	572	726	951
ROAST-HON	7	45	58	61	64	85
	34	236	285	302	329	439
	127	74	323	466	611	925
ROAST-MAL	7	67	87	95	102	125
	34	254	354	438	489	702
	127	54	296	486	630	946

Table C.1: server-side latency in milliseconds, regional distribution

Scheme	n	Lower 5th Percentile	First Quartile (Q1)	Median (Q2)	Third Quartile (Q3)	Upper 95th Percentile
FROST	7	154	164	219	229	247
	34	407	449	467	483	580
	127	52	371	550	694	947
ROAST-HON	7	224	329	339	354	545
	34	246	339	361	392	603
	127	54	337	465	589	959
ROAST-MAL	7	446	587	784	870	911
	34	145	493	597	706	926
	127	62	377	552	703	951

Table C.2: server-side latency in milliseconds, global distribution

C.2 Instance completion rates

We only list results where the completed invocation rate is above 0.33 to ensure that the results are meaningful. The tables show the invocation rate, the number of completed instances, and the completed invocation rate. The invocation rate is the number of messages sent per second, the count of completed instances is the number of instances that finished successfully, and the completed invocation rate is the ratio of completed instances to the total number of instances. The way FROST is implemented in Thetacrypt, only the nodes participating in the signing session will properly terminate, leading to a maximum completion rate of $\frac{t}{n}$. Note that $\frac{5}{7} = 0.714$, $\frac{23}{34} = 0.676$ and $\frac{85}{127} = 0.669$.

Scheme	Invocation rate	Number of completed instances	Completed invocation rate
FROST	1	300	0.714
	5	1500	0.714
	10	3000	0.714
	15	4500	0.714
	20	5925	0.705
	25	7465	0.710
ROAST-HON	1	420	1.000
	5	2100	1.000
	10	4200	1.000
	15	6300	1.000
	20	8393	0.9996
	25	10500	1.000
	50	21000	1.000
ROAST-MAL	1	420	1.000
	5	2100	1.000
	10	4193	0.998
	15	6293	0.999
	20	8400	1.000
	25	8001	0.762
	50	18396	0.876

Table C.3: Reliability results, $n = 7$, regional distribution

Scheme	Invocation rate	Number of completed instances	Completed invocation rate
FROST	1	300	0.714
	5	1500	0.714
	10	3000	0.714
	15	4500	0.714
	20	6000	0.714
	25	7025	0.669
ROAST-HON	1	420	1.000
	5	2100	1.000
	10	4200	1.000
	15	6300	1.000
	20	8400	1.000
	25	10500	1.000
	50	21000	1.000
ROAST-MAL	1	420	1.000
	5	2100	1.000
	10	4200	1.000
	15	6300	1.000
	20	8400	1.000
	25	9406	0.896
	50	11669	0.556

Table C.4: Reliability results, $n = 7$, global distribution

Scheme	Invocation rate	Number of completed instances	Completed invocation rate
FROST	1	1380	0.676
ROAST-HON	1	2040	1.000
	3	6120	1.000
	5	9247	0.907
ROAST-MAL	1	2040	1.000
	3	3495	0.571
	5	9518	0.933

Table C.5: Reliability results, $n = 34$, regional distribution

Scheme	Invocation rate	Number of completed instances	Completed invocation rate
FROST	1	1380	0.676
ROAST-HON	1	2040	1.000
	3	6120	1.000
	5	10164	0.996
	10	9174	0.449
ROAST-MAL	1	2040	1.000
	3	6120	1.000
	5	10200	1.000
	10	11552	0.566

Table C.6: Reliability results, $n = 34$, global distribution

Scheme	Invocation rate	Number of completed instances	Completed invocation rate
FROST	1	1003	0.132
ROAST-HON	1	7680	1.000
	2	12539	0.823
ROAST-MAL	1	7492	0.983
	2	11926	0.783

Table C.7: Reliability results, $n = 127$, regional distribution. FROST is included for comparison, even though the completed invocation rate is below 0.33.

Scheme	Invocation rate	Number of completed instances	Completed invocation rate
FROST	1	4277	0.561
ROAST-HON	1	7620	1.000
	2	8964	0.588
ROAST-MAL	1	6560	0.861
	2	9587	0.629

Table C.8: Reliability results, $n = 127$, global distribution

Declaration of consent

on the basis of Article 30 of the RSL Phil.-nat. 18

Name/First Name: Schacher, Lukas Leo

Registration Number: 17-114-935

Study program: MSc Computer Science

Bachelor ☐ Master ☑ Dissertation ☐

Title of the thesis: ROAST in Rust: Implementing and Benchmarking Threshold Signature Schemes for Schnorr Signatures

Supervisor: Prof. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of 5 September, 1996 is authorized to revoke the title awarded on the basis of this thesis.

For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with future theses submitted by others.

Place/Date

Bern, 09.09.2024

Signature