

Universidade da Beira Interior

Department of Computer Science



**Departamento de
Informática**

Aokiji - A Rust Wallet for Nano Group Transactions Using FROST

By:

Diogo Gomes de Araújo

Advisor:

Professor Doctor Paul Andrew Crocker

September 10, 2025

Acknowledgements

This project would not have been possible without the unwavering support of my family. Their constant encouragement — both emotional and financial — has been invaluable throughout this journey. I am also profoundly grateful to my girlfriend for her understanding, patience, and continuous support.

I would like to express my deepest gratitude to Professor Doctor Paul Crocker for his exceptional guidance and mentorship. His insightful advice and steadfast support were instrumental in the successful completion of this project. I am truly thankful for the opportunity he provided me to undertake this work.

Contents

Contents	iii
List of Figures	v
1 Introduction	1
1.1 Context	1
1.2 Motivation UBI	1
1.3 Objectives	1
1.4 Document Structure	1
2 State of the Art	3
2.1 Introduction	3
2.2 Shamir Secret Sharing	3
2.2.1 Commitments and Verifiable Secret Sharing	4
2.3 Schnorr Signatures	4
2.4 Threshold Signature Schemes	5
2.5 Flexible Round-Optimized Schnorr Threshold Signatures (FROST)	6
2.6 Threshold ECDSA	6
2.7 Nault	7
2.8 Conclusion	8
3 Technologies and Tools Used	9
3.1 Introduction	9
3.2 Rust	9
3.3 Crates	11
3.3.1 curve25519-dalek	11
3.3.2 ed25519-dalek-blake2b	11
3.3.3 blake2	11
3.3.4 tokio	12
3.3.5 reqwest	12
3.3.6 dioxus	12
3.3.7 serde	13

3.4	Nano	13
3.4.1	Node	13
3.4.2	Remote Procedure Call (RPC)	13
3.5	Conclusion	14
4	Implementation	15
4.1	Introduction	15
4.2	Pre-Implementation	16
4.2.1	Tokio Run-time Testing	16
4.2.2	Dioxus, Reqwest and Nano's RPC Testing	17
4.2.3	Secret Sharing Prototype with rug	18
4.3	FROST Library	20
4.3.1	Centralized Messaging	22
4.3.2	FROST	24
4.3.2.1	Key Generation	24
4.3.2.2	Preprocess	27
4.3.2.3	Sign	27
4.3.3	Sockets	30
4.3.3.1	Server	30
4.3.3.2	Client	33
4.3.4	Nano	34
4.3.4.1	Unsigned Block	34
4.3.4.2	Processing the Transaction	36
4.4	Application	37
4.4.0.1	Graphical User Interface (GUI)	38
4.4.0.2	Integrating the Library	39
4.5	Testing	41
4.5.1	Protocol and Nano Tests	41
4.5.2	Socket Test	43
4.6	Conclusion	44
5	Conclusions and Future Work	45
5.1	Main Conclusions	45
5.2	Future Work	45
	Bibliography	47

List of Figures

2.1	Nault's Multi-signature Page	7
3.1	Nano's block lattice showing send and receive blocks that are created for each transaction, each signed by their account owner (A,B,C) from https://www.mycryptopedia.com/nano-block-lattice-explained/	14
4.1	Early Stage of the Application	17
4.2	Key Generation Flow Diagram	26
4.3	Sign Flow Diagram	29
4.4	Print of the Dashboard	40
4.5	Test Run Demonstration	42
4.6	Server running and waiting for participants	43
4.7	First participant client successfully computing the shared Nano account	43
4.8	Second participant client successfully computing the shared Nano account	43

Code Snippets

4.1	Typed Message Channel	16
4.2	Account Balance Request	17
4.3	Account Balance Response Struct	18
4.4	Typed Message Channel	18
4.5	Modular Arithmetic Module	19
4.6	Integer's Number of Bits	19
4.7	Shamir Secret Sharing (SSS) with rug	20
4.8	Method for Proof of Knowledge Computation	21
4.9	Hash to Scalar Method	21
4.10	Typed Message Channel	22
4.11	Message's Methods	23
4.12	Nano Account Address Lookup Table	25
4.13	Preprocess Method	27
4.14	FROST Server State	30
4.15	Send Message Method	30
4.16	Barrier Wait	32
4.17	Run Method	32
4.18	FROST Client State	33
4.19	Receive Message Method	34
4.20	Unsigned Block Structure	34
4.21	Signed Block Structure	36
4.22	Header Method Excerpt	38
4.23	Open and Connect Socket Closure Excerpt	40
4.24	Key Generation and Sign Testing Excerpt	42

Acronyms

UBI	Universidade da Beira Interior
CS	Computer Science
SSS	Shamir Secret Sharing
ECDSA	Elliptic Curve Digital Signature Algorithm
DSA	Digital Signature Algorithm
FROST	Flexible Round-Optimized Schnorr Threshold Signatures
ECC	Elliptic Curve Cryptography
RSA	Rivest-Shamir-Adleman
CPU	Central Processing Unit
SHA	Secure Hashing Algorithm
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
WASM	Web Assembly
RPC	Remote Procedure Call
GUI	Graphical User Interface
JSON	JavaScript Object Notation
DKG	Distributed Key Generation
UML	Unified Modeling Language
SA	Signature Aggregator
CSS	Cascading Style Sheets
NAT	Network Address Translation

Glossary

crate A crate is a compilation unit in Rust [1] comparable to libraries from other programming languages.. xi, 11, 14

Discrete Logarithm For the expression $Y = g^y \bmod p$ where g is a number from 0 to p and p is a large prime number, if a malicious party knows Y , g and p , there is still no optimized way of discovering y . xi, 5, 18

ED25519 A public-key signature system that uses Elliptic Curve Cryptography (ECC) to produce high-speed and high-security signatures [2].. xi, 11

Garbage Collector It is a form of automatic memory management invented by John McCarthy around 1959 with the purpose of relieving the developer from memory allocation and deallocation tasks [3]. xi, 9

Nano Nano is a decentralized cryptocurrency with zero fees and instant payments, that is easy to use and implement for developers. iv, xi, 1, 7, 11–13, 17, 25, 27, 34–37, 42–45

Rust Rust is a statically and strongly typed programming systems programming language that focuses on memory safety and efficiency. iii, xi, xiii, 1, 9, 10, 12, 13, 16, 46

Use After Free This is a common vulnerability in C/C++, where the developer who is responsible for allocating/deallocating memory continues to access memory after freeing it. These type of errors are not detected at compile time and can be difficult to identify and debug [4]. xi, 9

Chapter

1

Introduction

1.1 Context

The following report was written as the Final Project for my Computer Science (CS) Bachelor's Degree at Universidade da Beira Interior (UBI).

1.2 Motivation UBI

This document represents my journey as a CS student, serving as a checkpoint for my academic achievements as well as my ever-growing passion for programming, the Rust language, and blockchain-related work.

1.3 Objectives

The goal of this project was to make a Web Assembly (WASM) application that would allow users to create group accounts on the Nano blockchain, sign transactions collectively and, in the process, learn how blockchain transactions work, how to produce valid digital signatures and, finally, how to apply them in a practical setting.

1.4 Document Structure

To clearly represent the work completed, the project will be structured as follows:

1. First chapter – **Introduction** – introduces the project, the motivation behind its selection and the goals it aims to achieve.
2. Second chapter – **State of the Art** – evaluates the market for group blockchain transactions and the different protocols available to compute them.
3. Third chapter – **Technologies and Tools Used** – the technologies that were selected to build the project and ensure its robustness.
4. Fourth chapter – **Implementation and Testing** – outlines the project's development process and the tests conducted to verify the correct implementation of its functionalities.
5. Final chapter – **Conclusions and Future Work** – summarizes the key outcomes of the project and the establishes a direction for further development and improvement.

Chapter

2

State of the Art

2.1 Introduction

In this chapter, existing applications that try to solve the same and similar problems as this project will be explored. The following helped to understand the current market for blockchain group transactions, as well as the different signing protocols available to compute them.

2.2 Shamir Secret Sharing

Shamir Secret Sharing (SSS) is a cryptographic technique used to share secret information among a group. Each participant is assigned a secret share, but the group cannot uncover the secret unless a predefined number of participants (threshold) within the group join their secret shares. Secret Sharing Schemes were announced independently by both Shamir [5] and Blakley [6] in 1979.

Informally a threshold secret sharing scheme can be thought of as a method to divide a secret into a set of shares by an entity known as the dealer who then distributes these shares among a set of participants, where only qualified subsets of participants, at least equal to a given threshold, can recover the secret. They were called threshold (k, n) schemes by Shamir where k is the threshold value and n the size of the participant set. A scheme in which any subset of participants below a given threshold does not obtain any "information" about the secret, other than their own share, is called perfect. Under the scope of theory of information we can say, if the length of every share is the same as the length of the secret, which is the best possible case, these schemes are ideal [7].

Shamir's basic scheme is based on polynomial interpolation, it is secure and simple, but it has weaknesses that make it inadequate for threshold and group signature schemes:

- **Not Verifiable:** SSS doesn't prove that the secret that was reconstructed is the secret that was initially distributed.
- **Single Point of Failure:** Both when the secret is first distributed and when the secret is reconstructed, the secret exists at a single entity (the dealer, or reconstructor).
- **Not Reusable:** When sharing the secret shares to uncover the secret, the shares become invalid since they are not private anymore, making this scheme usable only once.

2.2.1 Commitments and Verifiable Secret Sharing

A cryptographic commitment scheme allows a sender to commit to a value (or statement) while keeping it hidden, with the ability to reveal the committed value later. This type of scheme has two primitives: binding, meaning the sender can't change the committed value after committing, and hiding, meaning the commitment reveals no information about the value until revealed.

An example of a Verifiable Secret Sharing is Feldman's Scheme [8] which adds check and encrypt primitives to Shamir's original protocol, thereby providing a public commitment for each share, visible to all participants. In this way, each participant can validate their share using this commitment. If this check fails, the participant can issue a complaint against the dealer and take actions such as broadcasting this complaint. The net result is that the group can ensure that the secret to be reconstructed is actually the secret initially distributed. Later we shall see that the FROST protocol also uses this technique.

2.3 Schnorr Signatures

A cryptographic signature scheme [9] is a mathematical protocol used to provide authentication, integrity, and non-repudiation for digital data. It allows a party to digitally sign a message using a private key, such that anyone with the corresponding public key can verify the signature's validity.

Other than Correctness, a cryptographic signature scheme requires other properties. These are namely Unforgeability, that prevents signature forgery, Integrity which protects messages from tampering and Non Repudiation

which prevents the signer denying that the signature was created. Also important for *real world* implementations are performance considerations, computational efficiency and for network and storage efficiency the signature's compactness or size.

Schnorr's signing algorithm [10] is one of the most popular for producing digital signatures, and it is known for its simplicity and efficiency. It uses the Discrete Logarithm problem with an agreed group G prime order q , a generator g and a cryptographic hash function. Formally, it consists of three polynomial time algorithms.

- **Key Generation:** Returns a public and private key pair.
- **Sign:** Signs a message M .
- **Verify:** Checks if the signed message is verifiable for the initial message given M and the public key.

2.4 Threshold Signature Schemes

A threshold signature scheme is a type of digital signature protocol where a group of participants collaboratively generate a valid signature, such that:

- At least t out of n participants are required to produce a valid signature.
- Fewer than t participants learn nothing about the signing key and cannot forge a signature.

This type of scheme provides fault tolerance, decentralization, and enhanced security for key management. However, unlike signatures in a single-party setting, threshold signatures require cooperation among the signers, each holding a share of a common private key, as well as issues during the key generation phase. Consequently, generating signatures in a threshold setting imposes overhead due to the number of network rounds required among signers, as well as key management issues where signers are required to hold only shares of a private key. This is particularly important in a cryptocurrency context where most wallets are designed to hold only account keys and not shares of keys and lists of participants.

2.5 Flexible Round-Optimized Schnorr Threshold Signatures (FROST)

FROST [11] is a protocol that reduces the number of rounds of communication required to produce Schnorr digital signatures, focusing on efficiency over robustness. In relation to prior protocols, it is still secure against misbehaving participants, but after identifying and blacklisting them, it aborts rather than trying to recover from the misbehavior.

This protocol uses a semi-trusted participant called Signature Aggregator (SA) to reduce communication overhead that will perform the signature aggregation.

Like in regular Schnorr Signature schemes, FROST has the Key Generation, Sign, and Verify functions. What makes it unique is the ability to include a Preprocessing round, that simplifies the signing phase by having participants publish a list of precomputed nonces and commitments that the SA can choose from.

Additionally, it uses Lagrange interpolation for threshold signing, allowing any subset of t out of n participants to generate a valid group signature.

In conclusion, FROST has many characteristics that make it great for computing group transactions:

- **Efficiency:** simplifies the communication and computation complexity.
- **Scalability:** Scales well in relation to the increasing number of participants since the number and size of the messages grows logarithmically.
- **Security:** It offers many security guaranties over forgeries and attacks and also provides Zero Knowledge Proofs support.

2.6 Threshold ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is one of the most used modern cryptography tools. It is an alternative to Digital Signature Algorithm (DSA) but it uses elliptic curve cryptography to create a secure digital signature scheme [12].

There are many Threshold ECDSA schemes (such as GG18, GG20, and CGGMP21) but they all have limitations that make them not as suitable as Schnorr threshold schemes like FROST:

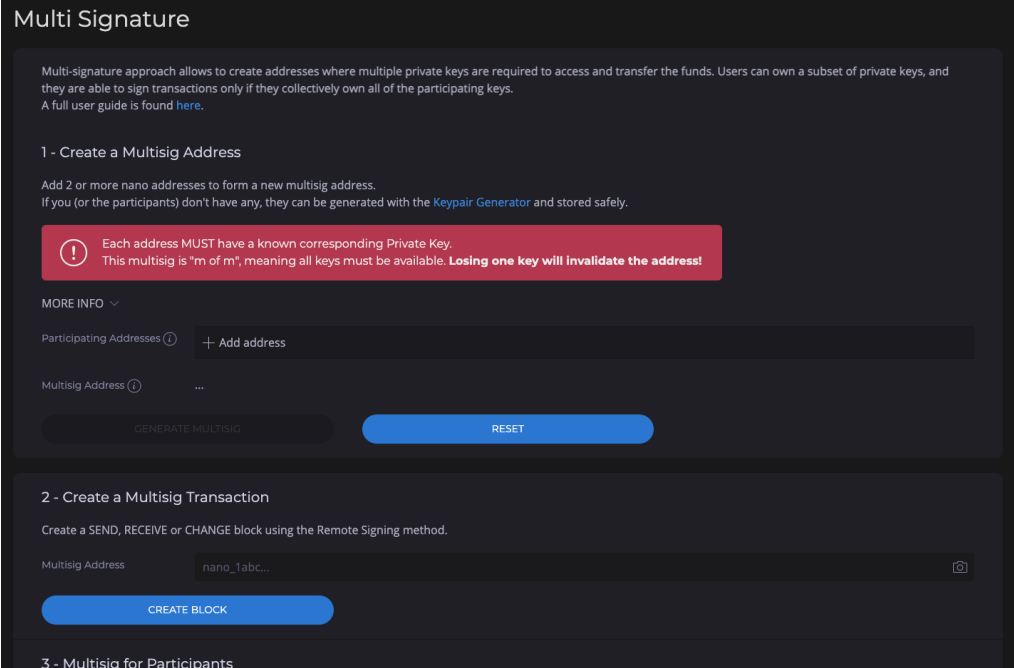
- **Requires Multiple Rounds:** Due to ECDSA mathematical structure, this threshold scheme requires at least 5 rounds of computations for adaptive security.

- **Latency:** Due to the large number of rounds and communications exchanged in this protocol, it can cause the user to experience higher latencies.
- **Security:** Signature schemes like FROST offer much more resistance to outside attacks and attempts of forgery.

2.7 Nault

Nault is a secure client-side only wallet for Nano. It allows users to sign transactions individually or as a group by selecting the *multisig* option in the advanced options.

Although group signing is provided, the process is quite complex. Users have to manually share the unsigned block hash with other participants, sign it, and process the block themselves, making it hard to use for inexperienced or non-technical users. Also it's a group not threshold signature.



The screenshot shows the 'Multi Signature' interface of the Nault wallet. It is divided into three main sections: 1 - Create a Multisig Address, 2 - Create a Multisig Transaction, and 3 - Multisig for Participants. Section 1 includes a warning that each address must have a known corresponding Private Key and that losing one key will invalidate the address. It features a 'Participating Addresses' list with an 'Add address' button and a 'Multisig Address' field. Section 2 includes a 'CREATE BLOCK' button. Section 3 is partially visible at the bottom.

Multi Signature

Multi-signature approach allows to create addresses where multiple private keys are required to access and transfer the funds. Users can own a subset of private keys, and they are able to sign transactions only if they collectively own all of the participating keys. A full user guide is found [here](#).

1 - Create a Multisig Address

Add 2 or more nano addresses to form a new multisig address.
If you (or the participants) don't have any, they can be generated with the [Keypair Generator](#) and stored safely.

Each address **MUST** have a known corresponding Private Key.
This multisig is "m of m", meaning all keys must be available. **Losing one key will invalidate the address!**

MORE INFO ▾

Participating Addresses ⓘ + Add address

Multisig Address ⓘ ...

GENERATE MULTISIG RESET

2 - Create a Multisig Transaction

Create a SEND, RECEIVE or CHANGE block using the Remote Signing method.

Multisig Address nano_1abc... ⓘ

CREATE BLOCK

3 - Multisig for Participants

Figure 2.1: Nault's Multi-signature Page

2.8 Conclusion

In summary, on one hand, there are many types of digital signatures and threshold signature schemes but FROST offers great security guarantees while keeping implementation and computation costs minimal. On the other hand, the Nano cryptocurrency doesn't have much support available for shared accounts, leaving a large market gap to be filled.

Chapter

3

Technologies and Tools Used

3.1 Introduction

Throughout the development of this project, there was a need to select different technologies, libraries, and languages to ensure its robustness. This chapter will explore them and their relevance to the project.

3.2 Rust

Rust is an up-and-coming systems programming language that focuses on memory safety (in contrast to languages like C). It is also known for its type safety, fast performance, and concurrency [13].

As opposed to other languages, Rust does not bound the developer to a single programming paradigm, offering functional support with immutability and pattern matching, but also object-oriented structs, enums and traits [14].

Although Rust is a memory-safe language it does not forsake performance. In particular, it does not use a Garbage Collector like Java or OCaml, nor does it allow the developer to handle memory dynamically like in C or Zig where memory leaks may occur. Rust disposes of a custom ownership model used by the *borrow checker* (Rust's compiler). The differences between the two are shown in 3.2. With compile-time memory safety checks, Rust is able to produce fast executables without the risk of Use After Free or memory corruption.

It also provides a default packet manager, Cargo, that centralizes the distribution and acquisition of dependencies and makes building and managing projects simpler. Additionally, it comes with a default documentation tool, `rust-doc`.

Table 3.1: Comparison of Rust Ownership Model vs Garbage Collection [15]

Feature	Rust ownership model	Garbage collection
Memory deallocation	Deterministic at compile time. Values are dropped when they go out of scope	Non-deterministic, typically during runtime pauses
Memory safety	Prevents memory errors (use-after-free, dangling pointers, etc.) at compile time	Relies on runtime checks to prevent memory errors
Performance overhead	Minimal runtime overhead. No background garbage collection processes	Can introduce runtime pauses and overhead due to tracing and collection
Predictability	Predictable memory usage and deallocation timing	Less predictable memory usage patterns, potential for runtime pauses
Development complexity	Requires understanding ownership and borrowing, can have a steeper learning curve	Often simpler for developers, as memory management is abstracted away

In recent years, Rust has become one of the most popular programming languages for cryptography and cryptocurrency-related applications due to its assurance over memory, thread safety, as well as performance.

Finally, I believe that Rust was a great choice for the development of this project due to its great ecosystem and tooling (Cargo) that made the development process easier and the application more robust. Moreover, the *borrow checker* always provided clear error handling and instructions, creating a frictionless and steady development cycle with a narrow gap for *bugs* and *human error*.

3.3 Crates

3.3.1 `curve25519-dalek`

Elliptic Curve Cryptography (ECC) is an approach to Public-Key Cryptography that uses elliptic curves over finite fields [16]. The curves are defined by the equation, $y^2 + xy = x^3 + ax + b$, and they share a property for *adding* two points to obtain a resulting one on the curve as well. By choosing an initial point g and adding the point to itself *scalar* amount of times (within a finite-field), the initial *scalar* and the resulting point on the curve form a public and private key pair that can be used in digital signature schemes. In relation to other systems like Rivest-Shamir-Adleman (RSA), it is more compact (an RSA 3,072-bit key has the same protection as a 256-bit ECC one [17]), it is faster and can operate with low Central Processing Unit (CPU) and memory resources.

The `curve25519-dalek` crate is a pure-Rust implementation of group operations on Ristretto and Curve25519 [18]. It provides an API to implement ECC key generation, digital signatures and zero-knowledge proof systems.

Lastly, it provides a great foundation for implementing FROST with ECC in a secure, performant and scalable manner.

3.3.2 `ed25519-dalek-blake2b`

The `ed25519-dalek-blake2b` crate is an implementation of ED25519 operations (key generation, signing and verifying). Since Dalek's default implementation uses Secure Hashing Algorithm (SHA) as its predefined hashing function, there was a need to search for an alternative.

This crate is a fork of the original `ed25519-dalek` repository, changing only the hashing algorithm used to Blake2b, the predefined hashing algorithm for Nano RPC operations. It will ensure that the FROST implementation produces signatures that are valid by Nano's standards.

3.3.3 `blake2`

The `blake2` crate is a Rust implementation of the BLAKE2 family of cryptographic hash functions [19], designed by Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2 was introduced as a faster and more secure alternative to older hash functions such as SHA-1 and MD5, focusing on high performance without sacrificing cryptographic strength.

A central variant provided by the crate is BLAKE2b, which is optimized for 64-bit platforms and produces digests of up to 64 bytes (512 bits). Given a

message M , it computes a hash value $H = \text{BLAKE2b}(M)$, offering strong collision and preimage resistance while being significantly faster than SHA-2 in software.

The `blake2` crate provides a flexible API suitable for FROST hash-based operations, ensuring that the resulting signatures are verifiable by Nano.

3.3.4 `tokio`

Tokio is an asynchronous runtime for Rust [20]. It provides support for developing fast, predictable and secure concurrent multi-threaded applications and an *async* version of the *standard library*.

Additionally, it is used to divide a thread's execution into several pausable tasks that yield so another can execute while another waits for resources. `tokio` re-implements components like `TCPStream` and `Mutex` making it great for producing robust network applications such as this project. It will provide building blocks to build the sockets used for generating keys and signing transactions as a group.

3.3.5 `request`

The `request` crate is an high level Hypertext Transfer Protocol (HTTP) Client for Rust [21]. Since the application needs to communicate with Nano's RPC via HTTP requests, an asynchronous HTTP client is essential for handling concurrent network operations efficiently.

3.3.6 `dioxus`

Dioxus is marketed as a developer-friendly framework that empowers developers to ship cross-platform apps with one codebase [22]. Unlike most modern front-end tools, it does not rely on JavaScript to build dynamic applications. Dioxus compiles the code to WASM, a portable binary-code format that can execute in a web environment [23].

There are many of the advantages of WASM in relation to regular JavaScript for building web applications.

- **Predictable Performance:** Since there is no garbage collector, there are no unpredictable pauses nor performance cliffs.
- **Smaller Sized Code:** Produces binary code that is less bloated (by removing garbage collection and dead code).

- **Cohesive Codebase:** By having the backend and frontend share the same programming language, the application is able to share types and methods between the two, and since Rust is a strongly typed language, it makes sharing states between the two simpler but also more secure.

Finally, despite this project being a desktop application, Dioxus is still a great choice as it is expandable to Linux, MacOS and Windows but also bestows the option to, in the future, make the application run on web or mobile environments.

3.3.7 serde

The `serde` crate is a framework for serialization and deserialization in Rust [24]. It implements many different data formats, most notably, JavaScript Object Notation (JSON), the predefined format for exchanging information on the *web*.

3.4 Nano

Nano is a decentralized digital payment protocol designed to be accessible and lightweight in contrast with cryptocurrencies like Bitcoin. It uses a Block Lattice (shown in 3.1), a DAG (Directed Acyclic Graph) based architecture. In this type of data structure, individual accounts control their own blockchain, making it ideal for higher-level signature schemes like FROST.

3.4.1 Node

A blockchain node is a device that runs the protocol software of a decentralized network (in this case, Nano) [25]. They execute and validate transactions based on majority consensus. Like so, they are able to ensure the blockchain's integrity and immutability. Data is stored transparently and can be accessed by any user.

To create accounts and sign transactions in Nano's blockchain, the application will need to communicate with a node.

3.4.2 Remote Procedure Call (RPC)

A RPC is a protocol that provides the high-level communications paradigm used in cryptocurrencies like Nano [26]. In order to consult and execute transactions in a node, the application will need to communicate via RPC calls

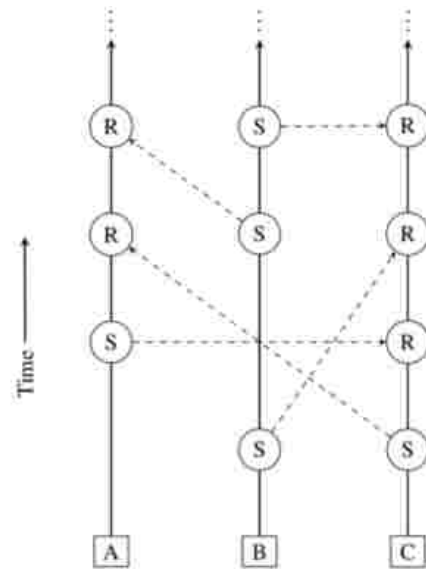


Figure 3.1: Nano's block lattice showing send and receive blocks that are created for each transaction, each signed by their account owner (A,B,C) from <https://www.mycryptopedia.com/nano-block-lattice-explained/>

(JSON HTTP POST requests), listed in Nano's RPC documentation. Some requests, like `work-generate`, might also require an API-KEY provided by the node in use.

3.5 Conclusion

In conclusion, to successfully develop this project there was the need to select and utilize different technologies, like crates, blockchain nodes and other tools, that together provide a steady foundation for this project, ensuring its integrity.

Chapter

4

Implementation

4.1 Introduction

This chapter portrays the development of the project, structured into sections that reflect the various phases and milestones along the way. Firstly, the pre-implementation and evaluation of technologies and protocols will be discussed, followed by the development of the library that integrates FROST and blockchain transaction processing. Finally, it concludes with the construction of the application itself and testing of its functionalities, security, resistance to forgery and the comparison between the result and the requirements.

1. **Pre-Implementation:** Before starting this project, it was important to grasp the fundamentals and understand the protocols available to produce digital signatures collectively as well as evaluate the technologies that would be used to make the front-end of the application.
2. **Library:** The implementation of FROST's Key Generation, Sign and Preprocess functionalities plus the network infrastructure to perform transactions in a group setting.
3. **Application:** Making the application's Graphical User Interface (GUI).
4. **Testing:** After each step, there was a need to validate the stability of each functionality against attacks, forgery attempts and improbable edge cases.

4.2 Pre-Implementation

Before starting this project, there was a need to establish its requirements and reshape them into a clear road-map for the development. They are listed as follows:

- Support for Distributed Group Signing;
- Strong Cryptographic Security;
- Blockchain Transaction Processing;
- Transaction History Transparency;
- Group Account Management.

Additionally, it was essential to conduct a thorough technology evaluation and testing to ensure suitability with the project's objectives.

4.2.1 Tokio Run-time Testing

Since the goal of this project was to create a wallet, in Rust, that would allow users to sign transactions together, it required a way to facilitate communications. On account of FROST already asking for a semi-trustworthy participant SA, a Transmission Control Protocol (TCP) socket, opened by the SA, that other participants could access as clients to sign the transaction, would be sufficient to let participants exchange messages remotely and securely, instead of a more decentralized peer-to-peer connection.

Rust does not provide asynchronous support for sockets in the *standard library* and, seeing that they would need to handle multiple connections and messages simultaneously, it was necessary to look for an alternative. As previously mentioned, *tokio* is an async run-time for Rust and it has building blocks like *TCPStream* and *TCPListener* for TCP async socket implementations. After testing out the chat app example, provided in *tokio*'s main Github repository, with functionalities like *UnboundedSender* and *UnboundedReceiver* that limit the message channel to a type (like a predefined *Message* enum) this library demonstrated itself as the right choice for handling group messaging.

```
/// Shorthand for the transmit half of the message channel
pub type Tx = mpsc::UnboundedSender<Message>;

/// Shorthand for the receive half of the message channel.
```

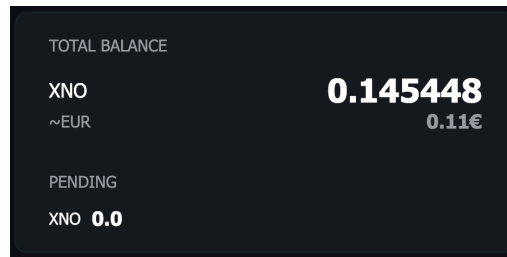



Figure 4.1: Early Stage of the Application

```
pub type Rx = mpsc::UnboundedReceiver<Message>;
```

Code Snippet 4.1: Typed Message Channel

4.2.2 Dioxus, Reqwest and Nano's RPC Testing

It was also formerly stated that the application should use a WASM framework, excluding popular options like GTK 4 and Tauri. Dioxus stood out for its performance, small learning curve (using concepts from ReactJS and HTML), but most importantly, because it allows the application to compile to different run time targets such as web or mobile environments in the same codebase.

Considering that Dioxus uses reqwest as the preferred HTTP client, to test both the front-end building tools and the requests to a Nano node RPC, a simple website was built that displayed an account's balance in Nano and the real-time equivalent in Euro.

To get the account's balance from the RPC, a POST request was sent with the account's address and the action (account-balance) in a JSON body like shown in 4.2. The struct in 4.3 represents the JSON body of the message received from the node. It includes the Serialize and Deserialize, making it parsable to/from JSON with serde. Finally, to utilize the request inside the application, use-resource defines the account balance future as a reactive asynchronous resource, receiving a Closure that calls get-account-balance. It retrieves the value of the account balance wrapped in an Option type (since the operation might fail or get delayed at run-time).

```
pub async fn get_account_balance(account: &str) ->
    AccountBalanceResponse {
    let client = reqwest::Client::new();
    let data: HashMap<_, _> = [("action", "account_info"),
        ("account", account)].into();
    let response = client.post(RPC_URL).json(&data).send().
        await.unwrap();
```

```
response.json::<AccountBalanceResponse>().await.unwrap()
}
```

Code Snippet 4.2: Account Balance Request

```
#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct AccountBalanceResponse {
    pub balance: Option<String>,
    pub pending: Option<String>,
    pub receivable: Option<String>,
    pub balance_nano: Option<String>,
    pub pending_nano: Option<String>,
    pub receivable_nano: Option<String>,
}
```

Code Snippet 4.3: Account Balance Response Struct

```
let balance_future = use_resource(|| async {
    get_account_balance(account).await });
let balance_info: AccountBalanceResponse = match &*
    balance_future.read_unchecked() {
    Some(res) => (*res).clone(),
    None => AccountBalanceResponse::new(),
};
```

Code Snippet 4.4: Typed Message Channel

4.2.3 Secret Sharing Prototype with rug

Understanding the theoretical concepts behind FROST was crucial before starting the main implementation. Since Shamir's Secret Sharing scheme shared many conceptual similarities with FROST but was considerably simpler, it served as an ideal foundation for grasping the knowledge required to build it.

In addition, before implementing ECC it was also essential to learn about finite field arithmetic and the Discrete Logarithm problem, making `rug`, a crate that provides arbitrary-precision numbers, ideal as a stepping stone before `curve25519-dalek`.

Finite field arithmetic differs from normal mathematics since, when performing operations (addition, subtraction, multiplication, and division), the resulting number is constrained within a bounded algebraic structure. If an operation exceeds the field's boundary, the result wraps around using modular arithmetic, keeping all values within the predefined range of the field. This was implemented in `modular.rs`, shown in 4.5.

```

/// Function to calculate the modular addition of two values
pub fn add(a: Integer, b: Integer, m: &Integer) -> Integer {
    ((a.modulo(m)) + (b.modulo(m))).modulo(m)
}

/// Function to calculate the modular subtraction of two
values.
pub fn sub(a: Integer, b: Integer, m: &Integer) -> Integer {
    ((a.modulo(m)) - (b.modulo(m)) + m).modulo(m)
}

/// Function to calculate the modular multiplication of two
values.
pub fn mul(a: Integer, b: Integer, m: &Integer) -> Integer {
    ((a.modulo(m)) * (b.modulo(m))).modulo(m)
}

/// Function to calculate the modular division of two values
pub fn div(a: Integer, b: Integer, m: &Integer) -> Integer {
    let a = a.modulo(m);
    let inv = b.invert(m).expect("No modular inverse exists
    ");
    (inv * a).modulo(m)
}

/// Function to calculate the modular power of two values.
pub fn pow(x: &Integer, y: &Integer, p: &Integer) -> Integer
{
    match x.clone().pow_mod(y, p) {
        Ok(i) => i,
        Err(_) => unreachable!(),
    }
}

```

Code Snippet 4.5: Modular Arithmetic Module

Integer is an arbitrary-precise integer type, bound to 256-bit by the constant value BITS in lib.rs 4.6.

```
pub const BITS: u32 = 256;
```

Code Snippet 4.6: Integer's Number of Bits

Finally, the implementation for the protocol was based on a C++ one found on <https://www.geeksforgeeks.org/computer-networks/shamirs-secret-sharing-algorithm-cryptography/>. The create-secret-shares function distributes a secret into multiple shares while the recover-secret re-

covers the secret from its shares precisely. This is only possible because finite-field arithmetic allows numbers to grow indefinitely within a limited space, avoiding variable overflows. Both functions can be found in 4.7.

```

/// Function that creates the secret shares.
pub fn create_secret_shares(
    key: Integer,
    k: u64,
    n: u64,
    prime: &Integer,
    rnd: &mut RandState,
) -> Vec<(Integer, Integer)> {
    let pol = generate_pol(key, k, rnd);
    let mut shares: Vec<(Integer, Integer)> = Vec::new();
    let mut xs = Vec::new();

    for _i in 0..n {
        let x = generate_unique(rnd, &xs);
        xs.push(x.clone());

        let y = calculate_y(&x, &pol, prime);
        shares.push((x, y));
    }

    shares
}

/// Function that recovers the secret.
pub fn recover_secret(shares: &[(Integer, Integer)], prime:
&Integer) -> Integer {
    lagrange_pol(&Integer::from(0), shares, prime)
}

```

Code Snippet 4.7: SSS with rug

The implementation of the prototype can be found at <https://github.com/diogogomesaraujo/frost-sig/tree/shamir>.

4.3 FROST Library

For the development of the FROST crate, a dedicated repository was created to ensure it remains adaptable for other applications beyond this specific use case. The repository is publicly available at <https://github.com/diogogomesaraujo/frost-sig>.

Additionally, to fundamentally understand the implementation of FROST, the method shown in 4.8 and the sub function 4.9 will be used as an example. In these methods, we can see the following

- The function `Scalar::random` from `curve25519-dalek` generates a random scalar using a deterministic random number generator, in this case sourced from the operating system.
- Edwards curve points, such as `ri`, are computed by multiplying a `Scalar` with the curve's base point, `ED25519-BASEPOINT-POINT`.
 - Notice that since `EdwardsPoint` structures in `curve25519-dalek` consume substantial memory, they are compressed when not in use and decompressed when needed.
- We can also see that the hash-to-scalar function hashes a `Vector`-typed argument using `Blake2b` and converts the resulting digest into a `Scalar`, as illustrated in 4.9.

```

/// Function that computes a participant's challenge and
/// encrypted response.
pub fn compute_proof_of_knowledge(
    rng: &mut OsRng,
    participant: &Participant,
) -> (Scalar, Scalar) {
    let k = Scalar::random(rng);
    let ri = k * ED25519_BASEPOINT_POINT;
    let ci = {
        let mut hasher = vec![];
        hasher.extend_from_slice(&participant.id.to_le_bytes());
        hasher.extend_from_slice(
            (participant.polynomial[0] *
             ED25519_BASEPOINT_POINT)
                .compress()
                .as_bytes(),
        );
        hasher.extend_from_slice(ri.compress().as_bytes());
        hash_to_scalar(&[&hasher[..]])
    };
    let wi = k + participant.polynomial[0] * ci;
    (wi, ci)
}

```

Code Snippet 4.8: Method for Proof of Knowledge Computation

```

/// Function that hashes a 'Scalar' using 'Blake2b'.
pub fn hash_to_scalar(inputs: &[&[u8]]) -> Scalar {
    let mut h: Blake2b<U64> = Blake2b::new();
    for i in inputs {
        h.update(i);
    }
}

```

```

    }
    let hash = h.finalize();
    Scalar::from_bytes_mod_order_wide(&hash.into())
}

```

Code Snippet 4.9: Hash to Scalar Method

4.3.1 Centralized Messaging

As stated earlier, `tokio` allows developers to create a centralized message data structure for communication and Rust has powerful enums where each member can hold any kind of data, like structs.

TCP is a stream-based protocol, not message-based. This means that data sent in chunks by the sender might arrive in different-sized chunks at the receiver and thus a single receive call may not receive a full message, or it may receive multiple messages. This is especially fragile when sending different enum variants, because their encoded sizes vary and therefore the client has no idea how many bytes to expect before trying to deserialize. Tokio provides this the solution *out of the box* using the `tokio-util::codec::Framed` trait.

To take advantage of these features, a `Message` enum was created, displayed in 4.10 and available on `message.rs`.

```

/// Enum that represents all the messages that will be sent
/// during the FROST protocol operations.
#[derive(Clone, Debug, PartialEq, Eq, Hash, Serialize,
    Deserialize)]
pub enum Message {
    /// Message utilized during the keygen round 1 phase.
    /// It represents the commitments and signature used to
    /// validate a user and create the aggregate public key.
    Broadcast {
        participant_id: u32,
        commitments: Vec<CompressedEdwardsY>,
        signature: (Scalar, Scalar),
    },

    /// Message that is sent during the keygen round 2 phase
    .
    /// It represents the secret sent from every participant
    /// to all others and it is used to calculate a
    /// participant's private key.
    SecretShare {
        sender_id: u32,
        receiver_id: u32,
        secret: Scalar,
    },
}

```

```

    /// Message that is sent during the signature phase.
    /// It is used by the main participant (SA) for others
    /// to verify the commitments chosen by the SA.
    PublicCommitment {
        participant_id: u32,
        di: CompressedEdwardsY,
        ei: CompressedEdwardsY,
        public_share: CompressedEdwardsY,
    },

    /// Message that is sent during the signature phase.
    /// It is used to compute the aggregate response and is
    /// sent by every participant to the SA.
    Response { sender_id: u32, value: Scalar },

    /// Message that is sent at the beginning of a FROST
    /// operation.
    /// It is used to do all the calculations needed for all
    /// the FROST operations.
    FrostState { participants: u32, threshold: u32 },

    /// Message that is sent at the begging of the FROST
    /// sign operation.
    /// It is used to atribute a temporary id to identify
    /// the participant as the operation is happening.
    Id(u32),

    /// Message that is sent at the end of an operation for
    /// the server to know when to close the socket.
    Completed(String),

    /// Message that is sent when something unexpected
    /// happens during an operation and it is used to close
    /// the socket when a problem occurs.
    Error(String),
}

```

Code Snippet 4.10: Typed Message Channel

In Rust, Structs can have methods related to them like classes in object-oriented languages. Leveraging that and the properties of `serde`'s Traits, `Message` implements functions for serialization and deserialization into JSON formatted Strings. These are shown in 4.11.

```

impl Message {
    /// Function that converts a 'Message' into a JSON
    /// formatted 'String'.
    pub fn to_json_string(&self) -> Result<String, Box<dyn

```

```

    Error + Send + Sync>> {
        Ok(serde_json::to_string(&self)?)
    }

    /// Function that converts a JSON formatted 'String'
    into a 'Message'.
    pub fn from_json_string(message: &str) -> Option<Message>
    {
        match serde_json::from_str::<Message>(&message) {
            Ok(message) => Some(message),
            Err(_) => None,
        }
    }
}

```

Code Snippet 4.11: Message's Methods

4.3.2 FROST

4.3.2.1 Key Generation

FROST's Key Generation process is utilized to produce the public aggregated key of the group and the individual private shares of each participant. It is built upon Pederson's Distributed Key Generation (DKG) [27]. What makes it unique is the protection against rogue key attacks by requiring each participant to prove knowledge of their private commitments, aborting on misbehavior.

The protocol is divided into two separate rounds and resides on `keygen.rs`.

The first round is a public commitment sharing phase (see 2.2.1), where participants compute their secret polynomial using `generate-polynomial`, their proof of knowledge signature with `compute-proof-of-knowledge` and their public commitment with `compute-public-commitments`. Each participant then has to broadcast their public commitment and challenge (`Message::Broadcast` variant shown in 4.10) and, after receiving commitments from all other participants, assert the proof's signature against the expected value with `verify-proofs`, aborting if it fails to verify at least one.

The second round is a secret share aggregation step where every participant calculates their secret share using `compute-secret-share` and for all other participants with `compute-share-for`. After receiving and verifying all the shares (`verify-share-validity`) each participant computes their own private key share, public key share and public aggregated key with `compute-private-key`, `compute-own-public-share` and `compute-group-public-key`. To en-

sure the public share that was generated is correct, it is compared to the result of `compute-others-verification-share`.

Finally, at the end of the protocol, the resulting 32-bit group public key is transformed into a Nano account address by calculating a 40-bit checksum and encoding the two using a custom lookup table to improve readability. This table is shown in 4.12. Finally, a prefix (*nano_*) is added and the address is complete.

```
/// Constant values to convert a public key into a Nano
    account address.
const ACCOUNT_LOOKUP: &[char] = &[
    '1', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', '
    d', 'e', 'f', 'g', 'h', 'i', 'j',
    'k', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'w', '
    x', 'y', 'z',
];
```

Code Snippet 4.12: Nano Account Address Lookup Table

To conclude, the Unified Modeling Language (UML) diagram 4.2 that follows shows the flow of the Key Generation protocol as a whole and how it will be applied in the application setting.

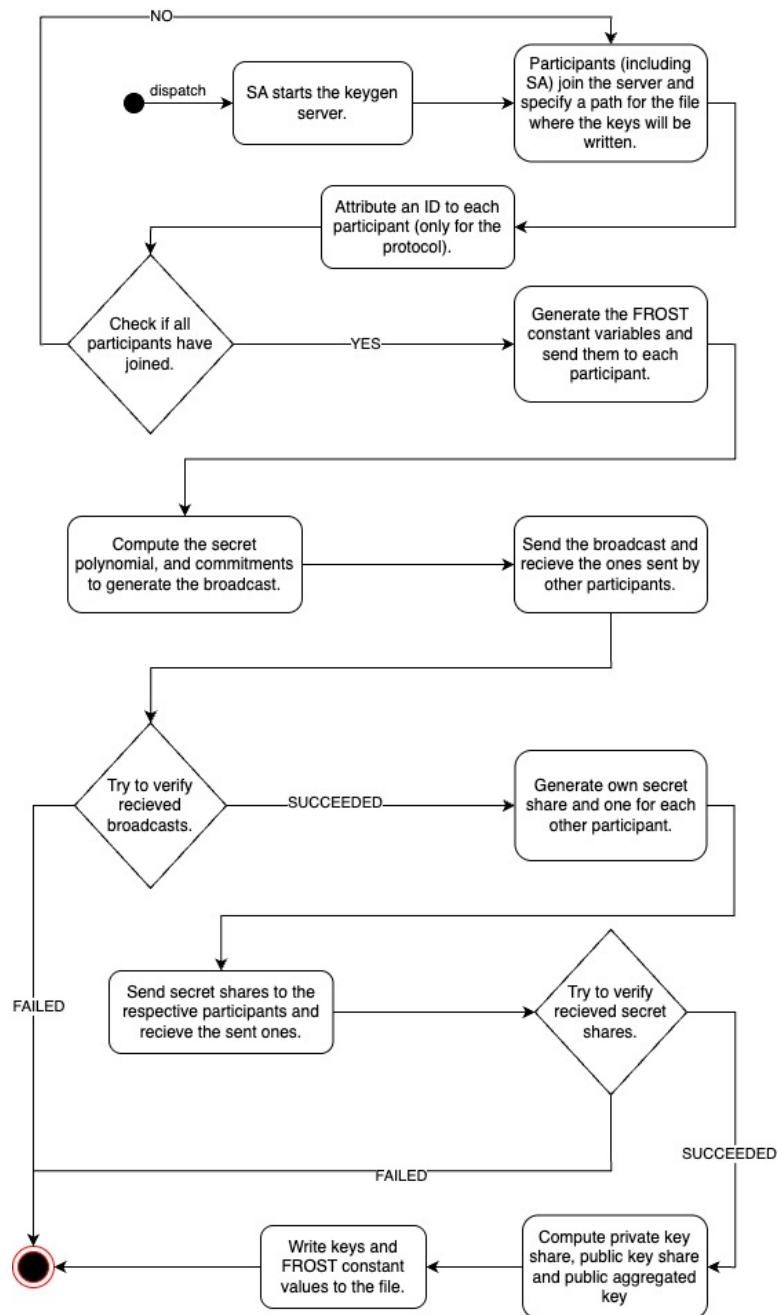


Figure 4.2: Key Generation Flow Diagram

4.3.2.2 Preprocess

Even though FROST allows for nonce and commitment preprocessing by having each participant publish a precomputed list, it was decided to perform the preprocessing phase at the start of the signing protocol. This approach was chosen because it requires minimal resources, introduces no significant delay, and guarantees that each nonce–commitment pair is used exactly once, thus avoiding potential management errors associated with maintaining pre-computed lists.

Nonces are generated by sampling a random `Scalar`, and the corresponding commitments are derived by multiplying this scalar with the elliptic curve’s base point like shown 4.13.

```
/// Function that generates a set of one-time-use nonces and
    commitments.
pub fn generate_nonces_and_commitments(
    rng: &mut OsRng,
) -> ((Scalar, Scalar), (CompressedEdwardsY,
    CompressedEdwardsY)) {
    let own_dij = Scalar::random(rng);
    let own_eij = Scalar::random(rng);
    let dij = own_dij * ED25519_BASEPOINT_POINT;
    let eij = own_eij * ED25519_BASEPOINT_POINT;
    ((own_dij, own_eij), (dij.compress(), eij.compress()))
}
```

Code Snippet 4.13: Preprocess Method

4.3.2.3 Sign

The Sign protocol enables participants to collaboratively and securely authorize transactions. To achieve this, the group must agree on a semi-trusted SA responsible for collecting the partial signatures and publishing the final transaction via Nano’s RPC interface. In order for the transaction to be valid under FROST, at least the defined threshold number of participants (set during the Key Generation phase) must sign it. The transaction itself is treated as a message input to the protocol and is represented as a `String`.

The protocol begins with each participant generating their nonces and commitments using `generate-nonces-and-commitments`, as described earlier. Participants then exchange their public commitments with each other. Next, each participant computes the aggregated group commitment and challenge, determines their Lagrange coefficient based on their identifier using `lagrange-coefficient`, and calculates their own response with `compute-own-response`. These responses are then sent to the SA.

Upon receiving the individual responses, the SA verifies each one using `verify-participant`. Once all responses have been validated successfully, the SA aggregates them. The final signature is constructed using the aggregated response and the group commitment.

For a clearer understanding, the process flow of this protocol is illustrated in the following UML diagram 4.3.

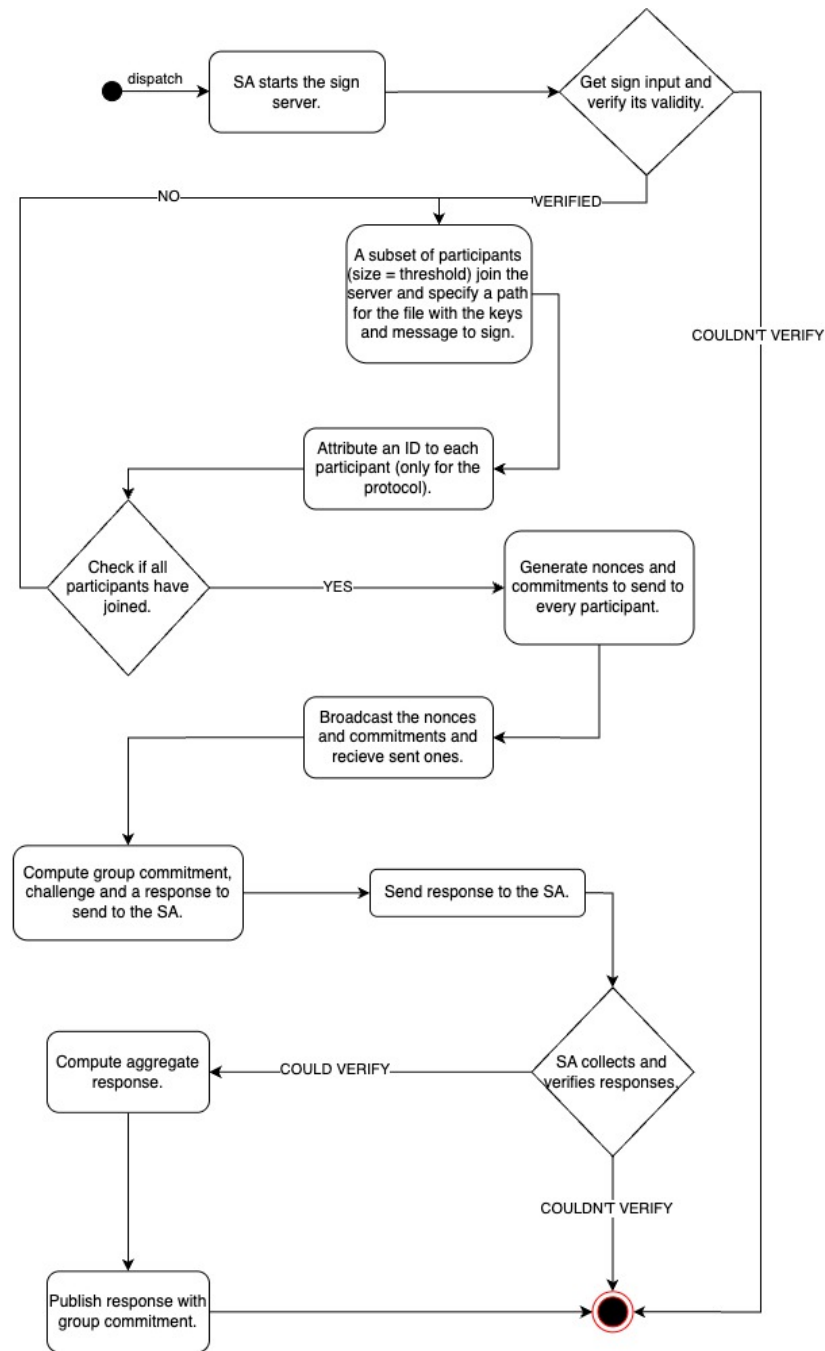


Figure 4.3: Sign Flow Diagram

4.3.3 Sockets

4.3.3.1 Server

Even though the protocol relies on a centralized server, instantiated by the SA, which other participants join as clients, its sole responsibility is to relay messages based on their type and specified arguments. The server maintains a `FrostServer` state 4.14 that keeps track of information about the participants who have joined the protocol, such as the transmitter mapped to each participant's address and identifier. Additionally, it stores the group's FROST parameters.

```
/// Struct that represents the server that will handle FROST
/// operations in real-time for multiple participant's
/// clients.
#[derive(Clone)]
pub struct FrostServer {
    /// The state that holds all the constants needed for
    /// the FROST computations.
    state: FrostState,
    /// The transmitter mapped to the socket address.
    by_addr: HashMap<SocketAddr, Tx>,
    /// The transmitter mapped to the participant's id.
    by_id: HashMap<u32, Tx>,
}
```

Code Snippet 4.14: FROST Server State

The `FrostServer` struct implements a send-message function that automatically handles sending messages according to their type to avoid mistakes. Messages intended for all participants, such as `Broadcast` or `PublicCommitment`, are sent using the `broadcast` method, which iterates over all connected participants except the sender and forwards the message. Messages intended for a single participant, such as `SecretShare` or `Response`, are sent using `send-to`, which directly targets a specific client by ID. These methods are shown in 4.15.

```
/// Function that handles the messages sent according to
/// type to avoid mistakes.
pub async fn send_message(
    &mut self,
    participant: &Participant,
    msg: Message,
) -> Result<(), Box<dyn Error + Send + Sync>> {
    match &msg {
        Message::Broadcast {
            signature: _,
            commitments: _,
        }
    }
```

```

        participant_id: _,
    }
    | Message::PublicCommitment {
        participant_id: _,
        di: _,
        ei: _,
        public_share: _,
    } => {
        self.broadcast(&participant.addr, msg).await;
    }
    Message::SecretShare {
        sender_id: _,
        receiver_id,
        secret: _,
    } => {
        self.send_to(*receiver_id, msg).await; // Should
            not fail.
    }
    Message::Response {
        sender_id: _,
        value: _,
    } => {
        self.send_to(1, msg).await; // defaulted to 1
            because SA should be the first one to enter.
    }
    _ => return Err("Tried to send an invalid message".
        into()),
}
Ok(())
}

/// Function that sends a message to all participants but
/// the one sending the message.
pub async fn broadcast(&mut self, sender: &SocketAddr,
    message: Message) {
    for participant in self.by_addr.iter_mut() {
        if &*participant.0 != sender {
            let _ = participant.1.send(message.clone());
        }
    }
}

/// Function that sends a message to a specific client in
/// the socket.
pub async fn send_to(&mut self, receiver: u32, message:
    Message) {
    if let Some(tx) = self.by_id.get(&receiver) {
        let _ = tx.send(message);
    }
}

```

```
}

```

Code Snippet 4.15: Send Message Method

The `handle` function manages each participant's connection to the server: it registers the participant, waits for all to join using a barrier (shown in 4.16), and then continuously relays messages between the server and the client.

```
// wait for all participants to join
barrier.wait().await;

```

Code Snippet 4.16: Barrier Wait

The `keygen-server` module defines and runs the FROST key generation server. It accepts a predefined number of participants, synchronizes them through a barrier, and then distributes the shared FROST state to all participants before completing.

The `sign-server` module defines and runs the FROST signing server. It accepts connections from a threshold number of participants required to sign, synchronizes them, and relays messages until the signing process is finished.

```
// init the socket
let address = format!("{}", ip, port);
let listener = TcpListener::bind(&address).await?;
// create the server instance
let server = Arc::new(Mutex::new(FrostServer::new(
    participants, threshold)));
logging::print(
    format!(
        "Keygen initialized on {}{}{}.",
        logging::YELLOW,
        address,
        logging::RESET
    )
    .as_str(),
);
// init id count and joining participants barrier
let mut count: u32 = 0;
let barrier = Arc::new(Barrier::new((participants + 1) as
    usize));
let mut handles = vec![];
while count < participants {
    // accept the participant's connection
    let (stream, addr) = listener.accept().await?;
    count += 1;
    let server = server.clone();
    let barrier = barrier.clone();
    // handle the participant in an isolated async thread
    let handle = tokio::spawn(async move {

```



```

        logging::print("Accepted a connection.");
        if let Err(e) = handle(count, barrier, server,
            stream, addr).await {
            eprintln!("{e}");
        }
    });
    handles.push(handle);
}
// Send the frost state.
{
    // Block until all have joined.
    barrier.wait().await;
    let server = server.lock().await;
    // send the shared 'FrostState' to the participant
    server.by_addr.values().into_iter().try_for_each(
        |tx| -> Result<(), Box<dyn Error + Send + Sync>> {
            tx.send(server.state.clone().to_message())?;
            Ok(())
        },
    )?;
}
// wait before closing the socket for messages that may be
// left unsent
for handle in handles {
    _ = handle.await;
}
logging::print("Successfully generated the key.");
Ok(())

```

Code Snippet 4.17: Run Method

4.3.3.2 Client

The client orchestrates the entire protocol, ensuring all participants exchange the correct information.

It defines a `FrostClient` state (4.18) that stores the variables needed throughout the protocol, such as the participant's own-id and the FROST parameters defined by the group. Additionally, it provides a receive-message method (4.19), which asynchronously waits for input from lines, parses the received data, and returns it if it can be successfully converted into a `Message` Enum variant.

```

/// Struct that holds the state for the FROST operations and
/// the id of the participant.
#[derive(Debug)]
pub struct FrostClient {

```

```

    /// State that holds all the constants needed for the
    /// FROST computation.
    pub state: FrostState,
    /// Id of the participant using the client.
    pub own_id: u32,
}

```

Code Snippet 4.18: FROST Client State

```

/// Function that receives a 'Message' of any type.
pub async fn receive_message(
    lines: &mut Framed<TcpStream, LinesCodec>,
) -> Result<Message, Box<dyn Error + Send + Sync>> {
    // Number of retries to account for missing messages.
    match lines.next().await {
        Some(Ok(line)) => {
            return Ok(Message::from_json_string(line.as_str
                ()))
                .expect(&format!("Failed to send message:
                    {}.", line)))
        }
        Some(Err(e)) => return Err(format!("Network Error: {
            e}").into()),
        None => return Err("Connection closed suddenly.".
            into()),
    }
}

```

Code Snippet 4.19: Receive Message Method

4.3.4 Nano

Finally, to enable the scheme to sign transactions that can be validated by Nano, there are a few important considerations.

4.3.4.1 Unsigned Block

The message that is signed by the protocol must be an UnsignedBlock, as illustrated in 4.20.

```

/// Struct that represents a block that has yet to be signed
/// by the FROST signature scheme.
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct UnsignedBlock {
    pub r#type: String,
    pub account: String,
    pub previous: String,
    pub representative: String,
}

```

```
pub balance: String,  
pub link: String,  
}
```

Code Snippet 4.20: Unsigned Block Structure

The `UnsignedBlock` structure represents a Nano block that has not yet been signed. Each field serves a specific purpose in defining the state and intent of the block within the Nano ledger:

- `type`: Indicates the type of block being created (send, receive, change, or state). This defines the block's behavior and how it updates the account's state.
- `account`: The Nano account address associated with the block. This identifies the owner of the block and links it to their account chain.
- `previous`: The hash of the previous block in the account's chain. It establishes continuity and ensures that each new block correctly extends the account's history.
- `representative`: Specifies the representative account chosen to vote on consensus. This field can be updated through a change block and is essential for Nano's delegated proof-of-stake mechanism.
- `balance`: The account balance after applying this block, represented as a string (typically in raw units). This enables the ledger to verify correctness of incoming and outgoing transactions.
- `link`: Serves multiple purposes depending on the block type. For example, it can represent the destination account in a send block or a source block hash in a receive block.

By ensuring that the `UnsignedBlock` is the message being signed, the resulting signature can be directly integrated into the Nano protocol, enabling the generation of valid, verifiable blocks that can be broadcast to and accepted by the network.

The block is constructed before initiating the signing process and is stored in the account's file until the protocol starts. Depending on the transaction type, the required arguments and block structure vary, which motivated the creation of dedicated methods: `create-open`, `create-receive`, and `create-send`.

The `create-open` method is used to generate a block that opens a new account. It takes the RPC state and the account address as arguments, retrieves

the first receivable block, and sets the initial balance using this amount. Since it is the first block, the previous field is set to "0", and the representative is set to the account's own address.

The create-receive method creates a block that allows an account to receive a pending transaction. It also receives the RPC state and the account address. This method obtains the current frontier (latest block hash) to set as the previous field and updates the account balance by adding the incoming amount from the receivable block.

Finally, the create-send method constructs a block to send funds to another account. It requires the RPC state, the sender's account address, the receiver's account address, and the amount to be sent (in Nano). The amount is converted to raw units, the balance is updated by subtracting this value, and the link field is set to the receiver's account key.

These methods ensure that each block is properly constructed for its specific transaction type, fully complying with Nano's block structure and making it ready to be signed and validated by the network.

4.3.4.2 Processing the Transaction

After computing the aggregated signature, the SA must convert the UnsignedBlock into a SignedBlock before publishing the transaction to Nano. This can be done using the methods shown in 4.21.

```
/// Function that signs an UnsignedBlock with a signature
/// and a proof of work.
pub fn to_signed_block(self, signature: &str, work: &str) ->
    SignedBlock {
    SignedBlock::new(
    self.previous,
    self.account,
    self.representative,
    self.balance,
    self.link,
    signature.to_string(),
    work.to_string(),
    )
}

/// Function that creates a new signed block with a valid
/// signature and work.
pub async fn create_signed_block(
    state: &rpc::RPCState,
    unsigned_block: UnsignedBlock,
    signature: &str,
    aggregate_public_key: &str,
```

```

key: &str,
) -> Result<SignedBlock, Box<dyn Error + Send + Sync>> {
let work = super::rpc::WorkGenerate::get_from_rpc(
&state,
match unsigned_block.previous.as_str() {
"0" => aggregate_public_key,
previous => previous,
},
&key,
)
.await?;
Ok(unsigned_block.to_signed_block(&signature, &work.work))
}

```

Code Snippet 4.21: Signed Block Structure

The `to-signed-block` method finalizes an `UnsignedBlock` by adding the aggregated signature and the computed proof of work. The `work` field is a nonce value required by Nano to satisfy its proof-of-work constraint, preventing spam and securing the network.

To simplify this process, the `create-signed-block` method can be used. This function not only attaches the signature but also automatically generates the necessary work by calling `work-generate` via RPC. Depending on whether the block is an open block (where `previous` is `"0"`) or a regular block, it chooses the correct hash base (either the aggregate public key or the previous block hash) to compute the work.

Finally, the block is published to Nano using the `process` action. If the result of the request is a hash, the transaction was successful.

4.4 Application

The repository for the application implementation can be found at <https://github.com/diogogomesaraujo/aokiji>.

To demonstrate the custom FROST signature method in practice, a desktop application was developed using `Dioxus`. This choice was driven by technical constraints: the FROST library relies on TCP sockets for inter-party communication during the signing protocol. Web browsers enforce strict security policies that block direct connections to arbitrary socket servers. As a result, a desktop application was the ideal solution, as it allows unrestricted access to system networking resources. This enables seamless TCP socket communication between signing participants, free from the limitations imposed by browser security models.

Despite changing from a web to a desktop environment, the building blocks developed in the pre-implementation phase were still applicable since Dioxus has consistent syntax and cross-platform support.

4.4.0.1 GUI

Before implementing the GUI with Dioxus it was important to establish a foundation and general direction in a more frictionless environment. As such, a mockup was developed and is accessible on <https://www.figma.com/design/uRPRVfRPf54z2KOAMlp8vC/Aokiji?node-id=0-1&t=4hH0T3cyCSMlhHjF-1>.

Dioxus, similar to frameworks like Vue and React, uses an HTML-inspired approach for building user interfaces. However, unlike JavaScript-based frameworks, Dioxus leverages Rust's powerful macro system. This allows developers to embed HTML-like blocks directly into Rust code using the `rsx!` macro, providing a seamless and expressive way to define UI components within a strongly typed language.

As an example, consider the header of the dashboard. In Dioxus, elements such as `div` and `button` can be assigned an `id` or `class` attribute, which allows them to be styled externally using Cascading Style Sheets (CSS). Additionally, elements can be styled directly using the `style` attribute, where CSS properties can be written inline. Interaction and dynamic behavior can be introduced through event handlers, such as `onclick`, which enable updating the application state when a user interacts with the interface.

```
rsx! {  
    div {  
        id: "header",  
        div {  
            style: "display: flex; align-items: center; gap:  
                12px;",  
            div {  
                class: "avatar",  
                img {  
                    src: "{AVATAR}",  
                    style: "width: 100%; height: 100%;  
                        object-fit: cover;"  
                }  
            }  
        }  
        div {  
            style: "display: flex; flex-direction:  
                column;",  
            div {  
                style: "display: flex; align-items:  
                    center; gap: 0px;",
```

```

    a {
      class: "nano-account",
      { app_state.read().nano_account.
        clone() }
    }
    button {
      class: "clipboard",
      onclick: copy_to_clipboard,
      style: "font-size: 20px; margin-left
        : -8px;",
      MaterialIcon { name: "content_copy"
        }
    }
  }
  div { id:"secondary", a { {
    // FROST parameters stored in the app
    state
    let frost_state = app_state.read().
      frost_state.clone();
    format!("{}", Participants", frost_state.
      participants)
  } } }
}
}
}
}

```

Code Snippet 4.22: Header Method Excerpt

To keep the application intuitive and lightweight, it is structured around two main sections: a home page, where participants can create or open an account, and a streamlined dashboard. The dashboard presents all relevant account information in an organized manner and offers clear options to start or join transaction sessions, as well as review account history like shown in 4.5. By focusing on simplicity and clarity, the interface ensures that essential actions remain accessible and easy to navigate.

4.4.0.2 Integrating the Library

After building the application's UI, the final step was to integrate the library so that participants can sign transactions and create shared accounts directly in-app. As shown previously, Dioxus supports calling asynchronous functions using futures, which allows the UI to remain responsive while running background tasks.

To exemplify, consider the StartTransaction section in the account dashboard. The open-socket-and-connect closure leverages use-future to establish an asynchronous execution context. Within this context, three

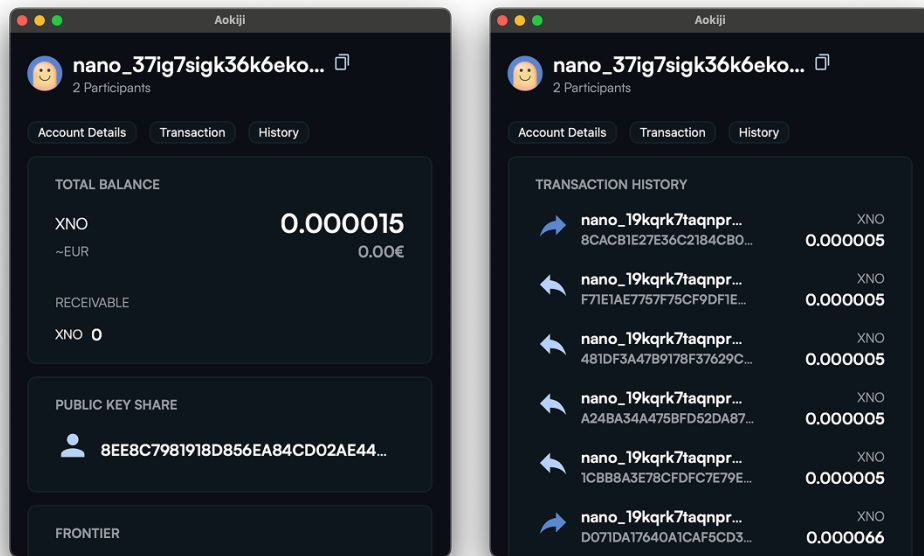


Figure 4.4: Print of the Dashboard

concurrent tasks are spawned via `tokio::spawn`: a server task that instantiates a message relay for coordinating participant communications, a client task that connects the current participant to the server after a 2-second delay using `tokio::time::sleep`, and a notification task that monitors both server and client completion via `tokio::join!` and updates the transaction state to `TransactionState::Successful`. This architecture ensures proper synchronization between the FROST protocol coordinator and participants while providing user feedback upon transaction completion.

```
// open the socket with the correct parameters
let server = tokio::spawn(async move {
    match frost_sig::server::sign_server::run(
        "localhost",
        PORT,
        state.participants,
        state.threshold,
    )
    .await
    {
        Ok(_) => {}
        Err(e) => {
            transaction_state.set(TransactionState::Error(e.
                to_string()));
        }
    }
});
```



```

        return;
    }
};
});

// connect to the socket that was opened
let client = tokio::spawn(async move {
    tokio::time::sleep(Duration::from_secs(2)).await;
    match frost_sig::client::sign_client::run(
        "localhost",
        PORT,
        &path,
        &config_file_path,
    )
    .await
    {
        Ok(_) => {}
        Err(e) => {
            transaction_state.set(TransactionState::Error(e.to_string()));
            return;
        }
    };
});

// after processing the transaction notify user
tokio::spawn(async move {
    let _ = tokio::join!(server, client);
    transaction_state.set(TransactionState::Successful);
});

```

Code Snippet 4.23: Open and Connect Socket Closure Excerpt

4.5 Testing

4.5.1 Protocol and Nano Tests

To verify the correct implementation of FROST and ensure compatibility with Nano's signature verification, extensive manual testing was performed.

In Rust, a test is a function annotated with the `#[test]` attribute. It uses assertions to check that actual results match expected values and, if an assertion fails, the test aborts, otherwise it passes. To run all tests the command `cargo test` can be utilized.

Let's take `test-keygen-and-sign` as an example. Removing the asynchronous environment and running the test in a controlled and predictable

```

diogoaraujo@Mac ~/Desktop/projects/frost-sig
> $ cargo test test_keygen_and_sign -- --nocapture
   Compiling frost-sig v0.0.1 (/Users/diogoaraujo/Desktop/projects/frost-sig)
   Finished `test` profile [unoptimized + debuginfo] target(s) in 1.40s
   Running unittests src/lib.rs (target/debug/deps/frost_sig-016c718c04af8305)

running 1 test
Aggregate Public Key: nano_18dos1h6mhppdrmbbj6io3u1emsqp1mr7remztxfi14izhzbpra8u87eku1i
test test::test_keygen_and_sign ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished in 0.04s

```

Figure 4.5: Test Run Demonstration

setting was crucial for debugging and identifying errors throughout the development process, making these issues reproducible and eliminating the overhead associated with distributed systems. This test not only performs all the FROST validations described in the library implementation section but also uses ed25519-dalek-blake2b to ensure that the resulting signature is valid on the curve utilized by Nano.

```

// sa computes signature
let (signature, _) = computed_response_to_signature(&
    aggregate_response, &group_commitment)?;

// Verify the signature
{
let verifying_key = PublicKey::from_bytes(group_public_key.
    as_bytes())
.expect("Couldn't create the public key!");
verifying_key
.verify(&hex::decode(&message)?, &signature)
.expect("Couldn't verify the signature with the public key
    !");
}

```

Code Snippet 4.24: Key Generation and Sign Testing Excerpt

In 4.24, the final signature is first derived from the aggregate responses and the group commitment. Then, it is verified using the reconstructed group public key to ensure correctness. This validates that the signing process is correctly implemented and compatible with Nano's cryptographic requirements. To validate the signature, the aggregated group public key is first converted into a `PublicKey` structure. The `verify` method is then called, which checks whether the signature correctly corresponds to the provided message and public key. This step ensures that the signature's response and the group commitment match the expected values, ultimately confirming the correctness and integrity of the signing process.

4.5.2 Socket Test

The FROST implementation was evaluated using a distributed client-server architecture. The interface is operated via the following command-line structure:

```
cargo run -- <mode> <operation> [parameters]
```

As an example, consider a scenario where two participants collaboratively create a Nano account with a threshold of two.

As illustrated in 4.6, the server correctly waits for both participants to connect before initiating the protocol. Once connected, each client successfully computes and derives the shared Nano account, as depicted in 4.7 and 4.8.

```
diogoaraujo@MacBook-Air-de-Diogo ~/Desktop/projects/frost-sig [0:23:48]
> $ cargo run -- server keygen 2 2 [±master ●]
   Compiling frost-sig v0.0.1 (/Users/diogoaraujo/Desktop/projects/frost-sig)
   Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.75s
   Running `target/debug/frost-sig server keygen 2 2`
Frost Server: Keygen initialized on localhost:6705.
Frost Server: Accepted a connection.
Frost Server: Accepted a connection.
Frost Server: Successfully generated the key.
```

Figure 4.6: Server running and waiting for participants

```
diogoaraujo@MacBook-Air-de-Diogo ~/Desktop/projects/frost-sig [0:52:37]
> $ cargo run -- client keygen account1.json [±master ●]
   Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.23s
   Running `target/debug/frost-sig client keygen account1.json`
Frost Client: Connected to the server successfully.
Frost Client: This is the group's nano account nano_1co1ce4df1wr6r7m5d19o33mgtcw8nnqesjh
imr4sock3rxuksuuz71z91p8.
Frost Client: The keygen process information was stored in account1.json.
```

Figure 4.7: First participant client successfully computing the shared Nano account

```
diogoaraujo@MacBook-Air-de-Diogo ~/Desktop/projects/frost-sig [0:53:00]
> $ cargo run -- client keygen account2.json [±master ●]
   Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.05s
   Running `target/debug/frost-sig client keygen account2.json`
Frost Client: Connected to the server successfully.
Frost Client: This is the group's nano account nano_1co1ce4df1wr6r7m5d19o33mgtcw8nnqesjh
imr4sock3rxuksuuz71z91p8.
Frost Client: The keygen process information was stored in account2.json.
```

Figure 4.8: Second participant client successfully computing the shared Nano account

4.6 Conclusion

In conclusion, this chapter presented the successful implementation of the FROST library, which generates signatures that are fully compatible with Nano and can be used for blockchain transactions.

Chapter

5

Conclusions and Future Work

5.1 Main Conclusions

This project successfully implemented a custom signature scheme that rigorously adheres to the FROST protocol and produces signatures fully compatible with validation on the Nano blockchain. Beyond the library implementation, it also integrated a user-friendly group wallet application, empowering users to create genuine Nano shared accounts capable of performing real transactions securely and efficiently.

Undertaking this work presented a uniquely demanding and rewarding challenge, compelling me to grow significantly as an engineer while deepening my expertise in Rust, modern cryptography, and distributed systems. More importantly, it reinforced my ability to transform an abstract, conceptual idea into a robust, concrete, and functional system. The journey from theoretical design to a fully operational implementation has been a testament to the power of disciplined engineering and has further strengthened my commitment to building secure and reliable cryptographic applications.

5.2 Future Work

Even though the project successfully implements threshold transactions with FROST for a real cryptocurrency Nano that are validated and follow the protocol's security requirements, it is still a prototype, lacking features to make it applicable in *real world* environments and for non-technical users. Some of these characteristics and future objectives are listed as follows:

- **TCP Sockets Alternative:** The sockets used in this project have a major

limitation: due to Network Address Translation (NAT), participants outside the local network cannot directly reach the server's IP address. A common solution is port forwarding, which exposes the necessary port on the router to allow external connections to the server, but is not ideal. A solution would be to use a more complex peer-to-peer connection or a web socket implementation.

- **Publishing the Library:** The core building blocks are generic enough to support any cryptocurrency and correctly implement FROST. The library could be further improved to accept different hash functions (such as Blake2b or SHA-512) and published on <https://crates.io/>, as it is already open-source and there are few FROST implementations available in Rust.

Bibliography

- [1] Rust By Example. Crate. [Online] <https://doc.rust-lang.org/rust-by-example/crates.html>.
- [2] Tanja Lange Peter Schwabe Daniel J. Bernstein, Niels Duif and Bo-Yin Yang. Ed25519: high-speed high-security signatures. [Online] <https://ed25519.cr.yp.to/>.
- [3] Andrew W Appel. Garbage collection. *Topics in Advanced Language Implementation*, pages 89–100, 1991.
- [4] Snyc. Use after free. [Online] <https://learn.snyc.io/lesson/use-after-free/?ecosystem=cpp>.
- [5] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [6] G. R. Blakley. Safeguarding cryptographic keys. *1979 International Workshop on Managing Requirements Knowledge (MARK)*, pages 313–318, 1979.
- [7] E. Karnin, J. Greene, and M. Hellman. On secret sharing systems. *IEEE Trans. Inf. Theor.*, 29(1):35–41, September 2006.
- [8] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science, SFCS '87*, page 427–438, USA, 1987. IEEE Computer Society.
- [9] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.
- [10] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
- [11] Chelsea Komlo and Ian Goldberg. Frost: Flexible round-optimized schnorr threshold signatures. In *Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-*

- 23, 2020, *Revised Selected Papers*, page 34–65, Berlin, Heidelberg, 2020. Springer-Verlag.
- [12] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.
- [13] Rust Foundation. Rust Programming Language. [Online] <https://www.rust-lang.org/>.
- [14] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.
- [15] Thomas Heartman. Understanding the Rust borrow checker, 2024. [Online] <https://blog.logrocket.com/introducing-rust-borrow-checker/#garbage-collection-vs-manual-memory-allocation-vs-borrow-checker>.
- [16] Vivek Kapoor, Vivek Sonny Abraham, and Ramesh Singh. Elliptic curve cryptography. *Ubiquity*, 2008(May):1–8, 2008.
- [17] Rahul Awati and Andrew Froehlich. What is elliptical curve cryptography (ECC)? [Online] <https://www.techtarget.com/searchsecurity/definition/elliptical-curve-cryptography>.
- [18] Dalek Cryptography. curve25519-dalek. [Online] https://docs.rs/curve25519-dalek/latest/curve25519_dalek/.
- [19] Blake2. [Online] <https://docs.rs/blake2/latest/blake2/>.
- [20] Tokio. Tutorial. [Online] <https://tokio.rs/tokio/tutorial>.
- [21] Sean McArthur. reqwest. [Online] <https://github.com/seanmonstar/request>.
- [22] Dioxus Labs. What is Dioxus? [Online] <https://dioxuslabs.com/learn/0.6/#what-is-dioxus>.
- [23] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 185–200, 2017.
- [24] Serde. [Online] <https://serde.rs/>.

-
- [25] Brooke Becher. What Are Blockchain Nodes and How Do They Work? [Online] <https://builtin.com/blockchain/blockchain-node>.
 - [26] IBM. Remote Procedure Call. [Online] <https://www.ibm.com/docs/en/aix/7.2.0?topic=concepts-remote-procedure-call>.
 - [27] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10*, pages 522–526. Springer, 1991.