Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Javier García

Threshold Signatures for FIDO Authentication

Master's thesis in Master's thesis in Digital Infrastructure and Cyber Security
Supervisor: Tjerand Silde
Co-supervisor: Andrés Marín, Trond Peder Hagen, and Magnus Ringerud

July 2025

NTNU
Norwegian University of
Science and Technology

PONE
BIOMETRICS

Javier García

Threshold Signatures for FIDO Authentication

Master's thesis in Master's thesis in Digital Infrastructure and Cyber Security
Supervisor: Tjerand Silde
Co-supervisor: Andrés Marín, Trond Peder Hagen, and Magnus Ringerud
July 2025

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Threshold Signatures for FIDO Authentication

Javier García

Submission date:	July 2025
Main supervisor:	Tjerand Silde (NTNU)
Co-supervisors:	Andrés Marín (ETSIT UPM),
	Trond Peder Hagen (PONE Biometrics), and
	Magnus Ringerud (PONE Biometrics)

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Problem description

Most online services require a username and password to authenticate users when logging in. This is usually handled in the following way:

- The user creates a fresh TLS connection with the server.
- The user sends their password to the server over TLS.
- The server checks if it matches the password in the database.

This approach has several weaknesses, such as the server seeing all passwords in plaintext sent from the user whenever someone tries to log in, and many users are victims of phishing attacks, in which an attacker can learn the password. A more secure mechanism is to use digital signatures for authentication, where a public key is stored on the server together with the username, and the client needs to cryptographically sign a fresh challenge message every time they want to log in. This is what is done in the FIDO protocol.

To use the FIDO protocol, you must store the signing key on a trusted device, for example, on your laptop or a specialised device, such as an OFFPAD. These devices may then be protected with biometrics or a local password to protect the signing key, however, it is possible to further improve the security and robustness of FIDO by using threshold signatures. This means that the signing key is split into several pieces that can be distributed to several devices, and then some of the devices collaborate to produce a fresh signature every time the user wants to log in. This allows the user to log in even if some of the devices are stolen or lost, and it puts a higher burden on the attacker to steal more than one device to be able to impersonate the user.

The goal of this project is to analyse and implement threshold signatures for the FIDO protocol using the OFFPAD.

Approved: 2025-02-18 – Tjerand Silde (NTNU) (Main supervisor)

Abstract

Our work aims to enhance the security of digital authentication through implementing threshold signatures as a part of the OFFPAD authentication mechanisms. The OFFPAD is a high-security biometric authentication gadget developed by PONE Biometrics that offers fingerprint or PIN authentication while otherwise being offline to avoid exposure to potential cyber threats. Currently, it supports the RSA and ECDSA signature schemes under the FIDO2 standard, which enables services to be used securely without depending on passwords.

In this project, we successfully integrate threshold cryptography into the OFFPAD's authentication process while maintaining its ease and efficiency thanks to the use of the FROST (Flexible Round-Optimized Schnorr Threshold Signatures) protocol, which uses Schnorr signatures. Using threshold signature schemes, the signing key is split across different devices so that no device has the whole key, avoiding single points of failure. In this work we evaluate various schemes to find the best fit for the OFFPAD's hardware, balancing security and usability by implementing a 2-out-of-3 threshold signature scheme.

Furthermore, a communication model is implemented to manage the different devices involved in threshold signature generation in a highly efficient way to retain its security without adding too much latency and complexity. This model makes use of a coordinator to ease the communication between participating devices in the implemented threshold signature scheme.

To sum up, we showcase the security of the authentication protocol in the OFFPAD using threshold cryptography while maintaining its strong points: efficiency, simplicity, and ease of use.

Sammendrag

Målet med dette arbeidet er å forbedre sikkerheten ved digital autentisering gjennom implementering av såkalte terskelsignaturer som en del av OFFPAD sin autentiseringsmekanisme. OFFPAD er en autentiseringsenhet med høy sikkerhet utviklet av Pone Biometrics som tilbyr autentisering ved hjelp av fingeravtrykk eller PIN, mens den resten av tiden er frakoblet andre enheter for å unngå eksponering mot potensielle cyberangrep. Foreløpig støtter den RSA- og ECDSA-signaturer som en del av FIDO2-standarden, som gjør det mulig å bruke tjenester sikkert uten passord.

I dette prosjektet integrerer vi terskelsignaturer som en del av OFFPAD sin autentiseringsprosess, samtidig som vi opprettholder dens effektivitet ved å bruke FROST-rammeverket, som anvender Schnorr-signaturer. Ved hjelp av terskelsignaturer er signeringsnøkkelen delt over forskjellige enheter, slik at ingen enheter har hele nøkkelen, og vi unngår at nøkkelen lekker dersom noen får tilgang til en enhet. I dette arbeidet evaluerer vi forskjellige signaturer for å finne den som passer best for OFFPAD, og vi balanserer sikkerhet og brukervennlighet ved å implementere en 2-av-3 terskelsignatur.

Videre implementerte vi en kommunikasjonsmodell for å administrere de forskjellige enhetene som er involvert i terskelsignaturer på en effektiv måte uten å legge til for mye forsinkelse eller kompleksitet. Denne modellen benytter seg av en sentral koordinator for å forenkle kommunikasjonen mellom deltakende enheter.

For å oppsummere, så viser vi hvordan sikkerheten til autentiseringsprotokollen i OFFPAD kan forbedres ved bruk av terskelsignaturer mens vi opprettholder effektivitet, enkelhet, og brukervennlighet.

Acknowledgements

This project would not have been possible without the support and guidance of many people.

First, I want to express my sincere gratitude to my main supervisor, Tjerand Silde, whose knowledge and dedication have been an invaluable help during this thesis. I can say I have been very lucky to be supervised by him and also learn from him.

I would like to thank my co-supervisor, Andrés Marín from ETSIT UPM, whose support and guidance have been essential to my work. Despite the distance, he was always available to provide help and assist me, demonstrating a dedication for which I am deeply grateful.

I also want to express my immense gratitude to the entire team at PONE Biometrics, especially Trond Peder Hagen, Magnus Ringerud, Kacper Wysocki, and Sigurhjörtur Snorrason, who welcomed me into their offices in Oslo. Working with such dedicated professionals has been both educational and inspiring.

To my friends, both those who have supported me from back in Spain and the incredible people I have met during this wonderful experience of studying in Norway, I am deeply grateful. The friendships I have made during all this journey have been priceless for me, and I want to thank you for your support.

Finally, and most importantly, I would like to express my deepest gratitude to my family. To my parents, thank you for your endless time, dedication, and sacrifices. I would not be where I am without you. You have shaped the person I am today, and I am proud of that. To my sisters, we have travelled this journey of life together, and I can say with confidence that I am the proudest brother that could exist. Your support has meant the world to me. To my aunt and uncles, thank you for your unconditional support and encouragement throughout this journey. To my grandparents, you are my inspiration in life, and I work every day to become even a fraction of the exceptional people you are.

This thesis is not only a result of my efforts, but also of the collective support, wisdom, and encouragement of all these outstanding people.

Contents

List of Figures

Listings

List of Acronyms

API Application Programming Interface.

BLE Bluetooth Low Energy.

CTAP Client to Authenticator Protocol.

DKG Distributed Key Generation.

DLP Discrete Logarithm Problem.

ECC Elliptic Curve Cryptography.

ECDH Elliptic Curve Diffie-Hellma.

ECDLP Elliptic Curve Discrete Logarithm Problem.

ECDSA Elliptic Curve Digital Signature Algorithm.

FIDO Fast Identity Online.

FIFO First In, First Out.

FROST Flexible Round-Optimized Schnorr Threshold.

HID Human Interface Device.

MCU Microcontroller Unit.

OFFPAD Offline Personal Authentication Device.

PIN Personal Identification Number.

RP Relying Party.

RSA Rivest-Shamir-Adleman.

TLS	Transport Layer Security.
UART	Universal Asynchronous Receiver-Transmitter.
VOLE	Vector Oblivious Linear Evaluation.
VSS	Verifiable Secret Sharing.
W3C	World Wide Web Consortium.
WebAuthn	W3C's Web Authentication.

Chapter 1 Introduction

In today's world, the growing number of digital services requires users to manage many passwords. This often leads to either choosing weak, easy-to-remember passwords or struggling to remember stronger, complex ones, leaving users vulnerable to cyber threats. To solve this, the FIDO Alliance introduced the FIDO Authentication standards [FID22], allowing easy and secure login without using traditional passwords.

The OFFPAD [PON24], a biometric authentication device developed by PONE Biometrics [PON18], provides fingerprint or PIN authentication while remaining offline to avoid exposure to potential cyber threats. It implements FIDO2, the latest standard developed by the FIDO Alliance, which provides seamless, secure, and easy authentication across many services. The goal of the project is to reinforce this by incorporating threshold signatures into its authentication mechanisms.

Distributing the authentication process across multiple devices provides better security with only a slight trade-off in efficiency. By spreading the key across devices, security is enhanced as an attacker is required to compromise multiple devices rather than just one to generate valid signatures, significantly increasing the attack complexity and cost. Additionally, with threshold signatures, there is no single point of failure, making the system more robust, as users can still log in even if a device is lost. However, efficiency may suffer since multiple devices need to be online, increasing complexity compared to using a single device. Depending on the threshold T, which defines how many devices out of the total N are required for signing, different levels of security and efficiency can be achieved. More devices increase security and robustness but decrease efficiency, and vice versa.

With all this in mind, we present in this thesis an implementation of threshold signatures into the OFFPAD authentication framework. We established the motivation, objectives, research questions, and some of the related works covered for this work in a previous project [Gar25b]. They remained the same throughout the project, with only slight changes made during development.

1.1 Objectives

Our main contribution in this project is integrating threshold signatures into the OFFPAD's authentication mechanism, enhancing security while maintaining compatibility with FIDO2 authenticators and settings. This involves incorporating threshold cryptography within the FIDO2 protocol. The OFFPAD will ensure fast, user-friendly authentication without imposing high computational or administrative overhead. The challenge is to add security measures without compromising the efficiency and simplicity that define the FIDO ecosystem.

Therefore, a key aspect of this work is selecting the most suitable threshold signature scheme and implementation. This involves evaluating various cryptographic schemes based on their security properties, computational efficiency, and compatibility with the OFFPAD's hardware and the FIDO framework. Since threshold signatures require multiple devices or entities to participate in signing, we will work to identify the optimal number of devices to balance security and usability.

Furthermore, in this project we will also design a communication model to coordinate the devices involved in the threshold signature. The approach selected considers a model where OFFPAD devices send their signature shares to a computer acting as a combiner, without direct communication between devices. The challenge lies in designing a secure communication model where most of the computational work rests with the computer, ensuring efficient coordination while maintaining the security properties of threshold signature schemes.

Finally, communication should balance security and efficiency by minimizing the complexity and frequency of exchanges between devices.

1.2 Research questions

From the motivation and objectives explained, we can extract the following research questions that we will try to answer during the project:

1. What is the best communication model for threshold signatures?
2. What is the total number of participants N, and the threshold T of devices that should participate to achieve a good balance between security and efficiency for the OFFPAD?
3. What is the best threshold signature scheme for the OFFPAD? And the best FIDO2-compliant?
4. What are the challenges in implementing the selected threshold signature scheme, and how can they be solved?

1.3 Our contributions

In this thesis, we contribute practical implementation work that advances the integration of threshold signatures in authentication systems with embedded devices. Our key contributions can be summarized as:

- **Threshold signature integration for the OFFPAD:** We present an implementation of threshold signatures working for the OFFPAD, where Schnorr signatures [Sch91] are used to implement the FROST protocol [KG20] in the authentication process, improving its security. With this implementation, we successfully perform threshold signatures in embedded devices, enabling them to execute cryptographic operations such as the computation of signature shares.
- **Communication model:** We design and implement an efficient communication model where OFFPAD devices exchange data thanks to a computer acting as a coordinator, which receives the data from all the devices and then redistributes it to each corresponding device.
- **Efficient threshold signing architecture:** Based on the work from Connolly, Komlo, Goldberg, and Wood [CKGW24], we implement an efficient way to carry out a threshold signature scheme thanks to the use of a trusted dealer in the key generation and a signature aggregator in the signing. These two roles reduce the burden on the embedded devices, making the process more efficient.

1.4 Related work

To better understand this work and its contribution, in this section we review existing research and implementations in the field of threshold signature schemes.

1.4.1 FROST: original protocol design

The Flexible Round-Optimized Schnorr Threshold Signatures (FROST) protocol was originally designed by Komlo and Goldberg [KG20], a threshold signature scheme based on Schnorr signatures [Sch91]. It allows a group of signers to jointly produce a valid Schnorr signature, all without any signer revealing their secret share or the full reconstruction of the secret key.

FROST maximizes the efficiency of rounds required for signing without sacrificing security. It minimizes network overhead compared to other existing threshold signature schemes and offers a new binding mechanism to prevent forgery attacks. Its main advantage is its intentional trade-off of some robustness for higher round efficiency.

The protocol can be used in two different modes: a preprocessing-based single-round signing protocol and a two-round signing protocol. In the preprocessing mode, the participants preprocess the nonces and commitments required, enabling a single-round signing. In contrast, the two-round mode eliminates this preprocessing by requiring that the participants reveal commitments in the first round.

1.4.2 FROST: enhanced security framework

Expanding on the original FROST protocol described above (Section 1.4.1), Crites, Komlo, and Maller present in their work a security framework for multi-party and threshold signature schemes based on the discrete logarithm problem [CKM21]. They address the limitations of the original FROST protocol by incorporating new techniques to ensure security in the random oracle model. Additionally, they introduce a modular security framework, enabling the analysis of the complex interactions that the threshold signature schemes normally face, hence avoiding possible security weaknesses that could appear during signing operations.

One of the main contributions of this work is the introduction of a revised version of the initial FROST protocol, with improved security and performance guarantees. It demonstrates that Schnorr signatures are sufficient proofs of possession in the algebraic group model without any tightness loss, strengthening the theoretical foundation of threshold signatures. Moreover, the proof framework not only improves the security analysis of FROST but also creates a reusable architecture that can be applied to many threshold and multi-signature protocols.

1.4.3 FROST: IETF standardization

The FROST protocol was further improved and standardized by Connolly, Komlo, Goldberg, and Wood [CKGW24], who present in their work an IETF (Internet Engineering Task Force) [Int25] standard from the protocol originally designed by Komlo and Goldberg (Section 1.4.1) and the security enhancements developed by Crites, Komlo, and Maller (Section 1.4.2).

The implementation they present uses two rounds of communication and employs Shamir's secret sharing [Sha79] to distribute the key among participants. This two-round communication implementation reduces network overhead compared to multi-round protocols, enhancing the efficiency when generating signatures, and the use of Shamir's secret sharing provides security in the scheme, ensuring that the secret key is never fully reconstructed at a single point.

The scheme in this case consists of, first, a key generation process, which depends on a trusted dealer, who generates a secret key uniformly at random and uses verifiable secret sharing to create secret shares of it, enabling participants to verify if their share is correct. With all participants holding a secret share, the signing of a message is performed with two rounds of communication, where the second round has a device with the role of coordinator, whose responsibility is to ease the communication between participants and gather all the signature shares to compute the final signature of the message.

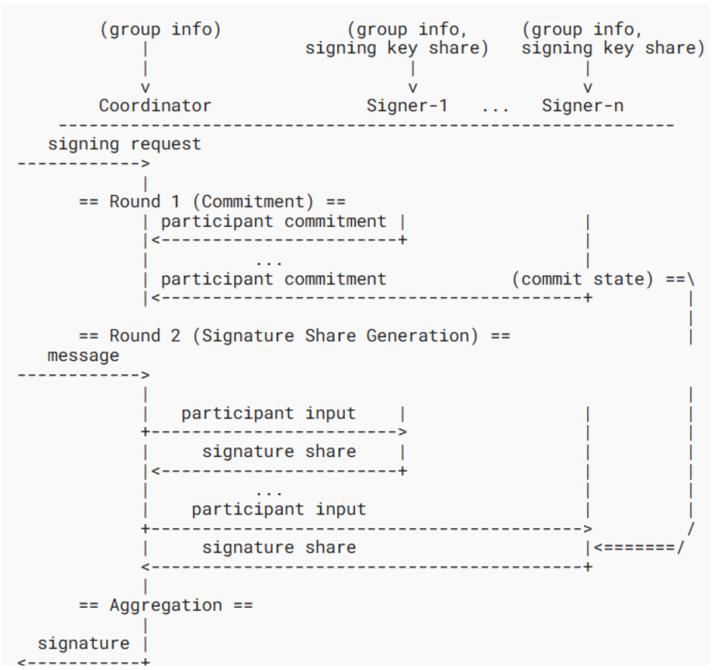


Figure 1.1: FROST protocol overview [CKGW24].

1.4.4 Fast multiparty threshold ECDSA

Gennaro and Goldfeder presented a threshold signature scheme for ECDSA [JMV01], which addresses important weaknesses in previous threshold ECDSA implementations [GG18]. This implementation is the first threshold ECDSA protocol allowing multiparty signing for any $T \leq N$ with an efficient key generation scheme without the use of a dealer. Unlike previous approaches that relied on a trusted dealer or the use of expensive distributed RSA key generation, this protocol achieves a significant improvement in both communication and computation efficiency. Moreover, the scheme has been proven secure against a malicious adversary, and even against a malicious majority.

The protocol covers two main phases: a distributed key generation phase and a signature generation phase. In the key generation phase, Feldman's verifiable secret sharing [Fel87] is used to generate shares of a private ECDSA key, avoiding the role of a trusted dealer. Each participant has to generate their secret contribution independently and use verifiable secret sharing to share it, and summing all the contributions results in the full private key.

The signature phase incorporates a way of transforming multiplicative shares into additive shares (MtA). This is done by first applying Shamir's secret sharing [Sha79] on the private key, and subsequently creating additive shares for all values required for ECDSA signing. Using homomorphic encryption, in this case, Paillier encryption [Pai99], enables the correlation between multiplicative and additive secret sharing schemes, making possible the calculation of products of secret values required by ECDSA.

1.4.5 Threshold ECDSA with identifiable abort

Gennaro and Goldfeder introduced a highly efficient scheme with identifiable abort mechanisms to address critical limitations in existing threshold ECDSA protocols [GG20]. Their work addresses two key defects in previous protocols: the absence of a mechanism to detect faulty participants and abort, and the need for several rounds of communication in the signing process.

The protocol introduced by them had the big innovation of incorporating a non-interactive online phase, which optimizes the signing process to a single round of communication, with each party performing one scalar multiplication. This is possible thanks to an offline preprocessing phase that is independent of the message to sign. These improvements enable asynchronous participation, which makes this implementation especially useful for practical deployments in real-world scenarios.

1.4.6 Three-Round ECDSA threshold signatures scheme

In Doerner, Kondi, Lee, and Shelat's work, they present a three-round threshold ECDSA signing protocol that is secure even against a malicious majority [DKLsa24]. The protocol enables a group of signers to generate valid ECDSA signatures [JMV01] while keeping each secret share confidential and without the full reconstruction of the secret key.

The protocol requires only three rounds of communication and uses Vector Oblivious Linear Evaluation (VOLE) [ADI+17] for secure multiplication, which supposes a big improvement in round complexity compared to existing threshold ECDSA protocols. It works by producing an intermediate representation of ECDSA signatures, which was first proposed by Lindell and Nof [LN18], allowing some nonlinear operations to be executed at the same time.

The signing process is divided into three main stages. In the first stage, each party submits a nonce contribution and starts VOLE instances with the other parties. The second stage involves decommitting nonce values, inputting these into the VOLE protocols, and performing tests to detect any misbehaviour. The third stage completes the signature construction with the communication of the signature shares and computing the final signature from them.

One significant improvement is the complete removal of zero-knowledge proofs from the protocol. Instead of relying on proof systems, this approach uses simple statistical consistency checks that are easily verifiable in the elliptic curve group. This method provides security while improving previous threshold ECDSA protocols.

1.5 Outline

This thesis consists of the following six chapters:

Chapter 2: Background introduces the background knowledge that is required for this thesis. It provides an overview of the OFFPAD, the FIDO2 standard and ecosystem, and covers important mathematical and cryptographic concepts such as elliptic curves, RSA, ECDSA and Schnorr signature schemes.

Chapter 3: Threshold signatures explains the core concept of this thesis, threshold signature schemes. It includes descriptions of secret sharing techniques such as Shamir's and Feldman's schemes and presents some implementations of threshold signature schemes using Schnorr and ECDSA signatures.

Chapter 4: Proposed solution outlines the decisions made to integrate threshold signatures into the OFFPAD's authentication mechanism. It justifies the choice of the selected threshold signature scheme and communication model, detailing the communication strategies selected for the communication between devices, the setup for key generation and distribution, and the process of share aggregation.

Chapter 5: Methodology describes the methodology followed in the different phases carried out during the project and the implementation of the proposed solution. This includes the full process of the selected threshold signatures schemes, tools, libraries, and hardware used. The chapter includes code snippets to illustrate key aspects of the implementation.

Chapter 6: Results and discussion summarizes the experimental results obtained with the application of threshold signatures in the context of the OFF-PAD's infrastructure. It evaluates and discusses the obtained results using different performance metrics, as well as discusses limitations during the implementation stage.

Chapter 7: Conclusions and future work offers a detailed overview of the main contributions and results of this project, related to the research questions discussed regarding the optimal threshold signature scheme, communication model and threshold parameters for the OFFPAD. It also discusses the limitations of the current implementation and future directions for research to improve it.

Chapter 2 Background

We reviewed the technical background to understand several key concepts in the fields of authentication and cryptographic security in the project preceding this thesis [Gar25b]. We include this background below with some additions in the fields of authentication and cryptographic security.

2.1 The OFFPAD

First of all, PONE Biometrics [PON18] is a cybersecurity company which aims to improve authentication on hardware authenticators with plans to use threshold signatures in the future, focusing on secure, well-designed, and user-friendly solutions. Their strategy provides scalable, advanced security for enterprises, public sectors, healthcare, and defense. Their first product, the OFFPAD (Offline Personal Authentication Device) [PON24], is a smart card-sized biometric authentication device offering high security with easy deployment and integration into existing IT infrastructures, being a biometric-based alternative to traditional passwords, using fingerprint or PIN authentication. Designed to stay offline, it remains powered down until activated by the user for authentication, significantly reducing its exposure to online threats, minimizing the attack surface.

Fundamentally, it is more secure compared to smartphones, which are always on and vulnerable due to third-party apps. The OFFPAD conforms to the authentication standard FIDO2, further enhancing security and improving the way users sign in to online services.

2.2 FIDO ecosystem

FIDO2 (Fast IDentity Online 2) is a group of protocols and standards [FID22], such as Web Authentication (WebAuthn) and Client-to-Authenticator Protocol 2 (CTAP2), developed by the FIDO Alliance in collaboration with the World Wide

Web Consortium (W3C). Its goal is to avoid security vulnerabilities and user experience challenges in traditional password-based authentication systems by offering a passwordless authentication framework.

In the FIDO2 framework, users authenticate in a passwordless manner using public key cryptography. The public key is shared with the service, while the private key remains securely on the device. Authentication uses a challenge-response method; the online service sends a challenge that the client answers by signing it with the private key corresponding to the one owned by the server. To do this, the private key will never be sent or stored on the server, sharply reducing the possibility of phishing, password theft, and replay attacks.

2.2.1 Architecture

There are different parties involved in the FIDO2 ecosystem [W3C21]:

- Relying Party (RP): The online platform or service that users aim to access. For authentication, users share their public key with the RP, which is stored by it to later generate a challenge that users need to sign with their private key to authenticate.
- Client: Party that serves as a bridge between the RP and the Authenticator, handling the exchange of data and communication between both parties for the authentication process. This can be, for example, a phone or a laptop.
- Authenticator: A device that validates a user's identity by signing challenges sent by an RP with the private key corresponding to the public key the RP has stored. There are two different categories:
  - Roaming Authenticators: Portable devices that can work across multiple platforms. This is the case of the OFFPAD.
  - Platform Authenticators: Embedded within a specific device permanently, acting as a trusted component for authentication.

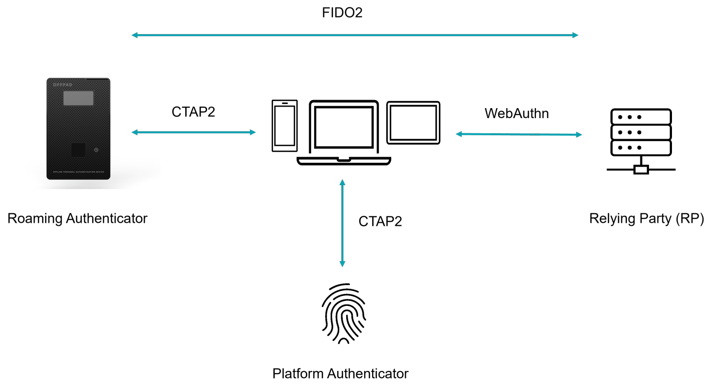


Figure 2.1: FIDO2 Ecosystem.

2.2.2 Client to authenticator protocol

Client to Authenticator Protocol (CTAP) is a protocol that establishes a standardized method for the interaction between a client and authenticator [W3C19a]. There are two protocol versions, CTAP1 and a modification of it, CTAP2, but the term CTAP may be used without clarifying whether it is referring to CTAP1 or CTAP2.

Before the protocol, the client and authenticator need to establish a confidential and mutually authenticated data transport channel. Moreover, the authenticator implementing this protocol requires to have a mechanism to obtain a user gesture, such as a consent button, password, PIN, or biometric.

Structure

The protocol consists of three parts:

- Authenticator API: In this abstraction level, operations are defined similarly to API calls, accepting input parameters and returning either an output or an error code. This API is conceptual; each platform provides actual APIs.
- Message encoding: To invoke a method in the authenticator API, the host needs to construct and encode a request and send it to the authenticator, which then processes it and returns an encoded response.
- Transport-specific binding: For each transport technology, bindings are specified for this protocol.

2.2.3 Web authentication

Web Authentication API, also known as WebAuth, is a specification by W3C and FIDO. It allows servers to register and authenticate users by means of public key cryptography instead of a password [W3C19b]. Instead of a password, a private-public key pair, known as a credential, is created, with the private key safely stored at the user's device and the public key sent and stored at the server, which can then use that public key to verify the user's identity.

2.2.4 Registration

The FIDO registration process allows users to create and link a new passkey credential to their account with a Relying Party. To do this, it is required to have the user, the device equipped with a FIDO authenticator, and the RP's server participate in this process. This involves verifying the user's identity through an authentication method such as the use of a fingerprint or PIN code.

The registration flow is the following [W3C19b]:

0. The user starts the registration on the RP's website or application, providing a username, after which the client sends a request to register an authenticator.
1. The RP's server generates a `PublicKeyCredentialCreationOptions` object containing information about the user's account, the RP's details, and the properties of the new credential, and sends it to the client.
2. The client communicates with the available FIDO authenticator and forwards relevant information from `PublicKeyCredentialCreationOptions`, and a hash of the serialized client data `clientDataHash`, containing the challenge, to this authenticator.
3. The authenticator prompts the user for authentication, and if it is successful, then the authenticator is authorized to generate a new passkey, credential ID, and attestation data, which proves the type and characteristics of the authenticator.
4. The authenticator returns an `attestationObject` to the client, which is an object that contains the generated public key, credential ID and attestation data.
5. The client generates a `ClientDataJSON` object that contains the public key and credential data, and this is sent together with the `attestationObject` to the RP's server.
6. The RP's server receives the data and performs some validation steps. This includes the verification of the signature of the attestation data and examining the `ClientDataJSON` to ensure authenticity and integrity. If the validation is successful, the RP securely stores the public key and associates it with the user's account and authenticator characteristics. This enables future authentication using the corresponding secret key held by the authenticator.

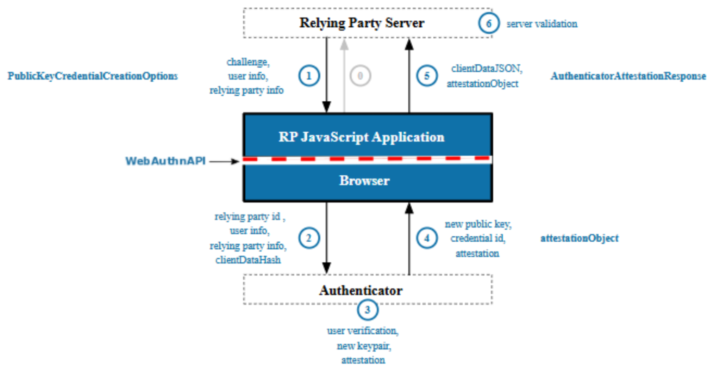


Figure 2.2: Registration Flow.

2.2.5 Authentication

The FIDO registration process allows users to prove their identity to a Relying Party after they have performed the registration procedure. To do this, it is again required to have the user, the device equipped with a FIDO authenticator, and the RP's server participate in this process. The user in this case needs to provide their username and complete authentication on their authenticator to demonstrate possession of the secret key associated with the registered public key.

The authentication flow is the following [W3C19b]:

0. The user starts the authentication on the RP's website or application, providing a username, after which the client sends a request to start the authentication procedure.

1. The RP's server generates a `PublicKeyCredentialRequestOptions` object containing a challenge generated by the RP that will be signed and a list `allowCredentials` of the registered credentials previously registered by the user, and sends it to the client.

2. The client calculates a hash with the client data `clientDataHash` and together with the RP's identifier `RP ID` sends it to the user's authenticator.

3. The authenticator finds the credential that matches the `RP ID` and prompts the user to perform authentication. If it is successful, then the authenticator creates an assertion by signing `clientDataHash` and `authenticatorData`, which contains information about the authentication event, with the private key associated with the credential.

4. The authenticator returns the `authenticatorData` and the generated digital signature to the client.

5. The client forwards the `authenticatorData` and the signature back to the RP's server, together with the `clientDataJSON`, which contains the client data that was originally passed to the authenticator, serialized in JSON format.

6. The RP's server receives the data and performs some validation steps. This includes the verification of the signature, the `authenticatorData`, and the `clientDataJSON`, ensuring the challenge and all parameters are as expected. If all these checks are correct, the RP confirms the user's identity and grants them access.

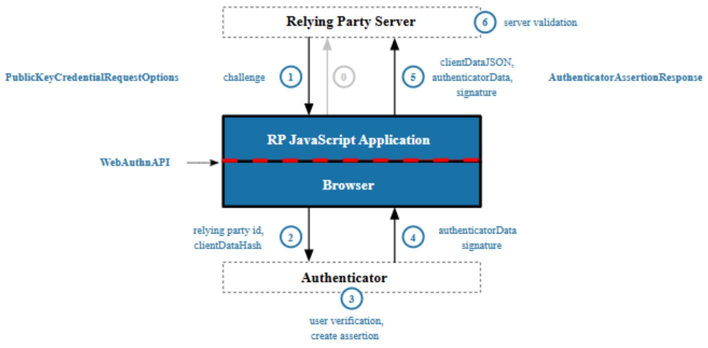

Relying Party Server
6
server validation
PublicKeyCredentialRequestOptions
challenge
1
0
5
clientDataJSON, authenticatorData, signature
AuthenticatorAssertionResponse
RP JavaScript Application
WebAuthnAPI
Browser
relying party id, clientDataHash
2
4
authenticatorData signature
Authenticator
3
user verification, create assertion

Figure 2.3: Authentication Flow.

2.3 Mathematical background

Cryptography is based on mathematical concepts to provide secure communication. A fundamental concept in many cryptographic techniques is the use of math problems that are easy to solve in one direction but difficult to undo without any prior knowledge. One example is the calculation of the product of two large prime numbers; it is easy to obtain the product, but factoring it is a much harder task.

This asymmetry is key in many cryptographic techniques. In these structures, Elliptic Curve Cryptography (ECC) has a key role; it takes advantage of the mathematical properties of elliptic curves over finite fields to design cryptographic protocols that provide security while maintaining relatively small key sizes.

2.3.1 Elliptic curves

For this project, we use elliptic curves over a finite field. A finite field $\mathbb{F}_p$ consists of integers $(0, 1, 2, ..., p-1)$, where p is a prime, and operations are performed modulo p. An elliptic curve over $\mathbb{F}_p$ is a set of points defined by the equation $y^2 = x^3 + ax + b$, including a point at infinity $\mathcal{O}$ [Sil09, Chapter 3].

$$E_{a,b} = \{(x, y) \in \mathbb{F}_p \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

Where the discriminant $\Delta = 4A^3 + 27B^2$ is nonzero. They form a structured group where points can be added and scaled efficiently, with group operations using algebraic rules adapted to finite fields.

Point addition

Given two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on $E_{a,b}$, some important properties to consider when adding points in elliptic curves are [Sil09, Chapter 3] [Och22]:

- Additive identity: $P + \mathcal{O} = P$.
- Additive inverse: if $x_1 = x_2$ and $y_1 = -y_2$, then $P + Q = \mathcal{O}$.
- Point addition:

$$P + Q = (x_3, y_3), \text{ where:}$$

$$x_3 = \lambda^2 - x_1 - x_2,\ y_3 = -y_1 - \lambda \pmod{x_3 - x_1}$$

$$\lambda = \begin{cases} \frac{3{x_1}^2 + a}{2y_1} & \text{if } P = Q \\ \frac{y_1 - y_2}{x_1 - x_2} & \text{if } P \neq Q \end{cases}$$

Additionally, for scalar multiplication of points, it consists of repeated addition, and it is denoted as $Q = [x]P$, where $[2]P = P + P$.

2.3.2 Elliptic curve cryptography

Elliptic Curve Cryptography (ECC) [Mil86] leverages the structure of elliptic curves over finite fields to implement secure cryptographic protocols like Elliptic Curve Diffie-Hellman (ECDH) or Elliptic Curve Digital Signatures (ECDSA) [JMV01]. Its security relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP), and provides a series of advantages [Mag16]:

- Shorter key sizes: a typical ECC key size of 256 bits provides equivalent security to a 3072-bit RSA key, and 2048-bit RSA only provides 112 bits of security.
- Performance: due to the smaller key sizes, ECC provides faster encryption and decryption, as well as lower computational overhead when compared with RSA.

Elliptic curve discrete logarithm problem

The security of ECC is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP) [Sil09, Chapter 5] [Men08] [HM11]. Having two points $P, Q \in E(\mathbb{F}_p)$, where $E(\mathbb{F}_p)$ is an elliptic curve over a finite field and p is a prime. The ECDLP consists of finding an integer x that satisfies the equation $[x]P = Q$.

Even with the best algorithms to solve ECDLP, like Pollard's ρ method [Pol78], it requires $O(\sqrt{p})$ operations. This means it is not currently feasible to solve ECDLP in $E(\mathbb{F}_p)$ if p is large enough.

2.4 Digital signatures

To ensure the authenticity and integrity of exchanged information, digital signatures are employed. A digital signature provides these characteristics through cryptography to implement a way to verify the origin of exchanged information and to check that it has not been altered [SY06]. It functions as the digital counterpart of a handwritten signature or stamped seal, not only confirming the identity of the sender but also ensuring that the content has not been modified during its transmission using public key cryptography. In a digital signature, a key pair consisting of a secret key and a public key is used; the secret key is used for signing, and the public key for verification. They have a big span of possible applications, ranging from secure email communication to electronic voting.

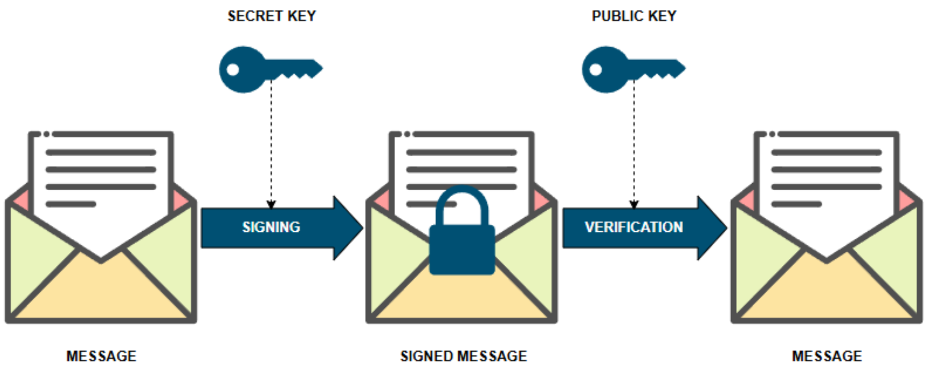


Figure 2.4: Digital signatures overview.

The OFFPAD supports FIDO2 with RSA and ECDSA schemes, and the Schnorr signature scheme is also a scheme that could be interesting towards implementation with the OFFPAD. All of them are digital signature schemes, where the security relies on only the signer holding the private key being able to sign.

2.4.1 RSA

First, the Rivest-Shamir-Adleman signature scheme, commonly known as RSA [RSA78], is based on the computational difficulty in finding the prime factorization of large bi-primes, the product of two distinct prime numbers, since no computer has been able to execute it in a feasible amount of time.

The process works as follows [LLM23]:

Key generation:

1. Privately sample two large random prime numbers p and q.
2. Compute the product n as $n = (p \cdot q)$, and then compute $\phi(n)$ as $\phi(n) = \phi(p) \cdot \phi(q) = (p-1) \cdot (q-1)$.
3. Choose an integer e with $\gcd(e, \phi(n)) = 1$, and compute d such that $e \cdot d \equiv 1 \pmod{\phi(n)}$.
4. The secret key sk is $\mathsf{sk} = (p, q, d)$ and the public key pk is $\mathsf{pk} = (n, e)$.

Signing:

1. Calculate a hash of message m as $h = H(m')$, where H is a hash function, and m' is the output of a suitable padding scheme applied to m such as PKCS#1v1.5.
2. Calculate a signature $s \equiv h^d \pmod{n}$.

Verification:

The signature is valid if $s^e \equiv h \pmod{n}$.

2.4.2 ECDSA

Second, the Elliptic Curve Digital Signature Algorithm, commonly known as ECDSA [JMV01], uses elliptic curve cryptography to enable secure digital signatures. ECDSA bases its security on the elliptic curve discrete logarithm problem since it is computationally challenging to solve.

The process works as follows [KKS10]:

Key generation:

A key pair is generated with the private key sk being a number selected uniformly at random such that $1 \leq \mathsf{sk} \leq (n-1)$, where n is the order of the generator point G on the curve, and the public key pk is computed as $[\mathsf{sk}]G$.

Signing:

1. An integer k is chosen uniformly at random with $1 \leq k \leq (n-1)$.
2. Compute a point on the curve $R = [k]G = (x, y)$.

3. Compute $r \equiv x \pmod{n}$. If $r = 0$, then go back to the first step.

4. Compute $k^{-1} \pmod{n}$.

5. Compute $s \equiv k^{-1}(H(\mathsf{pk}, m) + \mathsf{sk} \cdot r) \pmod{n}$, where H is the Secure Hash Algorithm SHA-2. If $s = 0$, then go back to the first step.

6. The signature on the message m is the pair $(r,\ s)$.

Verification:

1. Calculate $s' \equiv s^{-1} \pmod{n}$.

2. Calculate $R' = [H(\mathsf{pk}, m) \cdot s']G + [r \cdot s']\mathsf{pk} = (x', y')$, and $r' \equiv x' \pmod{n}$.

3. The signature is valid if $r \equiv r' \pmod{n}$.

2.4.3 Schnorr

Finally, the Schnorr signature scheme [Sch91] is also based on the discrete logarithm problem in finite groups. This scheme is valued for its efficiency and security, as it generates simpler and faster signatures compared to ECDSA.

The process works as follows [Won21]:

Key generation:

1. Sample the secret key sk uniformly at random from a finite field $\mathbb{Z}_p$.

2. Compute public key as $\mathsf{pk} = [\mathsf{sk}]G$, where G is the generator point.

Signing:

1. Sample a random value r from the finite field $\mathbb{Z}_p$ and compute $R = [r]G$.

2. Compute challenge $c = H(\mathsf{pk}, R, m)$, where H is a hash function that behaves as a random oracle and must be collision-resistant, which can be instantiated with a Secure Hash Algorithm such as SHA-2.

3. Compute response $z = r - c \cdot \mathsf{sk}$.

4. The signature is $\sigma = (c, z)$.

Verification:

1. Calculate $R' = [z]G + [c]\mathsf{pk}$.
2. The signature is valid if $c = H(\mathsf{pk}, m, R')$ is correct.

Chapter 3

Threshold signatures

In this chapter we present an analysis of threshold signatures we performed in the project preceding this thesis [Gar25b], with some other findings discovered during the project to better understand threshold signature schemes.

3.1 Threshold signature schemes

In a threshold signature scheme [BS15, Chapter 22], the secret signing key sk is divided into N shares, each stored in a different server or device. These collaborate in generating signatures, but the full key is never reconstructed at any single point to avoid having a single point of failure and improve security. Only a subset of T servers out of the total N are needed to produce a valid signature.

By doing this, even if $T-1$ servers are compromised, the attackers cannot retrieve useful information about the signing key or produce valid signatures on their own, making this scheme secure. These schemes also provide robustness, which we define as the ability to generate valid signatures as long as T participants are still available, even if the remaining $N-T$ participants are not. This means that even if the honest users lose their key, they can still sign if they have T shares left. Moreover, the signature process is designed so that the shares do not leak the secret key.

A generic threshold signature scheme consists of (G, S, V, C), where G is a probabilistic key generator to generate N key shares, S is a probabilistic signing algorithm, V is a deterministic verification algorithm, and C is a combiner algorithm.

The process works as follows:

Shares generation:

1. G generates N key pairs $(\mathsf{pk}, \mathsf{sk}_i)$ and gives sk_i to the corresponding device i. The public verification key and the combiner key are the same $\mathsf{pk} = \mathsf{pkc}$.

2. Each signing device i outputs a signature share $\sigma_i = S(\mathsf{sk}_i, m)$ where m is the message to sign.

Share combination:

The combiner $C(\mathsf{pkc}, m, \mathcal{J}, \{\sigma_j\}_{j\in\mathcal{J}})$ collects T signature shares and combines them to obtain the full signature σ, where $\mathcal{J}$ is a subset of the participants.

Share verification:

Algorithm $V(\mathsf{pk}, m, \sigma)$ ensures the authenticity and integrity of the signature, outputting accept if $|\mathcal{J}| = T$, meaning that the set contains exactly T participants, and each partial signature σ_j is a valid signature on m.

3.2 Secret sharing

Secret sharing is a technique in cryptography used to split a secret into multiple parts, called shares, in a way such that having a certain amount of those shares makes it possible to recover the initial secret, but knowing less than that does not leak any information about it.

One of the most well-known methods is that of Shamir's secret sharing scheme [Sha79]. In this scheme, we want to share a secret α among N different parties. To do that we work in a finite field $\mathbb{Z}_q = \{0, 1, ..., q-1\}$ where q a prime number.

The scheme uses polynomial interpolation to do the sharing, as any polynomial of degree $T-1$ can be uniquely determined by T points, and for fewer than T points, no useful information can be obtained about the secret. It consists of two different processes, one for generating the shares of the secret, and the other to combine them and recover the secret, and requires that $q > N$.

To share a secret α, the process works as follows [BS15, Chapter 22]:

Shares generation:

1. Choose $T-1$ random coefficients $(a_1, a_2, ..., a_{T-1})$ from $\mathbb{Z}_q$.
2. Define a polynomial of degree $T-1$, such that $f_i(0) = \alpha$, where α is the secret to share:
$$f(x) = a_{T-1}x^{T-1} + a_{T-2}x^{T-2} + ... + a_1x + \alpha.$$
3. Generate a share for each participant as $\alpha_i = f(i)$.

Share combination:

1. Collect T shares of a subset $\mathcal{J}$ of the participants.
2. Reconstruct the polynomial using Lagrange interpolation
$$f(x) = \sum_{j=1}^{T} \alpha_{i_j} \cdot \lambda_{i_j}(x) \pmod q, \text{ where } \lambda_{i_j}(x) = \prod_{\substack{k=1\\k\neq j}}^{T} \frac{x - i_k}{i_j - i_k} \pmod q.$$
3. Get the secret α by evaluating $f(0) = \sum_{j=1}^{T} \alpha_{i_j} \cdot \lambda_{i_j}(0) \pmod q = \alpha$.

While Shamir's secret sharing ensures confidentiality, it does not guarantee that the distributed shares are consistent or correctly generated. To ensure this, we rely on verifiable secret sharing (VSS), a secret sharing scheme that splits a secret into different verifiable shares, meaning that a party obtaining a share can verify its validity [Sch05].

A well-known example of a verifiable secret sharing scheme is Feldman's scheme [Fel87]. This scheme uses commitments, which allow a party to commit to a value while keeping it secret, with two main properties: hiding (the commitments do not reveal information about the committed value) and binding (after committing to a value, the commitment cannot be changed). This is employed to allow parties to verify that the shares they receive are consistent with the polynomial without revealing the coefficients.

In this scheme, a secret α is been chosen uniformly at random from a finite field $\mathbb{Z}_p$, where p is a large prime number. It is split into different shares using a polynomial f where $\deg(f) \leq T$ such that $f(x) = a_0 + a_1 x + \ldots + a_T x^T$, and the zero degree term $a_0 = \alpha$ is the secret that we want to distribute. A dealer sends shares $\alpha_i = f(i)$ for $1 \leq i \leq N$ privately to each party P_i, and broadcasts commitments $commit(a_j) = c_j = G^{a_j}$ for $1 \leq j \leq T$ of all the coefficients in the polynomial f except for the constant term. Each party can verify its share by checking if

$$G^{\alpha_i} = \prod_{j=0}^{T} c_j^{i^j}.$$

3.3 FROST protocol

Flexible Round-Optimized Schnorr Threshold, or FROST, is a cryptographic protocol using Schnorr [Sch91] for threshold signatures that optimizes the rounds required for signing while maintaining security [KG20]. It minimizes the network overhead caused by producing Schnorr signatures in a threshold signature scheme while enabling unrestricted parallelism of signing operations.

FROST trades off robustness in the protocol for improved round efficiency, as in settings where misbehaviour of participants is rarely expected, protocols can be more relaxed to be more efficient if all participants honestly follow the protocol. If a participant misbehaves, honest participants can identify it and abort the protocol to re-run it after excluding the misbehaving participant.

In the project's implementation, which follows the work from Connolly, Komlo, Goldberg, and Wood [CKGW24], the protocol can be carried out with each participant simply performing a broadcast, but a device can also be the one in charge of acting as the signature aggregator, a semi-trusted role that can be performed by any participant or even an external party. In the implementation of this project, we use this approach with a signature aggregator, who is in charge of reporting misbehaving participants and publishing the group's signature at the end of the protocol. Even if it deviates from the protocol, it remains secure as this role cannot learn the private key or cause improper messages to be signed.

3.3.1 Parameter setting

To perform this protocol, some parameters need to be first established:

- Let $\mathbb{G}$ be a cyclic group of group order q, where the operations take place. This group must satisfy the DLP hardness, which means that it must not be possible to compute x given G^x, where G is the generator.
- Let $\mathbb{Z}_q$ be a finite field of prime order q, where q is a large prime number.
- Let $G \in \mathbb{G}$ be a public generator of the group. It should be chosen such that $\mathbb{G}$ has prime order and no small subgroups.
- Let H_1, H_2 be two hash functions whose outputs are in $\mathbb{Z}_q^*$.

3.3.2 Key generation

Distributed key generation

FROST uses Pedersen's Distributed Key Generation (PDKG) [Ped91] for key generation. This consists of each participant executing Feldman's verifiable secret sharing [Fel87] to generate a share and send it to the other participants; the final secret share is the result of adding all the shares received from other participants. Moreover, each participant demonstrates knowledge of their secret by providing the rest of the participants with a zero-knowledge proof [GMR85], a cryptographic technique for demonstrating to another party that a statement is true without revealing any meaningful information aside from the truth of the statement.

Each participant performs the following steps:

1. Choose $T-1$ random coefficients $(a_{i1}, a_{i2}, ..., a_{iT-1})$ from $\mathbb{Z}_q$.
2. Define a polynomial of degree $T-1$, such that $f_i(0) = a_{i0}$, where a_{i0} is the secret to share:
$$f(x) = a_{iT-1}x^{T-1} + a_{iT-2}x^{T-2} + ... + a_{i1}x + a_{i0}.$$
3. Compute a proof of knowledge $\sigma_i = (R_i, \mu_i)$, where the values R_i and μ_i are $R_i = [k]G$ and $\mu_i = k + a_{i0} \cdot h_i$, where $h_i = H_i(i, \Phi, [a_{i0}]G, R_i)$, k is a random scalar sampled from $\mathbb{Z}_q$, and Φ is a context string to prevent replay attacks.
4. Create commitment $\mathit{commit}(a_{i0}) = c_i = G^{a_{i0}}$ and share it together with the proof.
5. Verify all proofs from other participants checking $R_j = [\mu_j]G \cdot {c_{j0}}^{h_j}$, and abort if verification fails. If everything is correct, delete all proofs.
6. Send to each other participant a secret share $f_i(j)$, deleting them afterwards and keeping $(i, f_i(i))$ for themselves.
7. Verify the shares by calculating $[f_j(i)]G = \prod_{k=0}^{T-1} h_{jk}^{i^k \pmod q}$, aborting if the check fails.
8. Calculate private signing share $\mathsf{sk}_i = \sum_{j=i}^{N} f_j(i)$, storing it securely and deleting each $f_j(i)$.
9. Calculate public verification share $\mathsf{pk}_i = [\mathsf{sk}_i]G$ and group's public key
$$\mathsf{pk} = \prod_{j=1}^{N} h_{j0}.$$

Trusted dealer

A different approach for key generation includes the use of a trusted dealer. With this approach, one device is in charge of generating all the shares instead of each device generating its own share. While this simplifies the implementation, it introduces a potential security risk, as all the shares exist on a single device at some point in time.

In this process, the trusted dealer performs the following steps:

1. Randomly samples a secret.
2. Choose $T-1$ random coefficients $(a_1, a_2, ..., a_{T-1})$ from $\mathbb{Z}_q$.
3. Generate a share for each participant as $\alpha_i = f(i)$.
4. Send the share $f(i)$ to each participant together with the commitments $\text{commit}(a_j) = c_j = G^{a_j}$ for $1 \leq j \leq T$.

Each participant then receives a share and verifies if it is correct by using the commitments received following verifiable secret sharing (Section 3.2). If everything is correct, then all participants are prepared for signing, and the dealer deletes all information.

3.3.3 Signing

Before signing a message, a preprocessing stage is carried out to generate nonces and commitments needed for the signing. But to not cache commitments, a two-round signing protocol can be implemented instead, with participants publishing a single commitment to each other in the first round.

Round 1: Nonce generation and commitments

For the preprocess, with an empty list L_i and j being a counter for a specific nonce/commitment share pair, and π the number of pairs generated at a time, each participant performs these steps:

1. Sample single use nonces (d_{ij}, e_{ij}) from $\mathbb{Z}_q^* \times \mathbb{Z}_q^*$.
2. Derive commitment shares $(D_{ij}, E_{ij}) = ([d_{ij}]G, [e_{ij}]G)$.
3. Append (D_{ij}, E_{ij}) to L_i and store $(d_{ij}, D_{ij}), (e_{ij}, E_{ij})$ for later use.
4. Publish (i, L_i).

Round 2: Signature aggregation

Once this preparation stage is finished, a set of participants $\mathcal{J}$ signs a message m. The process is the following:

1. Signature aggregator fetches the next available commitment share for each participant $P_i \in \mathcal{J}$ from L_i and constructs $B = \langle (i, D_i, E_i) \rangle_{i \in \mathcal{J}}$.
2. Signature aggregator sends P_i the tuple (m, B).

3. Each participant validates the message m, and then checks $D_j, E_j \in \mathbb{G}^*$ for each commitment in B, aborting if the check fails.

4. Each participant computes $\rho_j = H_1(j, m, B)$, derives the group commitment $R = \prod_{j \in \mathcal{J}} D_j \cdot [\rho_j]E_j$ and the challenge $c = H_2(R, \mathsf{pk}, m)$.

5. Each participant computes their response using their sk_i by computing
$$s_i = d_i + (e_i \cdot \rho_i) + \lambda_i \cdot \mathsf{sk}_i \cdot c.$$

6. Each participant securely deletes $(d_{ij}, D_{ij}), (e_{ij}, E_{ij})$ and returns s_i to the signature aggregator.

7. The signature aggregator derives $\rho_i = H_1(j, m, B)$ and $R_i = D_{ij} \cdot [\rho_i]E_{ij}$, and subsequently $R = \prod_{i \in \mathcal{J}} R_i$ and $c = H_2(R, \mathsf{pk}, m)$.

8. The signature aggregator checks $[s_i]G = R_i \cdot \mathsf{pk}_i^{c \cdot \lambda_i}$ for each s_i and aborts the process if the check fails.

9. The signature aggregator computes the group's response $s = \sum s_i$ and publishes $\sigma = (R, s)$ along with m.

3.3.4 Advantages

By following this protocol for threshold signatures, it is possible to efficiently produce a valid Schnorr signature on a message when having a subset of valid participants collaborate in the process, without revealing their secrets and with efficient use of rounds of communication. In conclusion, the use of this protocol achieves secure, robust, efficient and flexible threshold signing suitable for real-world scenarios. Some key achievements obtained from using this protocol are:

– Efficient threshold signing: Enabling a threshold number of participants to produce a valid signature using a minimal number of communication rounds. This reduces latency and network overhead.
– Security: The protocol guarantees that the secret key shares are never revealed during the signing process, and the full secret key is never reconstructed, using signature shares to compute the final signature. Moreover, it uses commitments, zero-knowledge proofs and nonce-hiding techniques to ensure attackers cannot interfere in the process.
– Support for trusted or trustless setup: The protocol is compatible with both a DKG generation and a generation with a trusted dealer. This flexibility allows one to choose the setup that best aligns with the trust assumptions and deployment limitations that a user who wants to implement it can find.

– Use of a signature aggregator: The use of a signature aggregator allows the coordination of the signing process. Although semi-trusted, this device is not capable of learning the secret key or generating signatures on its own, ensuring the protocol remains secure even if it misbehaves.

3.4 ECDSA three-round protocol

Doerner, Kondi, Lee, and Shelat have designed a three-round threshold ECDSA protocol [DKLsa24], which is a cryptographic protocol that generates distributed ECDSA signatures without compromising the security of the private key. This protocol is both secure and efficient, making it suitable for practical deployment in real-world environments.

Compared to other threshold signature protocols that require numerous rounds of communication or rely on expensive cryptographic primitives, this protocol allows the implementation of threshold signatures with only three rounds of communication, eliminating the need for zero-knowledge proofs thanks to the use of an enhanced statistical consistency check mechanism, improving efficiency without compromising security.

The main innovation of the protocol lies in the method for computing the ECDSA signing equation; it uses an intermediate representation to enable the execution of non-linear operations. This is complemented by an innovative consistency check that uses the natural structure of the computation, eliminating the need for additional cryptographic checks.

3.4.1 Parameter setting

To perform this protocol, several parameters need to be established:

– Let $\mathbb{G}$ be an elliptic curve of prime order q, where the operations take place. This group must satisfy the discrete logarithm problem (DLP) hardness assumption.
– Let $\mathbb{Z}_q$ be a finite field of prime order q, where q is a large prime number.
– Let $G \in \mathbb{G}$ be a public generator of the group, chosen such that $\mathbb{G}$ has prime order.
– Let H_1, H_2 be two hash functions whose outputs are in ${\mathbb{Z}_q}^*$.

3.4.2 Key generation

Relaxed distributed key generation

The protocol makes use of a "relaxed" key generation that does not require zero-knowledge proofs. This relaxed approach is enough to ensure security for the signing protocol while reducing computational overhead.

Each participant performs the following steps:

1. Choose $T-1$ random coefficients $(a_{i1}, a_{i2}, ..., a_{iT-1})$ from $\mathbb{Z}_q$.
2. Define a polynomial of degree $T-1$, such that $f_i(0) = a_{i0}$, where a_{i0} is the secret to share:
$$f(x) = a_{iT-1}x^{T-1} + a_{iT-2}x^{T-2} + ... + a_{i1}x + a_{i0}.$$
3. Compute the polynomial $P_i(x) = p_i(x) \cdot G$ in the elliptic curve group.
4. Commit to their polynomial in two ways:
   a) Public polynomial commitment: Commit to the first T evaluation points of the polynomial $P_i(x)$ in the group $\mathbb{G}$: $P_i(0), P_i(1), ..., P_i(T-1)$. Broadcast these commitments to all other parties.
   b) Pairwise share commitment: Commit to the secret share $p_i(j)$ and privately send it to participant j.
5. Verify that the received secret share is consistent with the public polynomial commitment by computing $P'(i) = \sum_{j \in N} p_j(i) \cdot G$ and verify that:

$$P'(i) = \begin{cases} P(i) & \text{if } i \in [T-1] \\ \dfrac{P(0) - \sum\limits_{j \in [T-1]} \text{lagrange}([T-1] \cup \{i\}, j, 0) \cdot P(j)}{\text{lagrange}([T-1] \cup \{i\}, i, 0)} & \text{otherwise} \end{cases}$$

3.4.3 Signing

The protocol achieves efficiency by using a rewriting of the ECDSA signing equation. It computes:

– $R = [r]G$, where r is the collective nonce.
– $w = (a + \mathsf{sk} \cdot b) \cdot \varphi$, where $a = H_1(m)$, $b = r_x$, r_x is the x-coordinate of R, and φ a masking value.
– $u = r \cdot \varphi$.
– $s = w/u$, which yields the final signature component.

This formulation enables the computation of the three nonlinear relations defining R, w, and u in parallel, resulting in a three-round complexity. Moreover, Vector Oblivious Linear Evaluation (VOLE) [ADI+17], a cryptographic primitive that represents a vectorized form of oblivious linear evaluation, allowing multiple multiplication operations to be performed efficiently in parallel, is employed to reduce round complexity. Therefore, the signing protocol consists of three rounds of communication, where each signing party performs the following steps:

Round 1: Nonce commitment and VOLE setup

1. Generate random nonces $k_i, \varphi_i \in \mathbb{Z}_q$ and compute commitment $c_i = \mathrm{Commit}[k_i]G$.
2. Initialize VOLE instances with all other participants $j \neq i$ to prepare for secure multiplication operations.
3. Broadcast the nonce commitment c_i to all other participants.
4. Store the nonce k_i and the masking value φ.

Round 2: Decommitment and VOLE execution

1. Reveal the nonce k_i to all other participants.
2. Verify that each received nonce k_j satisfies $c_j = \mathrm{Commit}[k_j]G$.
3. Compute the collective nonce $r = \sum\limits_{j \in \mathcal{J}} k_j$, where $\mathcal{J}$ is the set of signers participating.
4. Execute the VOLE protocol to compute shares of $w = (a + \mathsf{sk} \cdot b) \cdot \varphi$ and $u = r \cdot \varphi$, where $a = H_1(m)$ and $b = r_x$.
5. Perform consistency checks using the elliptic curve group structure to detect any misbehaviour.

Round 3: Signature aggregation

1. Using the outputs from the VOLE protocol, compute the signature share $s_i = w_i / u_i$ and send it to a coordinator.
2. A coordinator receives the shares and verifies their consistency by using statistical checks.
3. The coordinator combines the signature shares to compute $s = \sum\limits_{i \in \mathcal{J}} s_i$, where the final signature is $\sigma = (r_x, s)$.

3.4.4 Advantages

The implementation of this protocol for threshold ECDSA signatures makes it easier to produce a valid ECDSA signature for a message using a group of signers that cooperate with each other. This method protects the private information and involves only three rounds of communication.

To summarize, the protocol offers a secure, efficient, and effective threshold signing mechanism, representing an improvement over previous implementations of threshold ECDSA. Some of the most important advantages of this implementation include:

- Optimization of communication rounds: The covered protocol requires only three rounds of communication, which is a significant improvement over previous threshold ECDSA implementations. This reduction leads to lower latency and complexity.
- Elimination of zero-knowledge proofs: Unlike previous threshold ECDSA protocols that relied on expensive zero-knowledge proof techniques, this implementation takes advantage of statistical consistency checks to lower the computational requirements for a practical deployment.
- Security: The protocol is secure even if the majority of the participants are dishonest. The secret key shares are protected, and the entire secret key is never reconstructed during the signing process.
- Standardization: This implementation produces ECDSA signatures that comply with widely accepted cryptographic standards, and what's more important, with the FIDO2 protocol. This standardization allows for strong support and ensures interoperability among different systems and environments.

Chapter 4 Proposed solution

Following this project's main objective to incorporate threshold signatures in the OFFPAD's ecosystem, we have made several decisions in this project: the use of the Schnorr signatures over ECDSA as the signature scheme due to its simplicity and need for fewer rounds, the use of USB communication to provide secure data exchange between the involved devices, the use of a trusted dealer in the key generation to lessen the computational burden of the used embedded devices, and the adoption of a coordinator-based model together with the use of a signature aggregator to facilitate the communication between devices and aggregation of signature shares. We further explain these decisions in this chapter.

4.1 Signature scheme

With ECDSA (Section 2.4.2) and Schnorr (Section 2.4.3) signatures being the principal options considered in the project, we finally opted to use Schnorr signatures for the implementation. While both of them rely on the security of the Discrete Logarithm Problem, Schnorr was selected due to the advantages it offers compared to other protocols.

While ECDSA signatures offer the advantages of standardization and widespread adoption, Schnorr signatures are increasingly being integrated into modern cryptography protocols due to the benefits they offer. The mathematical structure of Schnorr signatures is simpler, requiring computational operations that are less demanding, and enabling the use of fewer rounds of communication. This made us decide to use Schnorr signatures as a good fit for the OFFPAD's embedded environment.

The implementation with Schnorr for the project is based on elliptic curve cryptography (ECC), where all group operations are performed over an elliptic curve group. Its security relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP), similar to ECDSA. It provides efficient key and signature sizes and faster computations, offering strong security and good performance.

Using Schnorr signatures, the implementation follows the FROST protocol, which offers a series of advantages:

- Round efficiency: minimizing the rounds of communication required for signing, allowing the implementation of a two-round or single-round signing protocol.
- Identification of misbehaviour: identifying faulty operations or participants, enabling the abortion of the signing process in case something is not working as intended.
- Improved privacy and efficiency: implementing the use of a threshold signature scheme that aligns with modern cryptography standards and practices.

4.2 Communication between devices

Communication is key for this project, as the devices need to share information like signature shares and commitments in a safe way. The communication must prevent unauthorized parties from eavesdropping or injecting information. Taking into account the hardware specifications of the boards used in the project, USB and Bluetooth communication were considered.

USB communication

Using USB for communication provides a reliable and fast connection between devices, while also providing security as a physical connection is required, reducing the risk of remote attacks. However, it is not an optimal way of communication when using several devices due to cable management and port availability, making it less practical in scenarios with multiple devices.

The STM32 Nucleo-L476RG development board used in the implementation supported only UART communication, while the OFFPAD development board supported USB HID communication (Section 5.7). The difference in hardware required implementing both types of communication. UART communication provides a simple serial interface that is reliable and well-established, but requires specific drivers and port management. On the other hand, USB HID communication provides a more standardized approach, making it easier to integrate and enable device detection and communication management.

Bluetooth communication

Bluetooth offers a wireless alternative for device communication, enabling a flexible deployment of a multi-device environment. In fact, Bluetooth Low Energy (BLE) optimizes power consumption. While wireless communication is more exposed than USB, Bluetooth implements different features to provide security, such as pairing, bonding, device authentication, encryption, and message integrity [Blu24]. Some limitations are the higher latency compared to USB, with possible distance constraints when communicating between devices, being unable to be too far away, and a higher complexity for implementation.

Selected communication

With the complexity involved in the development and integration of Bluetooth communications and the time limits of the project, we selected USB communication as the initial mode of communication for all phases of the protocol. In future works, the use of Bluetooth should be considered a priority enhancement, considering its potential to increase the scalability and practical deployment of the implementation.

In this project, we integrated both UART and USB HID communication protocols to meet the hardware requirements of the different development boards. This was accomplished using an interface that allows selection of which communication method to use, depending on the desired board to communicate with.

4.3 Signing preparation

To compute a valid signature, a setup needs to be carried out beforehand. This setup needs to provide all the participants with the needed resources to be able to correctly sign a message. This can be performed by means of a distributed key generation or a generation with a trusted dealer.

Distributed key generation

It is a cryptographic protocol that allows a group of devices to work together to generate a secret key without any of them ever knowing the full key, while also generating a public key known to all. To do this, each participant generates a share of a secret key using Shamir's secret sharing (Section 3.2), verifying that all shares are valid using commitments in a protocol such as Feldman's verifiable secret sharing (Section 3.3.2).

Therefore, each participant holds at the end a share of the secret key that can later be used to sign a message, computing a signature share of the message. Performing Lagrange interpolation with these shares results in a valid signature.

This setup provides good security as the secret key is never computed at a single point. This reduces the risk of getting the key stolen, with the attacker having to obtain a certain amount of the shares to be able to compute the full secret key. However, distributed key generation protocols need more complex algorithms that can be difficult to implement, needing communication between all the devices during the process to ensure the desired results.

Trusted dealer

Same as in a distributed key generation, a setup using a trusted dealer aims to end with each participant holding a share of a secret key, which is later used to compute a valid signature for a message (Section 3.3.2). The difference resides in how these shares are generated.

When using a trusted dealer, a device is responsible for generating all the shares as well as the public key to then distribute them amongst all the participant devices. As in the distributed key generation, this is performed using Shamir's secret sharing and verifiable secret sharing (Section 3.2).

This setup is less complex, as only one device is responsible for generating all the shares. The downside is that in this case, there is a single point of failure, with the device acting as the dealer holding all the shares needed to compute the full secret key; consequently, an attacker could obtain the secret key if they corrupt the dealer.

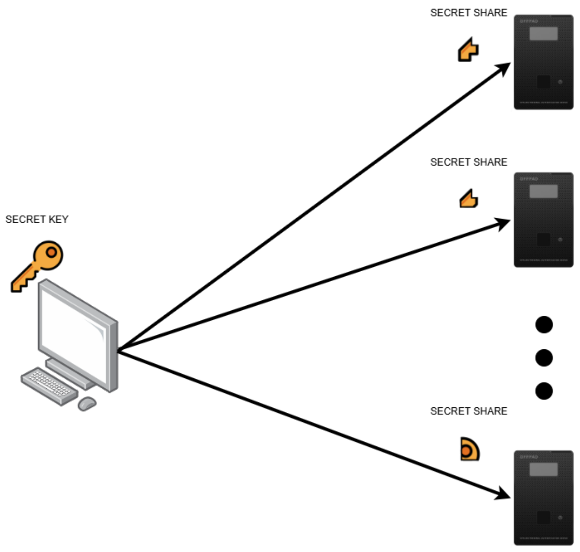


Figure 4.1: Key generation with trusted dealer.

Selected setup

With the trusted dealer setup being less complex and easier to implement, this approach was the one we used in the project. This is a less safe approach when compared with the distributed key generation, but was chosen as a first approach with the distributed key generation as a possibility for a future line of work.

4.4 Signing communication

In the FROST protocol, devices need to exchange information during the signing process, such as commitments or the message to sign. The communication model determines how this information is exchanged between participants, which impacts the complexity, security, and performance of the implementation. Two communication models were considered for this project.

Direct device communication

One possibility involves establishing direct communication between all participating devices, creating a fully connected network where each device can communicate directly with every other device. This provides decentralization, as no single device has access to all communications, reducing single points of failure. However, this requires more complex coordination strategies and places higher computational and communication burdens on the OFFPAD devices.

Coordinator

Another possibility is the use of a coordinator that facilitates all communication between participants. In this model, devices do not communicate directly with each other; instead, the messages are sent to a coordinator, which then redistributes the information to the corresponding devices. This reduces the communication burden for the OFFPAD devices but introduces a single point of failure and requires the coordinator to be a trusted device.

Selected generation

Given the OFFPAD's hardware limitations, we decided to make use of a coordinator, reducing the communication burden from the OFFPAD devices. A computer is designated with the role of the coordinator, being in charge of facilitating communication between all the participant devices while maintaining security, as no important information can be obtained from the exchanged information in the signing process for the selected signature scheme (Section 3.3).

4.5 Share aggregation

Once the signature shares have been computed, they must be aggregated to create the final signature. This aggregation is necessary to complete the FROST protocol, which needs to gather the signature shares from the participants involved in the signing protocol to be able to compute a final valid signature (Section 3.3.3). We considered two different methods to aggregate the signature shares for this project.

Distributed aggregation

One of the considered solutions allowed any device that is part of the signing process to perform the signature aggregation step. This involves any device that participates to be able to collect signature shares from other participating devices, aggregating them to generate the final signature, and then establishing a connection with the server for authentication.

This approach provides greater decentralization, eliminating the threat of a single point of failure in aggregation. However, this requires more computational capacity from the devices.

Signature aggregator

The other approach that was discussed consisted of setting a computer to act as a signature aggregator (Section 3.3), which includes the task of collecting all the signature shares from the corresponding devices, combining them to generate the final signature, and enabling direct communication with the server to authenticate.

This approach lessens the computational burden on the OFFPAD devices, as they only need to compute their individual signature shares and send them to the computer, without needing to communicate with the other participants. However, this creates a single point of failure, as the final signature will always end in the same device.

Selected aggregation

Due to the computational limits of the OFFPAD devices and the need for good performance, the project follows an aggregation with the role of a signature aggregator, taken by a computer. Moreover, due to the limitations of the OFFPAD to establish a connection with a server to perform authentication, the final signature would be required to be sent to a computer, as the OFFPAD devices would not be able to perform the authentication; therefore, even without the computer taking the role of a signature aggregator a single point of failure would still be created.

For this reason, we decided to have a computer carrying out the most computationally intensive operations, taking the role of a signature aggregator. By doing this, the computer is the one in charge of gathering the signature shares of all the participants, aggregating all of them to compute the final signature, and communicating with a server to complete the authentication.

Chapter 5 Methodology

In this chapter we outline the methodological approach used to achieve the goals of the project, which aims to integrate threshold signatures into the OFFPAD's authentication mechanism. The methodology follows a structured strategy we developed in the project preceding this thesis [Gar25b], as well as incorporating the challenges and learnings that we encountered during the implementation.

5.1 Project overview

The project followed a four-phase approach, starting with acquiring a theoretical background to finish designing a solution and implementing it, and finally testing and evaluating the results. The methodology, although looking linear, followed some iterative processes, where results in later phases were used to refine decisions made in earlier stages.

- **Research phase:** The beginning of the project involved a thorough review of literature to determine the current state of threshold signature schemes and communication models. This step was critical to enable informed decision-making and ensure the design of the project's proposed solution was in line with the OFFPAD's limitations.
- **Design phase:** Building on the research background acquired, this phase focused on the system design, including the decision of the threshold signature scheme and communication model to use.
- **Implementation phase:** In this phase, design decisions were translated into operational code using an iterative development process. Initial development focused on setting up basic functionality, which was then complemented by setting up the communication, and finally merging all together to obtain a functioning threshold signature scheme.

– **Testing and evaluation phase:** The final phase involved a comprehensive evaluation of the results obtained from the implementation. Functional testing was employed to determine the correct functionality of the protocol, and performance testing was performed to analyse different parameters.

This approach made it possible to develop a functional threshold signature scheme in an efficient and effective way. Each of the phases maintained flexibility to revisit and refine earlier decisions based on implementation and testing results. The methodology emphasised continuous validation and adaptation, ensuring that theoretical knowledge could translate into practical functionality.

5.2 Research phase

The research phase consisted of a literature review of existing threshold signature schemes, with a particular focus on those using Schnorr or ECDSA signatures. This research was targeted to search for existing implementations that could overcome the constraints of embedded systems while also being compatible with the OFFPAD's hardware. Not only were threshold signature schemes explored, but communication models were also an important focus of the research, looking for models that could fit into the project's requirements.

The literature review followed standard academic rules to ensure the validity and coherence of its theoretical foundations. Peer-reviewed publications from important conferences and journals in the field of cryptography were prioritized, as well as standardization documents by organizations like the FIDO Alliance. This approach ensured that design decisions were made based on reliable sources.

5.3 Design phase

The design phase followed a decision-making process to take the findings from the literature review and translate them into concrete technical decisions. The design decisions were made to use the knowledge from the literature review, taking into account the practical limitations, in particular, the limitation of being able to use only two boards for the implementation phase.

Threshold signature scheme selection

Based on the results of the literature review, the suitability of different candidates for the threshold signature schemes for the OFFPAD environment was evaluated. The selection process involved a careful consideration of each of the candidate schemes against the goals of the project.

An extensive analysis of both Schnorr-based and ECDSA-based approaches was carried out, taking into account security properties and their practical implementation requirements. The decisions regarding the threshold signature scheme included evaluating existing implementations and possible modifications that could be applied to them to comply with the limitations of the project.

Communication model design

The choice of a communication model required the adaptation of multi-party protocols to fit the constraints of the OFFPAD devices, all while maintaining the security properties of threshold signatures. An analysis of the different communication strategies presented in the literature was carried out, determining the adaptations required for each strategy to be able to work with the OFFPAD devices. Special attention was given to how a computer could be involved in the protocol without compromising security.

5.4 Implementation phase

The implementation phase used an incremental development approach, where features were developed separately so they could be evaluated before being integrated together. This enabled constant validation of design decisions and early detection of problems and errors when translating decisions into code.

The implementation followed a bottom-up approach, moving through different phases, each based on the preceding ones. This incremental approach was followed to identify and solve implementation problems early in the development.

1. **Library selection:** The implementation started with an evaluation and selection of cryptographic libraries suitable for embedded threshold signatures. The support for a Zephyr OS implementation was key when selecting a library, as well as the programming language, which was decided to be the C programming language. When choosing a library, the candidates needed to be able to perform operations for Schnorr and ECDSA signatures, as well as being able to perform operations for secret sharing, including Shamir's secret sharing and verifiable secret sharing. All cryptographic operations were performed using the secp256k1 elliptic curve, which was selected for its widespread adoption in cryptographic applications and excellent support in embedded cryptographic implementations.

2. **Hardware compatibility:** After selecting the library, a thorough examination was carried out to determine the ability to run all the necessary cryptographic operations consistently on the specified hardware. This phase was required to execute and test the performance of all the required operations in the boards

used for the implementation, to make sure the operations took place within a reasonable time for authentication scenarios.

3. **Development of communication:** The communication was developed taking into consideration the limitations of the boards used. Communication between a computer and each board was implemented using reliable USB communication and message serialization. Both UART and USB HID communication methods were developed as each board was compatible with only one of these methods.

4. **Threshold signature scheme:** With the communication configured and working, the threshold signature algorithms were implemented in a set of sequential steps.

    a) The first step was key generation, which was an important step as all subsequent operations depend on this one.

    b) With the key generation verified to work correctly, the signing process was implemented, making it compatible with both the already verified communication methods and the cryptographic library employed.

    c) Finally, both the key generation and signing were integrated to verify if the threshold signature scheme could be completed with a valid final signature being computed at the end.

Throughout implementation, logging and debugging were used to monitor the correct execution of cryptographic operations and communication between devices.

5.5 Practical implementation

Following these decisions the practical implementation will now be covered. All cryptographic operations were implemented using the secp256k1 library [Ban22], an optimized C implementation designed for embedded systems that supports elliptic curve operations on the secp256k1 curve, a 256-bit curve that is widely adopted in Bitcoin and other cryptographic applications due to its strong security properties and computational efficiency. In this section we cover the most important aspects of the implementation, and the complete implementation is publicly available at PONE Biometrics' GitHub repository [Gar25a].

5.5.1 Key generation

Before the signing, the computer and the participant devices must carry out a key generation process to set up all the required elements. To do that, the computer is in charge of generating all the values needed for the signing and distributing them to the participating devices, acting as a trusted dealer.

It is important to note that participants must complete the key generation process before proceeding to registration (Section 2.2.4) in an authentication system, which in turn must be completed before any signing operations can be performed. This ensures that all the required cryptographic material is properly generated and distributed before any authentication attempt.

Computer side

To generate the required values, the secp256k1 library [Ban22] allows to create the context and generate a secret using the function `secp256k1_frost_keygen_with_dealer`, which then divides the secret into different shares based on a defined number of total participants N and the threshold T of them needed to compute a valid signature. Therefore, at the start of the code, some variables and structures are defined:

- `N` is the total number of participants, and `T` is the threshold of the needed participants to perform the signing.
- `sign_verify_ctx` is the context to perform signing and verification operations.
- `dealer_commitments` are the commitments that the dealer generates so each participant can verify the share they receive.
- `keypairs` is an array to hold the values:
  - `public_keys` is an array to store the individual public keys of each participant.
  - `shares_by_participant` is an array to store the secret share assigned to each participant.

```
#define N 3 // Number of participants
#define T 2 // Threshold of needed participants
secp256k1_context *sign_verify_ctx;
secp256k1_frost_vss_commitments *dealer_commitments;
secp256k1_frost_keygen_secret_share shares_by_participant[N];
secp256k1_frost_keypair keypairs[N];
secp256k1_frost_pubkey public_keys[N];
secp256k1_frost_signature_share signature_shares[N];

sign_verify_ctx = secp256k1_context_create(SECP256K1_CONTEXT_SIGN |
SECP256K1_CONTEXT_VERIFY);
dealer_commitments = secp256k1_frost_vss_commitments_create(T);
return_val = secp256k1_frost_keygen_with_dealer(
        sign_verify_ctx,
        dealer_commitments,
        shares_by_participant,
        keypairs,
        N,
        T
);
```

Listing 5.1: Key generation.

The obtained values are then shared among the participants. This distribution is performed using a communication interface that supports both UART Serial communication (Universal Asynchronous Receiver-Transmitter) for basic serial data exchange and USB HID communication (Human Interface Device protocol) for structured USB data transfer, offering compatibility with the two different boards used in the implementation. The data is split into different types of messages for more efficient communication. Moreover, when transmitting these messages, a number `MSG_HEADER_MAGIC` is defined to add more security, as to receive the data the receiving end should define the same number to establish the connection.

Then, a structure for the messages to be sent is defined, with the header containing information to help transmit the information more efficiently:

- `magic` being the number to secure the connection, with the receiving end having to define the same number to be able to receive the data.
- `msg_type` being a number to inform about the type of message that is transmitted, making it easier for the receiving end to know the message they are receiving and process it accordingly. The possible types are:
  - A secret share.
  - A public key.
  - Commitments for verifiable secret sharing.
  - A message to mark the end of the transmission.
- `payload_len` being the length of the payload of the message.
- `participant` being the ID corresponding to the participant whose information is being sent.

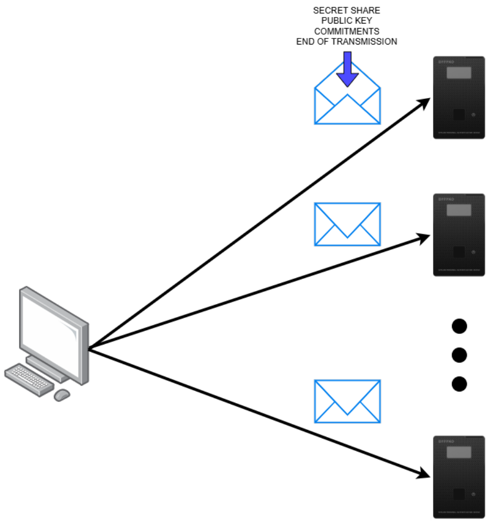


Figure 5.1: Message transmission to the boards.

Other structures are also defined to store the serialized data to transmit, ensuring proper data transmission across different communication channels. With all the data prepared, an interface was implemented to handle the supported communication methods, which are UART and USB HID. To use it, the program prompts the user to choose which communication method they want to use, and the user just needs to select the one that corresponds to the board they want to send the data to. This is done by pressing 1 or 2 on the keyboard, and in case of selecting UART communication, the program will also prompt the user to specify the COM port to use, which should be the one where the board is connected.

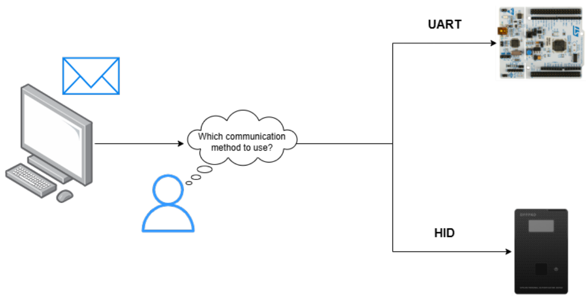


Figure 5.2: Communication method selection.

Then, a function is in charge of sending the data by combining the header with the payload into a single buffer before transmission. The payload containing the information to send must be one of the message types we have defined (a secret share, a public key, commitments, or a message to the end the transmission).

```
BOOL send_message(comm_handle_t* comm, uint8_t msg_type, uint32_t
    participant, const void* payload, uint16_t payload_len) {
    message_header header;
    header.magic = MSG_MAGIC;
    header.version = MSG_VERSION;
    header.msg_type = msg_type;
    header.payload_len = payload_len;
    header.participant = participant;
    size_t total_size = sizeof(header) + payload_len;
    uint8_t* combined_buffer = (uint8_t*)malloc(total_size);
    memcpy(combined_buffer, &header, sizeof(header));
    if (payload_len > 0 && payload) {
        memcpy(combined_buffer + sizeof(header), payload, payload_len);
    }
    BOOL result = send_data(comm, combined_buffer, total_size);
    free(combined_buffer);
    return result;
}
```

Listing 5.2: Function for message transmission.

Moreover, for UART communication, a function is needed to configure parameters for the serial port, which should have the same values for the transmitting and receiving sides, including:

– Baudrate of 115200 bits/s.
– 8 bits per character.
– One stop bit.
– No parity bit.

On the other hand, for USB HID communication, there is no need to specify the port where the board is connected; instead, the program searches through all available HID devices connected to the computer to find the one matching the specified Vendor ID and Product ID. These Vendor ID and Product ID also need to be configured in the receiving board with the same values. However, the transmission through USB HID requires special handling; the data is transmitted in chunks to fit within the device's output report size, with each chunk being prefixed with a report ID, the chunk length, and the actual data to ease the handling of them by the receiver. Working with a Windows device, the program uses the Windows HID API to transmit each chunk, continuing until all input data has been sent, which can be controlled thanks to the `payload_len` defined in the header, which defines the total length that the chunks should sum up to.

Finally, with the communication configured, all the data for the participants is generated and sent to each board with a loop that iterates through each participant, and all allocated resources are properly cleaned up to ensure security.

Board side

To receive the data generated by the computer, two codes are implemented, each of them specific to a different development board due to their communication compatibilities with either UART or USB HID.

UART: To receive the data with USB connection through serial communication the board needs to configure the corresponding Universal Asynchronous Receiver-Transmitter (UART) peripheral to the COM port used for the connection, as well as prepare the structures for the messages to be received, which need to be defined the same way they were in the transmitting side. Moreover, buffers to handle the messages are also configured to make the communication more efficient and decrease the burden on the board's resources.

The UART communication is configured with an interrupt-driven reception, which allows the handling of incoming data efficiently without blocking the main application. To handle the incoming data at the UART, the code makes use of interruptions, storing the received bytes into a buffer for later processing. With `dev` being a pointer to the UART device that triggered the interrupt, a while loop ensures all pending interrupt conditions are handled before exiting, and when a FIFO queue, which is a data structure that processes elements in "First In, First Out" order, has at least one byte to read, `uart_fifo_read` reads one byte from the FIFO queue and stores it in a buffer `rx_ring_buf` to later process the data.

```c
static void uart_cb(const struct device *dev, void *user_data) {
    uint8_t byte;

    while (uart_irq_update(dev) && uart_irq_is_pending(dev)) {
        if (uart_irq_rx_ready(dev)) {
            while (uart_fifo_read(dev, &byte, 1) == 1) {
                ring_buf_put(&rx_ring_buf, &byte, 1);
            }
        }
    }
}
```

Listing 5.3: UART interrupt handler.

With the structures and buffers prepared, the main reception loop processes the incoming data. It toggles between two different states, first, `WAITING_FOR_HEADER`, where it checks the buffer to see if there is any header available, if so, it obtains the information from the header about the message and validates the magic number for communication to then switch to `WAITING_FOR_PAYLOAD`. In this second state, the program reads the payload data until the full message is obtained and then processes it to switch back to the first state at the end.

Functions to process the different message types are defined. Thanks to the headers in the messages, the code knows which type of message is received and can process it to fill the structures required for the threshold signature protocol, which includes `keypair` and `commitments`.

```c
static void process_public_key(const uint8_t *payload, uint16_t len) {
    uint8_t *ptr = (uint8_t *)payload;
    memcpy(&keypair.public_keys.index, ptr, sizeof(uint32_t));
    ptr += sizeof(uint32_t);
    memcpy(&keypair.public_keys.max_participants, ptr, sizeof(uint32_t)
    );
    ptr += sizeof(uint32_t);
    memcpy(keypair.public_keys.public_key, ptr, 64);
    ptr += 64;
    memcpy(keypair.public_keys.group_public_key, ptr, 64);
}
```

Listing 5.4: Public key processing.

When the end transmission message is received, the system ends the key reception process and stores everything. To be able to use the received data in the signing, it is stored in the flash memory of the board for later use. To do that, a partition in the memory, `storage_partition`, is defined for storing the data, and the function `flash_are_open` gives access to the flash area for read/write/erase operations, being `fa` a pointer to the flash area. First, a structure `flash_data` is filled with the received data, following this, a page of the flash memory is erased with `flash_area_erase`, and lastly, a write buffer is configured with the correct size and the data is written with `flash_area_write`, closing the flash at the end with `flash_area_close`. For this operation to be successful, it is important that the flash page of the partition is fully erased and then ensure that the data is stored correctly, which is done thanks to the function `flash_area_read` that can check the values stored.

```c
int write_frost_data_to_flash(void) {
    const struct flash_area *fa;
    int rc = flash_area_open(FIXED_PARTITION_ID(STORAGE_PARTITION), &fa
    );

    frost_flash_storage flash_data = {0};

    // Copy all keypair and commitment data to flash structure
    flash_data.keypair_index = keypair.public_keys.index;
    memcpy(flash_data.keypair_secret, keypair.secret, 32);
    memcpy(flash_data.keypair_public_key, keypair.public_keys.
    public_key, 64);

    // Erase flash area and write data with proper alignment
    size_t write_block_size = flash_get_write_block_size(flash_dev);
    size_t padded_size = ROUND_UP(sizeof(frost_flash_storage),
    write_block_size);

    rc = flash_area_erase(fa, 0, erase_size);
    rc = flash_area_write(fa, 0, padded_buf, padded_size);

    flash_area_close(fa);
    return rc;
}
```

Listing 5.5: Flash storage writing.

USB HID: To receive the generated data in the other development board, the implementation includes this communication method, which works by reassembling chunked data that was segmented and sent by the computer to fit the device's report size. To do so, the device is first configured as a custom HID interface with specific report descriptors so it can be easily detected and identified by the computer, including:

- HID_USAGE_PAGE: Specifies that the used device belongs to the "Generic Desktop" category (standard for mice, keyboards, joysticks).
- HID_USAGE: Set to undefined as it doesn't fit a standard HID device.
- HID_COLLECTION: Groups related HID items together to inform the device that all of them work together as a single functional unit.
    - HID_REPORT_ID: To identify the report type.
    - HID_REPORT_SIZE: To specify that each byte is 8 bits.
    - HID_REPORT_COUNT: To specify that there are a total of 63 bytes. The report will be 64 bytes total: 1 byte for the report ID and 63 bytes of payload data.
    - HID_OUTPUT: To specify the direction of the communication, in this case, a host-to-device communication.

Unlike UART, which can handle variable-length streams, USB HID uses reports of a fixed length. Therefore, the implementation includes a way to handle messages that are larger than the HID report size and reconstruct the full message. When it finds a new message, it calculates the expected total size and starts to build up chunk data in a reassembly buffer, keeping track of the progress by means of the variable `reassembly_pos`. When all the expected bytes have been gathered and the original message can be reconstructed (`reassembly_pos` $\geq$ `expected_total_size`), it then handles the full message and resets the state.

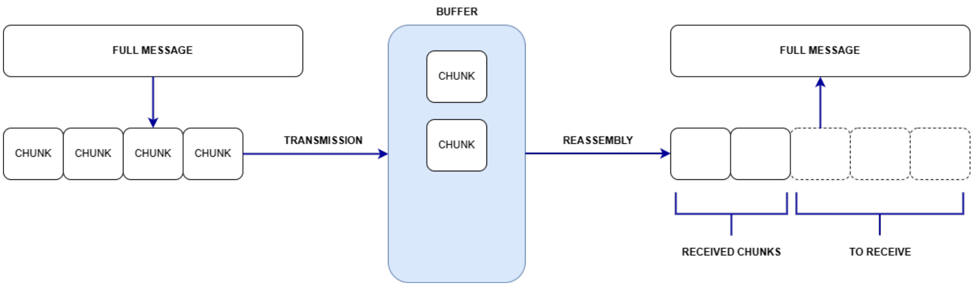


Figure 5.3: Message transmission and reception with reassembly buffer.

A callback function is in charge of handling the incoming data and USB events. The function `int_out_ready_cb()` is triggered when the HID device receives data, reading data up to 64 bytes using `hid_int_ep_read()` and passing any successfully read data to `handle_chunked_data()` for reassembly. The status changes are managed by `status_cb()`. In status USB_DC_RESET, the flag is reset and the data cleared, while on USB_DC_CONFIGURED, the device is marked as ready to receive data.

```c
static void int_out_ready_cb(const struct device *dev) {
    uint8_t buffer[64];
    int ret, received;
    ret = hid_int_ep_read(dev, buffer, sizeof(buffer), &received);
    if (ret == 0 && received > 0) {
        handle_chunked_data(buffer, received);
    }
}

static void status_cb(enum usb_dc_status_code status, const uint8_t *
    param) {
    switch (status) {
    case USB_DC_RESET:
        configured = false;
        reset_all_data();
        break;
    case USB_DC_CONFIGURED:
        if (!configured) {
            configured = true;
        }
        break;
    }
}
```

Listing 5.6: USB HID callbacks.

The received messages are processed as in the UART implementation, based on the type of the message that is marked in the header. When all the required data is received, it is stored in the flash memory, same as in the UART implementation, but using Zephyr's work queue system to prevent blocking the USB interrupt handlers.

All these functions are then used together to process the full message transmitted from the board, receiving it on the boards through the corresponding communication and storing the received data in the flash memory, leaving the receiving board prepared with the key material for the signing.

To summarize, in this implementation we provide an efficient way for the generation and distribution of key material required for threshold signatures, thanks to using the FROST protocol. A device acts as a trusted dealer, creating a secret key and distributing secret shares and the corresponding public keys and commitments to various devices via USB. On the board side, each board receives the information through either UART or HID, verifies if it is correct, and saves it in the flash memory for later use in the singing process. This setup ensures all participants end up holding a verified piece of the secret key so they can later participate in the signing process to collaboratively generate a valid Schnorr signature.

5.5.2 Signing

With the key generation process complete and each participant holding the necessary data, a signing process can then be carried out. As explained in previous sections (Section 4), this phase requires a coordinator, a role that is taken by a computer, which will coordinate the communication and take the signature aggregator role from the FROST protocol (Section 3.3), and a threshold T of devices to participate.

Computer side

To perform the signing, the computer prepares the required structures and defines the different types of messages that will be transmitted and received, as in the key generation. In this implementation, the computer is in charge of coordinating the communication, facilitating the data exchange of the participants so they can carry out the required cryptographic operations for the protocol, and acting as a signature aggregator at the end, gathering all the signature shares, aggregating them, and computing the final signature. Following the FROST scheme, the signing consists of two rounds of communication:

Round 1:

1. The computer sends a ready signal to trigger the nonce generation in the boards.
2. The computer collects the nonce commitments and keypair information, excluding the secret information that should not be shared, such as the secret share.
3. The computer verifies the commitments and stores them in an array.

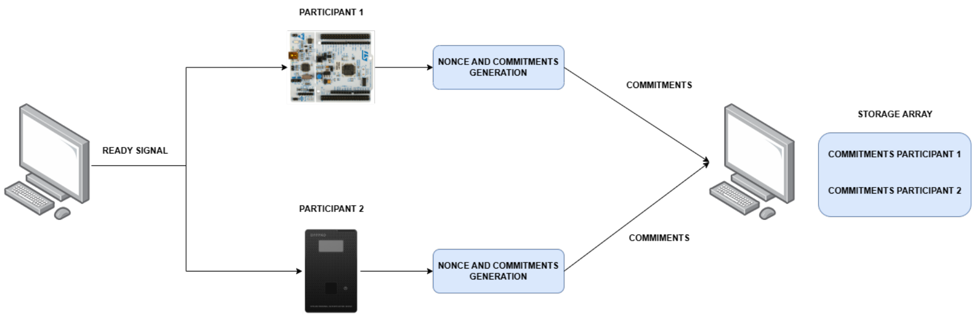

PARTICIPANT 1
READY SIGNAL
NONCE AND COMMITMENTS GENERATION
COMMITMENTS
STORAGE ARRAY
COMMITMENTS PARTICIPANT 1
COMMITMENTS PARTICIPANT 2
PARTICIPANT 2
COMMIMENTS
NONCE AND COMMITMENTS GENERATION

Figure 5.4: FROST's first round of communication.

Round 2:

1. The computer sends a hash of the message to sign and the array containing all the collected commitments to each board.

2. The computer collects the signatures shares computed by the boards and aggregates them into the final signature.

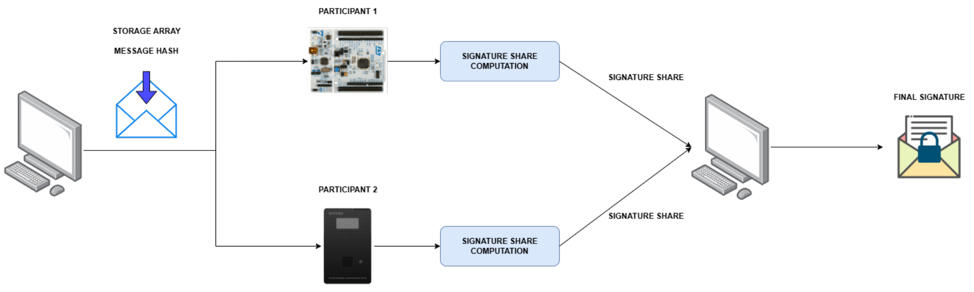


Figure 5.5: FROST's second round of communication.

This approach, with the use of a computer as a coordinator/signature aggregator, requires the computer to perform several tasks. This reduces the burden on the embedded devices, as the computer is the one performing the most computationally demanding operations and facilitates the communication of the participants, ensuring that all the operations are performed correctly, while the used boards only need to focus on some computations.

Board side

Same as with the computer, the boards need to prepare the structures and define the different types of messages that are required for the signing. The communication follows the same implementation as in the key generation, with two independent codes, one for each development board to work with either UART or USB HID communication due to their hardware limitations. Each board configure their corresponding parameters for communication and handle the messages received based on the type marked in the header, with the communication method changing based on the board designated to receive the data. Again, two rounds of communication are carried out for the signing:

Round 1:

1. The board receives a ready message and computes nonces and commitments with the data stored in flash memory.

2. The board sends the commitments and key pair information to the computer, omitting the data that should remain secret.

```c
static void process_ready_message() {
    LOG_INF("READY signal received ");

    if (generate_and_save_nonce() == 0) {
        send_nonce_commitment_and_keypair();
    }
}

static int generate_and_save_nonce(void) {
    current_session_id = sys_rand32_get();

    unsigned char binding_seed[32] = {0};
    unsigned char hiding_seed[32] = {0};
    fill_random(binding_seed, sizeof(binding_seed));
    fill_random(hiding_seed, sizeof(hiding_seed));

    secp256k1_frost_nonce* nonce = secp256k1_frost_nonce_create(ctx, &
    keypair, binding_seed, hiding_seed);

    int save_result = save_nonce_to_flash(nonce, current_session_id);
    secp256k1_frost_nonce_destroy(nonce);

    return save_result;
}
```

Listing 5.7: Ready message processing and nonce generation.

Round 2:

1. The board receives a hash of the message to sign and an array with commitments from all the boards involved in the signing, and verifies if they are correct.

2. The board computes a signature share using the received data and the nonce computed before, which was stored in the flash memory.

3. The board sends the signature share to the computer.

```
// Load nonce
    secp256k1_frost_nonce* nonce = load_nonce_from_flash(
    current_session_id);

    // Prepare commitments array for signing
    secp256k1_frost_nonce_commitment *signing_commitments = k_malloc(
    num_commitments * sizeof(secp256k1_frost_nonce_commitment));

    for (uint32_t i = 0; i < num_commitments; i++) {
        signing_commitments[i].index = serialized_commitments[i].index;
        memcpy(signing_commitments[i].hiding, serialized_commitments[i
    ].hiding, 64);
        memcpy(signing_commitments[i].binding, serialized_commitments[i
    ].binding, 64);
    }

    // Compute signature share
    memset(&computed_signature_share, 0, sizeof(
    computed_signature_share));
    int return_val = secp256k1_frost_sign(&computed_signature_share,
    msg_hash, num_commitments, &keypair, nonce, signing_commitments);

    if (return_val == 1) {
        signature_share_computed = true;
        send_signature_share();
    }

    k_free(signing_commitments);
    k_free(nonce);
}
```

Listing 5.8: Signature share computation.

These two rounds are implemented in the same code, and the boards are the ones that start each round based on the message received by the computer, beginning with the first round after receiving a ready message, or with the second after receiving the hash of the message with the array of commitments. All this is used together with an interactive interface, such as the one in the key generation, that allows choosing the communication method to use with each board.

To sum up, thanks to this implementation we allow each participating board in the signing to compute a signature share thanks to the secret share they received in the key generation, and these signature shares are collected by a computer, which then aggregates them to obtain the final signature without revealing secret information in the process, therefore, providing security.

5.6 Testing and evaluation

The evaluation methodology followed was designed to analyze the practical implementation of the FROST threshold signature scheme in embedded devices. The evaluation focused on four main aspects: time required, memory usage, protocol overhead, and communication behaviour.

The performance was measured using high-accuracy timing techniques, with system performance counters on the coordinator device and specific functions on the used boards, which allowed us to obtain millisecond accuracy for measuring the time taken to perform cryptographic operations.

During key generation and signing processes, memory usage was constantly monitored to analyze overhead and consumption. The analysis measured the starting memory usage and calculated the increase to determine memory efficiency for the implementation.

The protocol size analysis required all data elements exchanged during the FROST processes to be measured, including message headers, public keys, commitments, secret shares, and signature components. This evaluation determined the storage and transmission requirements that are relevant to the implementation.

Lastly, the effectiveness of communication was evaluated by measuring message transmission times on different communication channels: USB HID and UART. This allowed us to determine the performance of the communication in the two development boards used when coordinating with a computer.

5.7 Tools and resources

Different tools and resources were used to ensure efficiency and quality performance during the project's implementation phase. These tools and resources played a crucial role in various aspects of the development.

5.7.1 STM32 Nucleo-64 development board

The STM32 Nucleo-64 development board was used together with the OFFPAD devices due to its similarity in hardware when compared with the OFFPAD, especially in its processor, using an Arm Cortex M4. This development board is model Nucleo-L476RG, using a STM32L476RG Microcontroller Unit (MCU), which facilitated the development and implementation of new code for the integration of threshold signatures in the OFFPAD devices.

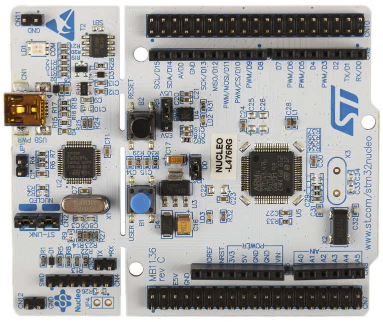


Figure 5.6: Nucleo L476RG board.

5.7.2 OFFPAD development board

The OFFPAD development board was used to verify that the implementation worked for the OFFPAD devices. This board is based on an STM32WB5x/35xx board, and during the development and implementation of the code, it was programmed using the STM32 Nucleo-64 development board.

5.7.3 Zephyr OS

The Operating System used by both the OFFPAD and STM32 Nucleo-64 Development Board is Zephyr [Zep23]. It is an open-source, real-time operating system (RTOS) developed by the Linux Foundation [Lin25] to build secure, connected, and future-proof devices.

5.7.4 C programming language

When working with embedded systems C programming language is one of the standardized programming languages. It is a procedural language offering modularity with plenty of libraries developed by programmers, while also being fast and efficient. For this reason, the programming language used for this project is C, with Zephyr OS being compatible with both C and C++ programming languages.

5.7.5 Libraries

To be able to work on implementations that use both Schnorr or ECDSA signatures, an optimized C library was used [Ban22], enabling to perform public/secret key operations in the curve secp256k1. It enables to perform different type of operations such as:

- Field operations
- Scalar operations

- Modular inverses
- Group operations
- Point multiplication for verification
- Point multiplication for signing

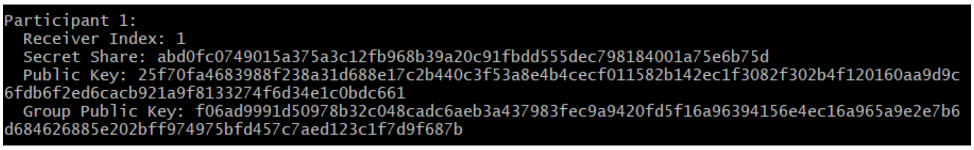
This library is an optimized implementation in C of the secp256k1 library [Bit14], developed by the Bitcoin Core developers. It was originally designed to perform efficient and secure ECDSA operations on the secp256k1 curve, but it has been extended to support Schnorr signatures. This effort to extend the library is part of the Banca d'Italia's "itcoin" project, which aims to research solutions for digital payments and decentralized agreement mechanisms. By integrating threshold signatures with FROST, the Banca d'Italia seeks to improve security and robustness in its protocols.

Chapter 6
Results and discussion

In this chapter we offer a full analysis of the threshold signature implementation we developed for the OFFPAD's infrastructure using the FROST scheme. We first show a walkthrough of the functional implementation, specifying how the practical execution of a 2-out-of-3 threshold signature is obtained, with illustrations of the system output and interface. Second, we provide an analysis of the implementation with extensive testing and measurements of key metrics like execution time, memory consumption, and sizes of data. Next, we describe the limitations we faced in the project, which include issues of hardware communication, compatibility with libraries, performance limitations, and security model weaknesses that affected the implementation. Finally, we provide an analysis of all the findings and discuss all the obtained results and their potential application and performance in real-world applications.

6.1 Schnorr implementation

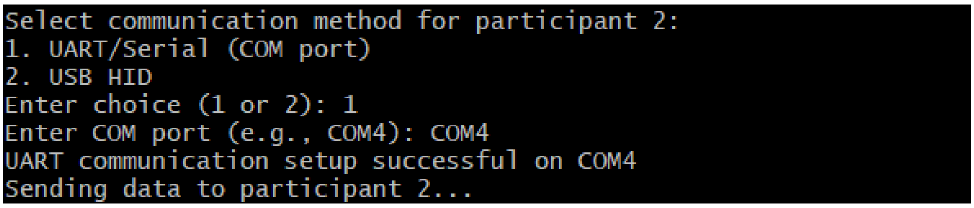
Following the FROST scheme, we achieved a 2-out-of-3 threshold signature using Schnorr signatures, meaning that secret shares are generated and distributed to 3 total participants, and 2 out of them are required to participate in a signing process to obtain a valid signature. As explained before (Section 5.5.1), the implementation uses a trusted dealer approach, where the shares and key material are generated in a computer that then sends this data to each corresponding participant. This can be observed in Figure 6.1, where the data generated for one participant can be observed.



```
Participant 1:
  Receiver Index: 1
  Secret Share: abd0fc0749015a375a3c12fb968b39a20c91fbdd555dec798184001a75e6b75d
  Public Key: 25f70fa4683988f238a31d688e17c2b440c3f53a8e4b4cecf011582b142ec1f3082f302b4f120160aa9d9c
6fdb6f2ed6cacb921a9f8133274f6d34e1c0bdc661
  Group Public Key: f06ad9991d50978b32c048cadc6aeb3a437983fec9a9420fd5f16a96394156e4ec16a965a9e2e7b6
d684626885e202bff974975bfd457c7aed123c1f7d9f687b
```

Figure 6.1: Key generation in a computer.

The generated shares are distributed to the participants using a USB connection through two different communication methods. The STM32 Nucleo-L76RG development board is only compatible with UART communication, and the OFFPAD development board is only compatible with USB HID. Therefore, the user is allowed to select the communication method to send the data with, and in case of using the UART one, to select the serial COM port where the board is connected, as can be observed in the example from Figure 6.2.

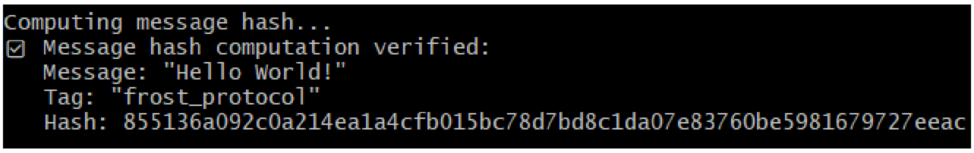


```
Select communication method for participant 2:
1. UART/Serial (COM port)
2. USB HID
Enter choice (1 or 2): 1
Enter COM port (e.g., COM4): COM4
UART communication setup successful on COM4
Sending data to participant 2...
```

Figure 6.2: Key distribution to participant through UART.

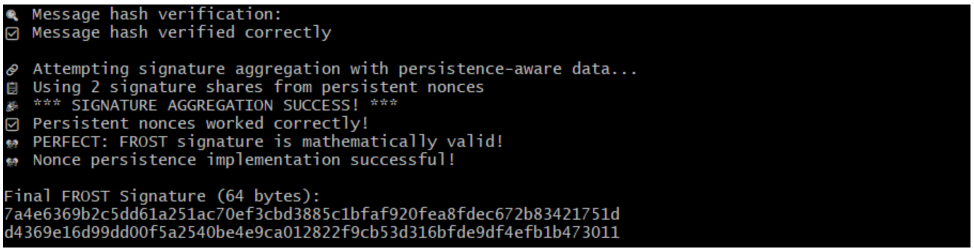
These shares are stored on each board, so when needing to sign a message, 2 out of 3 boards that hold a share need to participate to compute a valid signature for the message. With the key generation completed, the signing can be called at any time. It is the computer, acting as a coordinator, that starts it, sending a message to the participants so they can start generating the nonces and commitments required, with the commitments being collected by the computer to complete the first round of communication required. When the defined threshold of 2 devices have sent their nonces, the computer passes to the next round of communication.

For the second round, as seen in the implementation, to test the signing, the message 'Hello World' is the one signed. This message, as shown in Figure 6.3, is hashed and shared with the commitments to each participating device that was involved in the first round. These devices then compute a signature share and send it to the computer, which aggregates them, obtaining a signature that is verified to check if it is correct, and if everything went as expected, an output like the one shown in Figure 6.4 should be observed.



```
Computing message hash...
☑ Message hash computation verified:
  Message: "Hello World!"
  Tag: "frost_protocol"
  Hash: 855136a092c0a214ea1a4cfb015bc78d7bd8c1da07e83760be5981679727eeac
```

Figure 6.3: Message to sign in FROST.



```
🔍 Message hash verification:
☑ Message hash verified correctly

🔗 Attempting signature aggregation with persistence-aware data...
📋 Using 2 signature shares from persistent nonces
🎉 *** SIGNATURE AGGREGATION SUCCESS! ***
☑ Persistent nonces worked correctly!
🎊 PERFECT: FROST signature is mathematically valid!
🎊 Nonce persistence implementation successful!

Final FROST Signature (64 bytes):
7a4e6369b2c5dd61a251ac70ef3cbd3885c1bfaf920fea8fdec672b83421751d
d4369e16d99dd00f5a2540be4e9ca012822f9cb53d316bfde9df4efb1b473011
```

Figure 6.4: Completed signature aggregation in FROST.

6.2 Evaluation and testing

For evaluating the practical implementation of the FROST scheme into the OFF-PAD, we performed an in-depth analysis for both the key generation and signing processes. An evaluation with different tests performed targeted to investigate some fundamental aspects: time analysis, memory usage, protocol overhead with data size, and communication efficiency.

6.2.1 Key generation evaluation

The FROST key generation performance evaluation demonstrated the computational efficiency of threshold cryptography using a 2-out-of-3 scheme.

Time measurements

Performance tests used very precise Windows timing methods to measure the computational costs related to FROST key generation operations. The timing approach uses the system performance counter, which allows for millisecond accuracy to measure cryptographic operations. The `get_time_milliseconds()` function uses `QueryPerformanceCounter` for millisecond precision timing, ensuring accurate measurements. This is key as FROST key generation operates fast, requiring high-precision measurement to capture meaningful performance data.

This time measurement was employed in an analysis performed using 50 runs to obtain statistical measures that summarize the performance and reliability from the FROST deployment. The most important statistics include:

- Mean execution time of 107 milliseconds. This indicates the expected time for key generation under normal circumstances.
- Standard deviation of 9 milliseconds. This is a measure of the dispersion from the mean, and being small relative to the mean indicates a stable performance.

- Execution time interval ranging from 102 milliseconds to 144 milliseconds. This defines optimal and worst-case performance results.
- Coefficient of variation of 8.41%. This coefficient $CV = \sigma/\mu$ shows a moderate level of variability in execution times. The variation comes from different factors, including memory allocation during context initialization, system scheduling effects, CPU caching, as well as variations in elliptic curve operations.

Memory measurements

Memory monitoring revealed efficient resource handling during the complete FROST key generation. The analysis continuously monitored memory usage to evaluate the maximum consumption and overhead.

The results showed an initial memory usage of 6200 KB, a maximum memory usage of 6404 KB, and an overhead of 204 KB, representing a 3.3% increase. This low memory usage is really significant as it includes all the operations for key generation and distribution, including the setup of secp256k1 contexts, verifiable secret sharing commitments, and the temporary computation of buffers.

Protocol sizes

The elements used in the protocol were measured to determine the storage and transmission requirements for the implementation used. The protocol components include a message header of 12 bytes, a public key of 136 bytes, and commitments of 232 bytes, making the total data per participant 452 bytes. Additionally, each participant required 36 bytes for secret shares and additional protocol data, summing up to 368 bytes, including public keys and commitments.

Communication measurements

Some measurements were performed to estimate the message transmission times and analyze the burden of sending threshold cryptographic information to participants. In the execution of a key generation process, USB HID transmission for one participant showed:

- Secret share (36 bytes) was transmitted in 102 milliseconds.
- Public key (136 bytes) was transmitted in 324 milliseconds.
- Commitments (232 bytes) were transmitted in 433 milliseconds.

These times are related to how messages are sent, having to segment large messages into reports of 64 bytes, with an overhead of 2 bytes per report, leaving a payload of 62 bytes.

For UART communication in a key generation process for one participant results showed:

- Secret share (36 bytes) was transmitted in 1 millisecond.
- Public key (136 bytes) was transmitted in 5 milliseconds.
- Commitments (232 bytes) were transmitted in 11 milliseconds.

This communication showed much faster transmission as the messages do not need to be segmented, and with the communication being configured with a baudrate of 115200 bits/s.

6.2.2 Signing evaluation

The FROST signature generation evaluation demonstrated the computational efficiency of the signing process through the use of a 2-out-of-3 threshold scheme.

Time measurements

Performance tests used systematic and careful timing techniques to measure the time needed for the execution of the scheme. The timing used system performance counters on the coordinator, as in the key generation, and `k_uptime_get()` on the development boards, to provide millisecond accuracy.

The signing process was analyzed across two main phases, which correspond to the two rounds of communication: nonce generation and commitments, and signature aggregation. The anaylisis was performed for both the coordinator and the two development boards used, some of the most important statistics include:

- Signature share average computation time in the development boards of ranged from 503 milliseconds to 523 milliseconds per participant. This indicates the expected time needed for creating a signature share under normal conditions in embedded devices.
- Signature aggregation time in the computer of 2 milliseconds. This represents the time required for the coordinator to combine the signature shares of two participants into the final signature.
- Total signing process time varied from 25 seconds to 50 seconds. This includes the gathering of nonce commitments and the signature share computation of all the participants. The results showed high execution times, but the majority of this time was related to the manual USB connection management required during testing. Since both boards used USB connectivity, the experimental setup required physically disconnecting and connecting each device.

Memory measurements

The memory monitoring throughout the entire signing process showed efficient use of resources. The tests continuously monitored memory usage during the computation and aggregation of signatures, as done during key generation.

The results showed that the initial memory usage was 6108 KB, and the maximum memory usage was 6836 KB, with an overhead of 728 KB, corresponding to an increase of 11.9%. This memory usage involves operations related to computing signatures, fetching nonces from flash storage, verifying commitments, and processes involved in signature aggregation.

Protocol sizes

The protocol elements used for signing were analyzed to determine the storage and transmission needs of the implementation. These elements include:

- Nonce commitments of 132 bytes per participant, consisting of 66 bytes for binding and hiding commitments. These commitments are generated in the first round of communication and must be collected to be shared amongst all participants.
- Commitments array of 264 bytes containing the commitments collected from two participants. This array is transmitted to each participant in the second round of communication.
- Signature shares of 36 bytes, each representing the contribution of each participant to the final signature.
- Final signature of 64 bytes produced by the coordinator through the aggregation of the signature shares.

Communication measurements

Communication efficiency was tested for the transmission of signature data and the gathering of signature shares between the coordinator and the participants. When carrying out the signing process, the time for USB HID transmission concerning the transmitted data, which includes the message hash, nonce commitments and additional metadata (300 bytes), was measured to be 1.788 seconds, while the collection of the signature shares (36 bytes each) took 1.223 seconds.

For UART communication, transmission concerning the same transmitted data (300 bytes) required 521 milliseconds, while reception of signature shares (36 bytes each) took 526 milliseconds. This communication showed again better speeds due to the lack of segmentation limitations and the use of a baudrate of 115200 bits/s.

6.3 Limitations

During the project, several limitations needed to be taken into account, affecting the decisions and implementation adopted. These limitations ranged from problems with the hardware specifications of the devices used to problems with compatibility with the OS used. A more detailed explanation of these limitations is essential for a better understanding of the decisions made in the project.

6.3.1 Hardware limitations

One of the boards used in the implementation, the STM32 Nucleo-L76RG development board, had significant communication constraints that impacted the decisions and implementation adopted in the project. The board features a single ST-LINK virtual COM port, which was mainly used for debugging purposes. Due to this limitation, carrying out communication and debugging purposes was not possible at the same time, as no other port was available, forcing to switch between device communication and debugging support. This restriction made testing and development more difficult, as it was not possible to monitor the system behaviour while performing communication activities.

To overcome this limitation, an auxiliary COM port was added to the board, attaching it through the pins of the board and assigning a different UART node to it, using the different UART channels for communication and debugging. This setup allowed for simultaneous debugging and communication, but introduced another level of hardware complexity and points of failure.

The STM32 Nucleo-L76RG development board also lacked onboard Bluetooth functionality, which would have been more suitable for the communication infrastructure of the OFFPAD. Providing Bluetooth connectivity would have required the use of external modules, but the incorporation of new modules was considered to be out of the scope of the project due to time limit and integration complexity. Due to these limitations, the communication was restricted to a USB interface, limiting the portability and usability in real-world scenarios.

Furthermore, due to having boards with different communication protocols, one constraint during the implementation phase was the basic incompatibility between the two communication channels of the two development boards used in the project. The STM32 Nucleo-L476RG development board only supported UART communication, while the OFFPAD development board, based on the STM32WB5x/35xx architecture, only supported the USB HID communication protocol and had no capability for UART communication.

This incompatibility required us to develop two completely different implementations of communication for a threshold signature scheme. Having this duplicate communication infrastructure significantly increased the code's complexity, such that each board needed customized message handling procedures. This prolonged the development process and compromised the reliability of the implementation, as having two communication methods creates more potential weaknesses.

The OFFPAD development board also presented some restrictions on its programming. Unlike other development boards that allow direct programming, the OFFPAD required programming through another board, in this case, the STM32 Nucleo-L76RG development board, which made development and testing more difficult. This led to longer times for deployment and testing, as each new code implementation required careful control of the physical interconnections between the boards and careful handling of the programming process.

In addition, the use of a board to program the other required to create a communication interface, which enabled the system to automatically determine the correct communication channel for every interaction with the board. The interface needed to determine the specific board in use and the corresponding communication protocol, whether using UART for the STM32 Nucleo-L76RG development board or USB HID for the OFFPAD development board. This added complexity to the system and demanded additional user involvement in manually setting the communication.

We were also limited to the use of just two development boards in both the implementation and testing phases, which prevented a thorough examination of situations related to device failures, network disruptions, or varied threshold settings. This limited the system to a 2-out-of-2 or 2-out-of-3 threshold signature scheme, selecting the latter one for the project. This limited the investigation of different threshold signature approaches.

6.3.2 Zephyr OS library compatibility

One big limitation faced from the start of the project was the lack of threshold signature libraries specific to Zephyr OS, the operating system used by the OFFPAD. The selected cryptographic library contained functions that changed depending on the operating system where they are executed, being optimized for conventional operating systems like Windows and Linux, but not having adaptations for embedded systems running within the Zephyr framework.

This incompatibility required some important changes in the library to ensure the correct operation on the boards used in the implementation. To solve this, the library was adapted, changing basic system calls, memory management, and the implementation of cryptographic primitives to match the architecture and constraints of Zephyr.

6.3.3 Communication and performance limitations

Threshold signature schemes require many exchanges of information between the involved devices, leading to higher communication overhead compared to traditional single-device authentication schemes. While the schemes covered in the project (Section 3.3 and Section 3.4) are optimized to improve round efficiency, they still require devices to exchange coordination messages, which introduces more latency in the system.

The chosen communication model, where devices send signature shares to a coordinator/signature aggregator without communication directly between each other, was partially selected to address these limitations. However, this still caused significant delays in the authentication process.

Moreover, the usage of threshold signatures leads to larger message sizes compared to standard single-device signatures since it requires the transmission of signature shares, commitments, and verification data. These increased message sizes require additional bandwidth and longer processing times, which can be a serious problem in bandwidth-limited environments or where multiple authentication operations are performed concurrently. Even though this supposes a limitation, it represents a trade-off between enhanced security and system performance.

The limited processing power of the used development boards placed significant limitations on the complexity of cryptographic operations that could be performed efficiently. Threshold signatures require more computational power compared to traditional signatures, which include share generation, computations related to commitments, and reconstruction of signatures. Though operations using elliptic curves provide better computational efficiency over those that use RSA, they are still computationally demanding on embedded systems.

For this reason, a computer was designated to not only handle signature aggregation but also the most computationally demanding operations, including key generation, secret share distribution, and coordination of communication in the signing. This decision allowed the reduction of the computational burden on the embedded devices. The implementation also required careful control of memory allocation to avoid overflow situations, which is due to the limited memory size of embedded systems.

6.3.4 Security model limitations

The established communication model is based on the role of a trusted device that is tasked with managing the threshold signature process. While this device cannot generate signatures on its own or obtain the secret key, it represents a potential weak point, limiting the fully decentralized security threshold that signatures are supposed to provide. This was necessary due to the difficulties inherent in communication and coordination, and the hardware limitations of the boards used.

6.3.5 FIDO2 protocol compatibility

One important limitation of using the FROST protocol for threshold signatures is that Schnorr signatures are not FIDO2-compliant. The FIDO2 protocol only supports RSA and ECDSA signature schemes for authentication mechanisms, making the implemented solution incompatible with the existing FIDO2 infrastructure and limiting its integration into real-world use cases.

While the project scope initially included the development of a threshold signature scheme using ECDSA signatures, time constraints and the complexity of ECDSA threshold signature schemes, requiring more rounds of communication and higher computational capacity, led to the project focusing on a Schnorr signature implementation. To solve this, we researched a scheme based on ECDSA to pave the way for a future implementation that integrates threshold signatures in the OFFPAD's authentication mechanism, being FIDO2-compliant.

6.4 Discussion

In this project, we implemented a threshold signature scheme with the FROST protocol on OFFPAD devices. The results prove the feasibility of integrating threshold cryptography into an authentication mechanism on embedded devices, improving the security by having a decentralized authentication. However, many of the key aspects of the implementation and the decisions made require detailed discussion.

6.4.1 Scope limitations and strategic decisions

A critical limitation of the project's implementation needs to be addressed: the final system is not FIDO2-compliant. At the start, the project scope included the integration of both FROST-based and ECDSA-based threshold signatures schemes, with ECDSA being the only FIDO2-compliant. Due to time constraints and the complexity of the ECDSA schemes, requiring more rounds of communication and higher computational capacity, it was decided to focus on the FROST implementation.

The idea of this decision was to first focus on FROST, and when completed, move to the ECDSA implementation. Both schemes were researched, performing a literature review on them, but only FROST was implemented at the end, as it took more time than expected, with the need to work on different communication methods adding more time for the development of the functional implementation. Therefore, ECDSA is included in this project as a future work implementation following the scheme presented (Section 3.4).

6.4.2 FIDO2 compliance implications

With Schnorr signatures not being compliant with FIDO2, the project's implementation is not compatible with FIDO2 authenticators, which makes it impossible to integrate it into existing FIDO2 infrastructures. This limits the application of the implementation in real-world scenarios where the standard supports RSA and ECDSA schemes.

However, even with this limitation, as seen in the project, Schnorr signatures have great potential that justifies the decision to choose them for this project (Section 3.3.4). The cryptographic advantages of the FROST protocol offer security, robustness, efficiency, and flexibility, which is promising for authentication systems.

6.4.3 Performance analysis and practical implications

The results obtained from the evaluations and tests showed that threshold signatures with the FROST protocol can achieve acceptable execution times for authentication scenarios, as both key generation and signing present acceptable execution times for embedded applications.

The memory monitoring confirmed that threshold signature schemes can be implemented even with the constraints of embedded devices. The approach taken in the project includes a good handling of the memory requirements to fulfil all the requirements for the different cryptographic operations and rounds of communication.

Communication also proved to be consistent, presenting an inherent trade-off in threshold signatures, where the authentication takes more time but is proven to be more secure. The use of a computer as a coordinator also showed to minimize the computational burden from the used development boards and reduced the complexity involved in the coordination of them. The use of this computer as a coordinator would also facilitate communication with the authentication server.

6.4.4 Implementation challenges and solution

Several challenges were identified and solved during the project. The used library needed to be adapted to work with Zephyr OS, including the adaptation of system calls, memory management, and cryptographic primitives to work on embedded devices. Hardware communication limitations were addressed by developing separate communication channels for USB HID and UART. Memory management involved careful handling to use the flash memory of the development boards to store and retrieve the required data for the cryptographic operations of the scheme.

These modifications took more time than initially estimated, but they represent a valuable contribution to authentication in embedded devices, specifically in threshold signatures for authentication.

Chapter 7

Conclusions and future work

Based on the findings from the implementation, evaluation, and challenges we encountered during the project, in this section we aim to provide some conclusions and answers to the initial research questions and cover the future work that could be developed from them.

7.1 Conclusion

What is the best communication model for threshold signatures?

The project's implementation found the use of a coordinator model as an optimal approach to the integration of threshold signatures in embedded systems, especially for the OFFPAD's infrastructure, given its hardware limitations. In this model, devices exchange data with a computer acting as a coordinator without communicating directly with other participants. This coordinator shares the needed data among the participants and also has the role of a signature aggregator, who is in charge of collecting the signature shares to compute the final signature.

This approach provides some benefits in systems with limited computational capacity, like embedded devices, as the most demanding operations are carried out by a computer while security is maintained thanks to the use of threshold signatures. However, this model introduces a dependency on a trusted computer, something that must be considered in security assessments.

What is the total number of participants N, and the threshold T of devices that should participate to achieve a good balance between security and efficiency for the OFFPAD?

Even though the number of development boards for the project was a limitation to perform tests with different values for N and T, the 2-out-of-3 ($N = 3, T = 2$) approach showed a good performance. This approach provided fault tolerance against single device failures with a low level of coordination complexity for communication.

Tests and evaluation results proved good execution times and memory use, meaning that there is a good trade-off between security, robustness, and efficiency, where security and robustness are improved thanks to the use of threshold signatures without requiring much more time or computational costs.

Given these performance results, configurations with more participants could potentially be implemented and achieve acceptable trade-offs. However, the 2-out-of-3 scheme provides a good balance in the OFFPAD's environment, as adding more devices would provide more security, but would also require more time for the execution of the protocol, impacting the communication overhead and adding complexity to the communication model. Therefore, even though the 2-out-of-3 scheme we implemented showed good results, the evaluation of schemes with more participants to find the optimal setup will be an important part for future work.

What is the best threshold signature scheme for the OFFPAD? And the best FIDO2-compliant?

Among threshold signature schemes for embedded devices like the OFFPAD, the FROST protocol presents itself as a good choice with great potential due to its round efficiency, security properties, and adaptability to the computational capability of embedded systems. FROST's ability to minimize the number of communication rounds while maintaining strong security makes it well-suited for limited resources environments.

However, for FIDO2-compliant implementations, FROST cannot be employed due to being based on Schnorr signatures rather than ECDSA signatures, which are included in FIDO2 standards. Based on the literature review and research performed, threshold signature schemes using ECDSA, such as the one covered in this project (Section 3.4), are key to achieving FIDO2 compliance using threshold signatures. This scheme, while more complex to implement, represents the required path to achieve a true FIDO2-compliant threshold authentication.

What are the challenges in implementing the selected threshold signature scheme, and how can they be solved?

Several important implementation challenges were identified and addressed in the project. First, it was important to realise that most cryptographic libraries for threshold signatures need to be adapted to work with Zephyr OS. Second, hardware communication limitations required configuring different communication methods depending on the embedded devices used. Lastly, memory management required careful handling of the flash memory of the participants to store and retrieve the needed data for the protocol.

Furthermore, the most important challenge that remains unsolved is achieving FIDO2 compliance. Even though Schnorr signatures in FROST implementations provide great security and efficiency, only schemes using ECDSA or RSA would be compliant with FIDO2 authentication environments. For that reason, a protocol was covered and is included as a future line of work.

Final assestment

This project successfully demonstrates the feasibility of implementing threshold signatures on OFFPAD devices using the FROST protocol. While the limitation of not being FIDO2-compliant due to the priority of first implementing Schnorr signatures over ECDSA signatures, which restricts usability in practical use cases, the project provides a solid foundation for implementing distributed authentication in embedded environments.

The decision to focus on the FROST protocol enabled the implementation of an efficient and tested threshold signature scheme. The problems and limitations identified and the proposed solutions provide valuable knowledge to cryptography in embedded devices. Moreover, the research performed on threshold signature schemes that use ECDSA paved the way for future work on integrating threshold signatures in the OFFPAD's authentication mechanism while complying with FIDO standards.

7.2 Future work

Using the findings from the project and addressing the current limitations of threshold signature implementations in FIDO2 authentication, several aspects need to be improved. In this section we will suggest some improvements to allow the development of a fully functional threshold authentication system that is FIDO2-compliant.

ECDSA threshold signatures for FIDO2 compliance

The biggest flaw of the current implementation is the lack of compliance with the FIDO2 protocol. Future work involves the use of the three-round threshold ECDSA protocol outlined in the project (Section 3.4), as that would enable full compliance with FIDO2 while keeping good round efficiency and security. However, such an implementation has important challenges, such as increased computational cost compared to FROST, the need for more complex coordination techniques due to one more round of communication, and the need to adapt to the hardware of the OFFPAD device.

Bluetooth communication

We relied on USB communication for the implementation, but this places an important limitation on real-world usability and scalability in multi-device environments. A change towards Bluetooth Low Energy (BLE) communication would significantly improve the usability of the selected threshold signature scheme. Future work should adapt the implementation to operate efficiently over Bluetooth with the coordinator model for communication.

Bluetooth implementation must address additional security considerations, including proper pairing and bonding mechanisms, encryption of threshold signature shares during transmission, and protection against wireless attack vectors. Furthermore, the implementation should also leverage BLE's power-saving features to maintain minimal power consumption of the involved devices.

Threshold parameter optimization

The current setup consists of a 2-out-of-3 threshold model because of the number of development boards that were available for the project. Even though the setup we used presented good results, future work should investigate and evaluate other configurations for N (number of participants) and T (threshold) to optimize the balance between security, robustness, and efficiency in different use cases.

Enhanced security model

Due to hardware limitations, the security model employed relies on a coordinator model, which introduces potential vulnerabilities and limits the fully decentralized nature of threshold signatures. Future research should aim to create peer-to-peer communication to eliminate the use of a trusted coordinator.

Post-quantum threshold signatures

Finally, the emergence of quantum computing poses an important threat to current cryptographic systems, including threshold signature schemes we implemented in this thesis. The National Institute of Standards and Technology (NIST) [US 25] and other standardization bodies are actively working on post-quantum cryptographic standards, with several post-quantum signature schemes that are already getting standardized, which will be included in the FIDO standard.

We believe an important field for future work includes the study and development of possible threshold versions of these post-quantum schemes. While these post-quantum schemes are more computationally demanding, adapting them to work with threshold signature protocols would ensure long-term security against quantum threats.

References

[ADI+17] B. Applebaum, I. Damgård, *et al.*, "Secure arithmetic computation with constant computational overhead", in *Advances in cryptology – CRYPTO 2017, part I*, J. Katz and H. Shacham, Eds., ser. Lecture notes in computer science, vol. 10401, Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2017, pp. 223–254. [Online]. Available: https://doi.org/10.1007/978-3-319-63688-7_8.

[Ban22] Banca d'Italia, *GitHub - bancaditalia/secp256k1-frost: Implementation of Flexible Round-Optimized Schnorr Threshold Signatures on secp256k1*, Jun. 2022. [Online]. Available: https://github.com/bancaditalia/secp256k1-frost.

[Bit14] Bitcoin Core, *Bitcoin-core/secp256k1*, Aug. 2014. [Online]. Available: https://github.com/bitcoin-core/secp256k1.

[Blu24] Bluetooth, *Core Specification*, en-US, 2024. [Online]. Available: https://www.bluetooth.com/specifications/specs/core-specification-5-3/.

[BS15] D. Boneh and V. Shoup, "A Graduate Course in Applied Cryptography", en, in 2015. [Online]. Available: https://toc.cryptobook.us/.

[CKGW24] D. Connolly, C. Komlo, *et al.*, "The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures", Internet Engineering Task Force, Request for Comments RFC 9591, Jun. 2024. [Online]. Available: https://doi.org/10.17487/RFC9591.

[CKM21] E. Crites, C. Komlo, and M. Maller, *How to Prove Schnorr Assuming Schnorr: Security of Multi- and Threshold Signatures*, 2021. [Online]. Available: https://eprint.iacr.org/2021/1375.

[DKLsa24] J. Doerner, Y. Kondi, *et al.*, "Threshold ECDSA in three rounds", in *2024 IEEE symposium on security and privacy*, San Francisco, CA, USA: IEEE Computer Society Press, May 2024, pp. 3053–3071. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00178.

[Fel87] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing", in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, ser. SFCS '87, USA: IEEE Computer Society, Oct. 1987, pp. 427–438. [Online]. Available: https://doi.org/10.1109/SFCS.1987.4.

[FID22] FIDO Alliance, *FIDO2*, en-US, May 2022. [Online]. Available: https://fidoalliance.org/fido2/.

[Gar25a] J. García, *PoneBiometrics/JavGar_master*, 2025. [Online]. Available: https://github.com/PoneBiometrics/JavGar_master.

[Gar25b] J. García, "Threshold Signatures for FIDO Authentication", en, 2025.

[GG18] R. Gennaro and S. Goldfeder, "Fast multiparty threshold ECDSA with fast trustless setup", in *ACM CCS 2018: 25th conference on computer and communications security*, D. Lie, M. Mannan, *et al.*, Eds., Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 1179–1194. [Online]. Available: https://doi.org/10.1145/3243734.3243859.

[GG20] R. Gennaro and S. Goldfeder, *One Round Threshold ECDSA with Identifiable Abort*, ACM CCS 2020 (as a joint paper), 2020. [Online]. Available: https://eprint.iacr.org/2020/540.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems", in *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, ser. STOC '85, New York, NY, USA: Association for Computing Machinery, 1985, pp. 291–304. [Online]. Available: https://dl.acm.org/doi/10.1145/22145.22178.

[HM11] D. Hankerson and A. Menezes, "Elliptic Curve Discrete Logarithm Problem", en, in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds., Boston, MA: Springer US, 2011, pp. 397–400. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_246.

[Int25] Internet Engineering Task Force, *Introduction to the IETF*, en, 2025. [Online]. Available: https://www.ietf.org/about/introduction/.

[JMV01] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)", en, in *International Journal of Information Security*, ser. Johnson, Don, Alfred Menezes, y Scott Vanstone. «The Elliptic Curve Digital Signature Algorithm (ECDSA)». International Journal of Information Security 1, n.º 1 (august 2001): 36-63. Aug. 2001, pp. 36–63. [Online]. Available: https://doi.org/10.1007/s102070100002.

[KG20] C. Komlo and I. Goldberg, "FROST: Flexible round-optimized Schnorr threshold signatures", in *SAC 2020: 27th annual international workshop on selected areas in cryptography*, O. Dunkelman, M. J. Jacobson Jr., and C. O'Flynn, Eds., ser. Lecture notes in computer science, vol. 12804, Halifax, NS, Canada (Virtual Event): Springer, Cham, Switzerland, Oct. 2020, pp. 34–65. [Online]. Available: https://doi.org/10.1007/978-3-030-81652-0_2.

[KKS10] A. Khalique, S. Kuldip, and S. Sood, "Implementation of Elliptic Curve Digital Signature Algorithm", in *International Journal of Computer Applications*, May 2010. [Online]. Available: https://doi.org/10.5120/631-876.

[Lin25] Linux Foundation, *Linux Foundation - Decentralized innovation, built with trust*, en, 2025. [Online]. Available: https://www.linuxfoundation.org.

[LLM23] Z. Luo, R. Liu, and A. Mehta, "Understanding the RSA algorithm", Aug. 2023. [Online]. Available: https://www.researchgate.net/publication/372962415_Understanding_the_RSA_algorithm.

[LN18] Y. Lindell and A. Nof, “Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody”, in *ACM CCS 2018: 25th conference on computer and communications security*, D. Lie, M. Mannan, *et al.*, Eds., Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 1837–1854. [Online]. Available: https://doi.org/10.1145/3243734.3243788.

[Mag16] K. Magons, “Applications and Benefits of Elliptic Curve Cryptography”, en, 2016. [Online]. Available: https://ceur-ws.org/Vol-1548/032-Magons.pdf.

[Men08] A. Menezes, “The Elliptic Curve Discrete Logarithm Problem: State of the Art”, en, in *Advances in Information and Computer Security*, K. Matsuura and E. Fujisaki, Eds., vol. 5312, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 218–218. [Online]. Available: https://doi.org/10.1007/978-3-540-89598-5_14.

[Mil86] V. S. Miller, “Use of Elliptic Curves in Cryptography”, en, in *Advances in Cryptology — CRYPTO ’85 Proceedings*, H. C. Williams, Ed., vol. 218, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426. [Online]. Available: https://doi.org/10.1007/3-540-39799-X_31.

[Och22] E. Ochekliye, *Elliptic Curves and the Discrete Log Problem*, en, May 2022. [Online]. Available: https://enigbe.medium.com/about-elliptic-curves-and-dlp-ed76c5e27497.

[Pai99] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes”, in *Advances in cryptology — EUROCRYPT ’99*, J. Stern, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. [Online]. Available: https://doi.org/10.1007/3-540-48910-X_16.

[Ped91] T. P. Pedersen, “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”, en, in *Advances in Cryptology - CRYPTO ’91*, J. Feigenbaum, Ed., vol. 576, Berlin, Heidelberg: Springer, 1991, pp. 129–140. [Online]. Available: https://doi.org/10.1007/3-540-46766-1_9.

[Pol78] J. M. Pollard, “Monte Carlo methods for index computation ()”, en, *Mathematics of Computation*, vol. 32, no. 143, pp. 918–924, 1978. [Online]. Available: https://doi.org/10.1090/S0025-5718-1978-0491431-9.

[PON18] PONE Biometrics, *Welcome to Pone Biometrics*, en-US, 2018. [Online]. Available: https://ponebiometrics.com/.

[PON24] PONE Biometrics, *Offpad*, en-US, 2024. [Online]. Available: https://ponebiometrics.com/product/offpad/.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, en, in *Communications of the ACM*, vol. 21, New York: Commun. ACM, Feb. 1978, pp. 120–126. [Online]. Available: https://doi.org/10.1145/359340.359342.

[Sch05] B. Schoenmakers, “Verifiable Secret Sharing”, en, in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg, Ed., Boston, MA: Springer US, 2005, pp. 645–647. [Online]. Available: https://doi.org/10.1007/0-387-23483-7_452.

[Sch91] C. P. Schnorr, “Efficient signature generation by smart cards”, en, *Journal of Cryptology*, pp. 161–174, Jan. 1991. [Online]. Available: https://doi.org/10.1007/BF00196725.

[Sha79] A. Shamir, “How to share a secret”, en, in *Communications of the ACM*, vol. 22, New York: Commun. ACM, 1979, pp. 612–613. [Online]. Available: https://doi.org/10.1145/359168.359176.

[Sil09] J. H. Silverman, *The Arithmetic of Elliptic Curves* (Graduate Texts in Mathematics). New York, NY: Springer, 2009, vol. 106. [Online]. Available: http://link.springer.com/10.1007/978-0-387-09494-6.

[SY06] S. Subramanya and B. Yi, “Digital signatures”, *IEEE Potentials*, vol. 25, no. 2, pp. 5–8, Mar. 2006. [Online]. Available: https://ieeexplore.ieee.org/document/1649003/.

[US 25] U.S. Department of Commerce, *National Institute of Standards and Technology*, en, Jun. 2025. [Online]. Available: https://www.nist.gov/.

[W3C19a] W3C, *Client to Authenticator Protocol (CTAP)*, 2019. [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html.

[W3C19b] W3C, *Guide to Web Authentication*, en, 2019. [Online]. Available: https://webauthn.guide.

[W3C21] W3C, *Web Authentication: An API for accessing Public Key Credentials - Level 2*, 2021. [Online]. Available: https://www.w3.org/TR/webauthn-2/#biblio-fido-ctap.

[Won21] D. Wong, “Real-World Cryptography”, en, pp. 129–149, 2021.

[Zep23] Zephyr, *Zephyr Project*, en-US, 2023. [Online]. Available: https://www.zephyrproject.org/.

Norwegian University of
Science and Technology

BIOMETRICS