Multi-signatures

In order to secure communication and data sharing on multiple devices, digital signatures are essential. However, situations might arise in terms of communication where there is a need to have several signatures since one signature might not be sufficient for verifying its validity. Multisignatures provide a solution to this issue by enabling a group of signers to jointly sign a message. This ensures that the message is only recognized as legitimate if the necessary number of signers have signed it. With multisignature, multiple signers can each add their own signature to a message, creating a single signature that can be verified by anyone with access to the public key. Blockchain technology, digital certificates, encrypted communications, and authentication protocols all make extensive use of multisignatures. Multisignature systems have the major benefit of increasing security and accountability while also offering flexibility and scalability. The drawbacks of conventional digital signatures, which only permit one signer to sign a message, can be solved by multisignature. Similarly to conventional digital signatures, multi-signatures consist of three algorithms: key generation, signing, and verification. The biggest impact on speed and security of multisignature schemes is key generation, as participants have to agree on private and public keys. After that, participants use their private key to generate a signature on the message, and finally, a verifier, usually one entity, checks the validation of the signature.

FROST signature

Context

FROST signature is a method for cryptographically signing communications that enables many users to sign messages using a single secret key. It is based on the Schnorr signature technique and generates the shared secret key using the Pedersen DKG protocol. In January 2020, Chelsea Komlo and Ian Goldberg described FROST in a research article.[2] It has drawn interest from the cryptography community because to its efficiency and security trade-offs, as well as its potential application in decentralized systems like block-chain networks.

FROST has several advantages:

1. **Threshold security**: FROST offers threshold security, which implies that in order to access the private key, an attacker would need to successfully compromise a significant number of signers.
2. **Flexibility**: FROST may be modified to meet various threshold and signer criteria due to its adaptability.
3. **Efficiency**: FROST uses less bandwidth and has a high processing efficiency [2].

4. **Round-optimized**: FROST is created to reduce the number of rounds that are necessary for communication between the signers [2].

DKG in FROST

DKG is used to generate and distribute the shared secret key among all participants for later signing. Pedersen DKG protocol serves as the foundation for the DKG in FROST. Generally any DKG scheme, Pedersen DKG included, consist of following stages:

5. **Initialization**: The parties decide on a generating point on a shared EC. In order to calculate their matching public key on the EC, each side produces a random value .
6. **Share distribution**: The parties divide their random value into shares and provide those shares to the other members of the group using SSS. Each party obtains shares from every other participant and computes its own polynomial interpolated reconstructed private key .
7. **Share verification**: The parties calculate a shared public key on the EC using their private keys that they have rebuilt. They then trade promises to their rebuilt private keys, using these commitments to confirm the authenticity of the shared public key.
8. **Key generation**: If the key verification procedure is successful, the parties utilize their shared public key to create their cryptographic keys. The parties' respective public keys are used to form the shared public key, which is then used to generate the matching reconstructed private keys.

Pedersen DKG is done in 2 rounds. Since FROST in this thesis is based on EC computations, DKG is done as following (Please, consider the use of ECC):

Round 1

- Every participant P_i computes polynomial $f_i(x) = \sum_{j=0}^{t-1} a_{i,j} * x^j \pmod{Q}$ where $a_{i,j}$ is random number [2].
- Every participant P_i computes public commitment and send it to every participant: $\vec{X_i} = (\phi_{i,0}, ..., \phi_{i,(t-1)})$, where $\phi_{i,j} = a_{i,j} * G \mid 0 \leq j \leq t-1$

Round 2

- Every participant P_i securely sends to all participants P_j a secret share $(j, f_i(j))$ [2].
- Every participant P_i verifies secret share from participant P_j as follow:

  $G * f_j(i) \equiv \sum_{k=0}^{t-1} \phi_{j,k} * i^k \pmod{Q}$. If verification does not hold protocol is aborted.
- Generate keys as following:
  - Secret share: $s_i = \sum_{j=1}^{n} f_j(i) \pmod{Q}$ [2]
  - Verify share: $Y_i = G * s_i$
  - Public key: $Y = \sum_{j=1}^{n} \phi_{j,0}$

Signing in FROST

FROST proposal introduce 2 options of signing. Standard signing is done within 2 rounds. Alternatively, option with 1 round signing is presented that is done by preprocess stage and by adding entity commitment server [2]. By that, commitment phase is not counted to signing part as it is done before signing considered as pre-requisite to participate in signing operation. Overall, both versions are based on the same computations.

The signing part can be divided into 3 phases (Please, consider the use of ECC):

1. **Commitment phase**
   - Selected number of participants t out of n participate in signature, where they calculate each single-use public commitments share $D_i = G * d_i$; d_i is random number. Then the commitments are sent to aggregator.
   - Aggregator checks if all selected participants t have sent commitment shares. If not protocol is aborted, Otherwise, public commitment is created as: $R = \sum_{i=1}^{t} D_i \pmod{Q}$
   - Aggregator sends tuple (m, R, S) to all participants t, where S is set of participants t [2]

2. **Challenge phase**
   - every participant P_i computes challenge $c = H(R||m)$ [2]
   - every participant P_i computes signing share $z_i = d_i + \lambda_i * s_i * c \pmod{Q}$, where λ_i is coefficient of Lagrange polynomial interpolation [2].
   - every participant P_i send signing share to aggregator and deletes d_i, D_i
3. **Signature phase**
   - Aggregator verifies each response by checking: $G * z_i \equiv D_i + Y_i * c * \lambda_i \pmod{Q}$. If verification does not hold, protocol is aborted.
   - Aggregator computes group's response $z = \sum_{i=1}^{t} z_i \pmod{Q}$
   - Release signature $\sigma = (z, c)$ along with message m [2]

The following algorithms show how signing and verifying is done in this work:

Algorithm 1 Signing Algorithm for EC-Schnorr Signature

9. Select random nonce $k \in_R Z_q$
10. Calculate the point on curve $R = k * G$
11. Calculate hash $c = H(R||m)$
12. Calculate the challenge with secret key S_k $s = k + c * S_k$
13. Signature is defined as $\sigma = (s, c)$

Algorithm 2 Verification Algorithm for EC-Schnorr Signature

1. $Parse\ \sigma\ into\ (s, c)$
2. Calculate $R' = s * G - c * P_k \mid P_k = G * S_k$
3. Calculate hash $z' = H(R'||m)$

if $z' = c$ **then**
    output is 1; Valid
else
    output is 0; Rejected
end if

The main difference between Schnorr Signature and EC-Schnorr Signature is fact that exponentiation is replaced by simpler and faster multiplication in EC. Lastly, generator G, public commitment R and public key P_k are points on EC.

Implementation

The next section focuses on the implementation of multi-signature in the programming language C, with carefully chosen and included essential libraries. Moreover, C code interacts with Kotlin by Java Native Interface (JNI). The project can be found on GitHub.

Multi-signature Comparison

During the research on the defined problem, five potential candidates were selected for further implementation of multi-signature for. IoT devices can be considered secondary devices that extend the functionality and connectivity of standard devices such as laptops or smartphones. In most cases, they have limited computing capacity, which plays a significant role in choosing an appropriate multi-signature scheme. Therefore, the main attention was given to the number of rounds required by each scheme and the size of exponentiation computed during the process. The optimal solution can be found by striking a balance between the speed of the scheme and sufficient security. The following table 2.1 describes the number of iterations and the computational complexity of each part of the scheme for each scheme.

Signature scheme	FROST 1 round	FROST 2 round	MuSig2	BN06	mBCJ
Complexity	OMDL + PROM	OMDL + PROM	OMDL	DL + ROM	DL + ROM
KeyGen (# iter.)	2	2	n	n	n
KeyGen (# exp.)	3n + nt + t + 1	3n + nt + t + 1	1	1	2
Sign (# iter.)	1	2	2	3	2
Sign (# exp.)	2	t + 2	n + 3	1	4
Verify (# exp.)	2	2	n + 2	n+1	8
Type	Threshold	Threshold	Mu-Sig	Mu-Sig	Mu-Sig
Party Involved	(n,t)	(n,t)	(n,n)	(n,n)	(n,n)
Life-time	N/A	N/A	N/A	N/A	N/A

Tab. 1: Comparison of Multi-signature

Implementation of BoringSSL Library

This section clearly reveals functions and cryptography's methods implemented for further signature including hash function, EC curve and generator of random numbers based on the library. The implementation is done in *globals.c* and is called when is needed.

BoringSSL was chosen as one of the best libraires for Andorid as it is a fork library of OpenSSL well-liked software library that offers developers useful cryptographic functionalities. Therefore, the implementation of FROST signature is based on this library.

Implementation of SHA-256

SHA-256 was chosen for use in the implementation for ensuring integrity and prevent tampering as it is a popularly used cryptographic hash algorithm that accepts arbitrary-length input messages and generates a fixed-size output (256 bits) that is specific to the input.

Hash function is used for concatenation of message and public commitment for creating signing share in signing part. Moreover, hash function is also needed for verification of final signature.

Implementation of Secp256r1

The EC employed, the hash function, and the structure of the code with adequate parameters all play major roles in the security of the final signature implementation. EC secp256r1, also known as prime256v1, is a widely used ECC curve. It is defined over a prime field, and its parameters are standardized by NIST. The curve equation is defined as [1]:

$$y^2 = x^3 - ax + b$$

where parameter a, b are defined as:

$a = FFFFFFFF\ 00000001\ 00000000\ 00000000\ 00000000\ FFFFFFFF$
$FFFFFFFF\ FFFFFFFC$

$b = 5AC635D8\ AA3A93E7\ B3EBBD55\ 769886BC\ 651D06B0\ CC53B0F6$
$3BCE3C3E\ 27D2604B$

On the other hand, obtaining appropriate parameters is essential. The emphasis is on a suitably big modulo P, as the parameters (a, b) are constant. This application makes advantage of the modulo "P" of the size of $2^{224}(2^{32} - 1) + 2^{192} + 2^{96} - 1$ that co-responds with SECG recommendation [1].

The implementation uses uncompressed generator G in form [1]:

G = 04 $6B17D1F2$ $E12C4247$ $F8BCE6E5$ $63A440F2$ $77037D81$ $2DEB33A0$ $F4A13945$ $D898C296$ $4F$ $E342E2$ $FE1A7F9B$ $8EE7EB4A$ $7C0F9E16$ $2BCE3357$ $6B315ECE$ $CBB64068$ $37BF51F5$

Finally, the order Q is defined as following [1]:

$Q = FFFFFFFF$ 00000000 $FFFFFFFF$ $FFFFFFFF$ $BCE6FAAD$ $A7179E84$ $F3B9CAC2$ $FC632551$

Randomization

The implementation uses generation of multiple random numbers that have to be securely generated with unpredictability. Generation of random numbers relies on function $RAND\_bytes()$ that is implemented in library BoringSSL.

The specified RAND method, a collection of instructions for producing random numbers, determines the algorithm used by RAND_bytes() . The Deterministic Random Bit Generator (DRBG) technique is used as the default RAND method in OpenSSL 3.0 from NIST SP 800-90A . DRBG belongs to a group of Pseudo-Random Number Generators (PRNGs) that are cryptographically safe. Function $generate\_rand()$ is called every time, 32 byte random number is needed.

FROST Implementation

FROST Setup

Setup of FROST is basically implementation of Pedersen-DKG that is done withing 2 rounds.
Secret share, verify share and public key are generated for all participants that are stored for later signing. Setup diagram is showed on following figure 2.2:

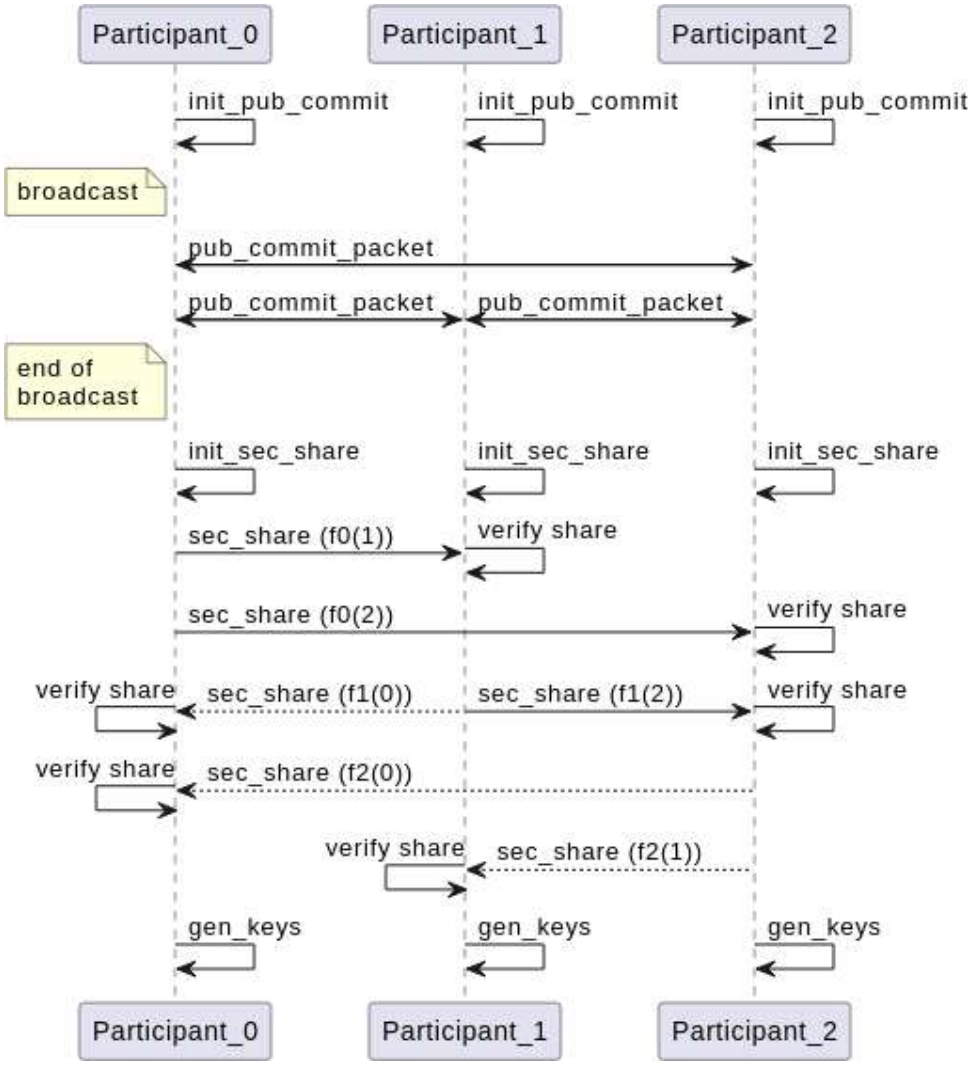


Fig. 1: FROST Setup Diagram

FROST Signing

After setup of FROST is successfully finished, signing part is started by pre-selected number of t participants P_i located in array $threshold\_set$[].

At the end, signature packet is created by called function *signature*(). *Gen_signature*() is called within library for sum of all partial signatures. Then packet is published as signature and hash. Exact sequence of signing is showed on diagram with figure 2.3.

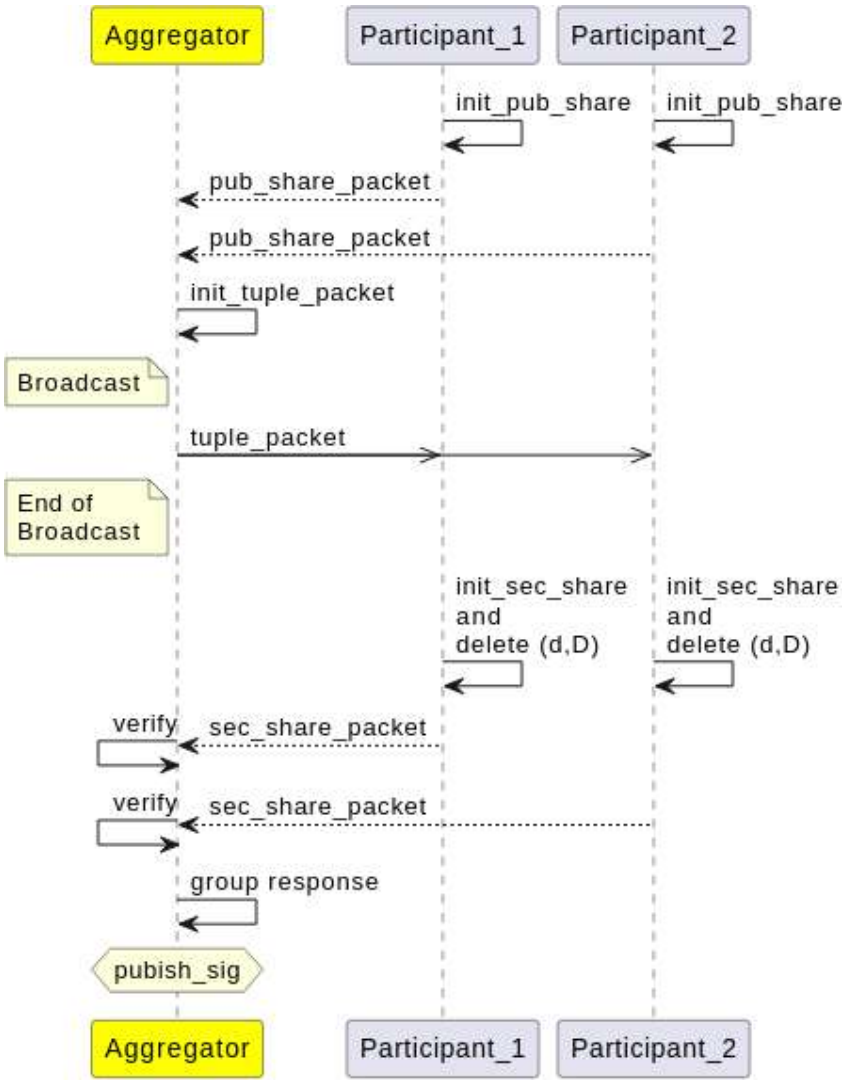


Fig. 2: FROST Signing Diagram

FROST Verification

The EC-Schnorr signature verification algorithm is a process used to verify the validity of an EC-Schnorr signature on a message that is more clearly described by Algorithm 2. Idea behind the verification is in comparison of the calculated

hash value z' with the value of c in the signature. If the values match, then the signature is considered valid. Otherwise, the signature is rejected. Verification of signature works by reconstructing the point R' from the signature components and the public key, and then verifying that its hash value matches the value of c in the signature. If the hash values match, it provides strong evidence that the signature was produced by the holder of the private key corresponding to the public key used in the verification process.

Cross-compilation from x86_64 to ARM64-v8a

Cross-compilation is the process of compiling code on one platform to run on a different target platform. In my Android application project, I performed cross-compilation to build a native C library using BoringSSL for cryptographic signing and verification, targeting Android devices with ARM architecture. Since my development environment was different from the target (ARM-based Android), I used CMake along with the Android NDK (Native Development Kit) to configure and cross-compile the native code. BoringSSL itself also needed to be built for ARM to ensure compatibility and optimal performance. The entire project, including the native components and the Kotlin-based GUI, was managed and built using Gradle within Android Studio, which streamlined the build process and dependency management.

A native library is compiled code, typically written in C or C++, designed for high-performance tasks and low-level system access. In this project, I implemented cryptographic signing and verification in a native library using BoringSSL, enabling secure and efficient processing of sensitive data. The native library was compiled into a shared object (.so) file that could be loaded into the Android application.

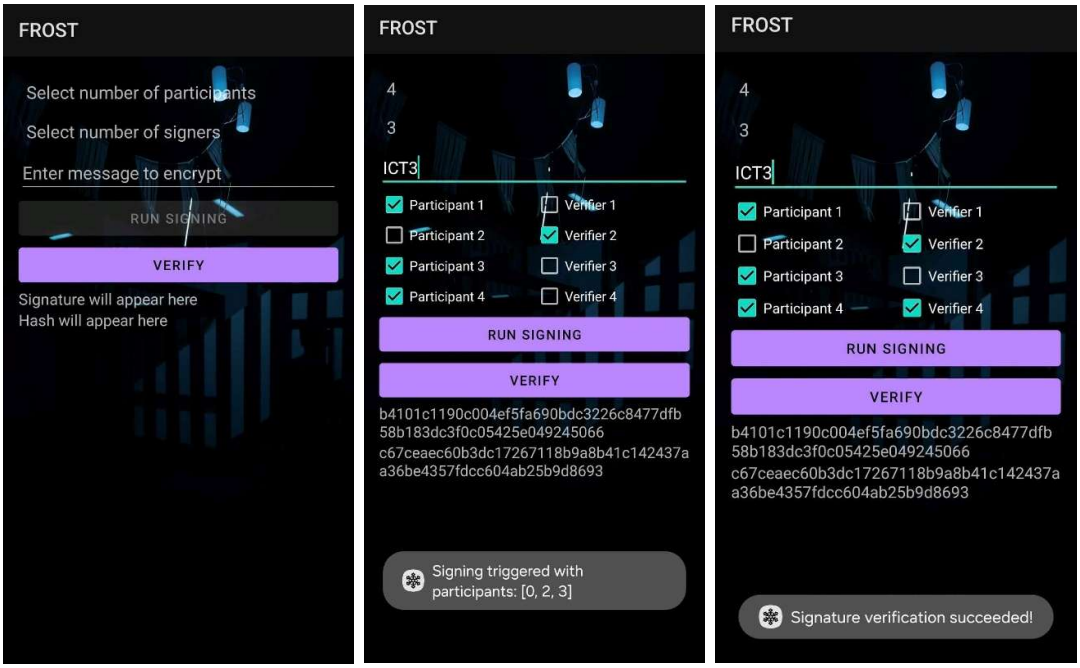
To connect the native library with the Android application's GUI, I utilized the JNI, which allowed Kotlin code to call native C functions. Through JNI, I implemented functions like *Java_cz_but_myapplication_MainActivity_executeSigning* and *Java_cz_but_myapplication_MainActivity_verifySignature*. These functions acted as a bridge, allowing the Kotlin-based GUI to interact with the cryptographic logic in the native library. For example, *executeSigning* performed signing operations and returned the results, which were displayed in the app's user interface. Similarly, *verifySignature* validated the generated signature and provided the verification results.

By working in Android Studio with Gradle, I was able to integrate Kotlin for modern, maintainable application development with C for high-performance cryptographic operations, ensuring a robust and efficient solution for Android

GUI

The graphical user interface (GUI) of the application is designed to facilitate cryptographic signing and verification processes in a user-friendly and interactive manner. It enables users to dynamically configure parameters such as the number of participants, the signing threshold, and the message to be signed or verified. Users can select participants through checkboxes, ensuring the signing process meets the threshold requirements. The interface validates inputs in real-time, providing immediate feedback if any condition is not met, such as an insufficient number of selected signers or a missing message.

The GUI also includes buttons to trigger the signing and verification processes, which are only enabled when all prerequisites are satisfied. Upon triggering these operations, the application seamlessly interacts with the underlying native library, and the results, such as the generated signature and hash, are displayed on the screen. Error handling is integrated throughout the interface, with clear prompts and notifications to guide the user. Overall, the application's GUI ensures an intuitive experience while leveraging the cryptographic capabilities of the underlying system.



[1] National Institute of Standards and Technology. Digital signature standard (dss), 2019-10-31 2019. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5-draft.pdf.

[2] Chelsea Komlo. Rsa vs. ecc comparison for embedded systems. 2020. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/00003442A.pdf.