BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

LIGHTWEIGHT MULTI-SIGNATURE SCHEMES FOR IOT

ODLEHČENÉ VÍCENÁSOBNÉ PODPISY PRO IOT

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	Jakub Jarina
SUPERVISOR VEDOUCÍ PRÁCE	M.Sc. Sara Ricci, Ph.D.

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Information Security**

Department of Telecommunications

Student:	Jakub Jarina	***ID:***	230086
Year of study:	3	***Academic year:***	2022/23

TITLE OF THESIS:

Lightweight Multi-signature schemes for IoT

INSTRUCTION:

The work is focused on the implementation and comparison of multi-signature schemes for Internet of Thing (IoT) environment. The student will analyze current multisignature schemes and compare them from the point of view of security, computing and memory requirements. The thesis aims to implement, analyze and compare the threshold signature proposed in [1] and the best state-of-the-art proposal. Since the schemes allow multi-devices signing, the implemented protocols are expected to be run on different devices with limited computing power, e.g., microcontrollers.

RECOMMENDED LITERATURE:

[1] Ricci, S.; Dzurenda, P.; Casanova-Marques, R.; Čika, P.: Threshold Signature for Privacy-preserving Blockchain. In Business Process Management: Blockchain, Robotic Process Automation, and Central and Eastern Europe Forum. Münster, Germany: Springer, 2022. p. 1-15. ISBN: 978-3-031-16167-4.

[2] Komlo C, Goldberg I. FROST: flexible round-optimized Schnorr threshold signatures. InInternational Conference on Selected Areas in Cryptography 2020 Oct 21 (pp. 34-65). Springer, Cham

Date of project specification:	6.2.2023	***Deadline for submission:***	26.5.2023

Supervisor: M.Sc. Sara Ricci, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

ABSTRACT

The focus of this work is to introduce the topic of multi-signatures and subsequently implement a scheme supported for Internet of Things (IoT) devices. The work analyzes known multi-signature schemes from the perspective of security, computational complexity, and memory requirements. The work includes the implementations of a Flexible Round-Optimized Schnorr Threshold signature and Threshold Signature for Privacy-preserving Blockchain in the C programming language.

KEYWORDS

Multi-signature, Threshold signature, Internet of Things, Schnorr signature, Shamir's Secret Sharing, Distributed Key Generation, Elliptic Curves Cryptography, Security, Proof of Knowledge, Sigma protocols, Flexible Round-Optimized Schnorr Threshold signatures

ABSTRAKT

Zameraním tejto práce je predstaviť problematiku hromadných podpisov a následne implementovať schému podporovanú pre zariadenia internetu vecí (IoT). Práca anlyzuje známe viacnásobné podpisy z pohľadu bezpečnosti, výpočetnej a pamäťovej náročnosti. Práca obsahuje implementáciu Flexibilne, optimalizovaného Schnorrovo prahového podpisu a prahového podpisu pre zachovanie súkromia v blockchaine v programovacom jazyku C.

KĽÚČOVÉ SLOVÁ

Viacnásobné podpisy, Prahové podpisy, Internet vecí, Schnorrov podpis, Shamirovo zdieľanie tajomnstva, Distribuované generovanie Kľúčov, Kryptografia eliptických kriviek, bezpečnosť, Dôkaz znalostí, Sigma protokoly, Flexibilné, optimalizované Schnorrove prahové podpisy

Typeset by the thesis package, version 4.07; http://latex.feec.vutbr.cz

ROZŠÍŘENÝ ABSTRAKT

Zameraním tejto práce je predstaviť problematiku hromadných podpisov a následne implementovať schému podporovanú pre zariadenia internetu vecí (IoT) zo dôvodu čoraz väčšej popularite IoT a dôležitosti bezpečnosti pri výmene a ukladaní dát, ktoré sú často dôverné alebo citlivé. Digitálne podpisy sa ukázali ako sľubné riešenie na zabezpečenie výmeny dát, ale tradičné schémy jedného podpisu nie sú pre IoT častokrát vhodné z dôvodu limitácii vo výkonostnej kapacite. Preto cieľom tejto práce je taktiež analyzovať moderné schémy z pohľadu bezpečnosti, výpočetnej a pamäťovej náročnosti. Táto práca pozostáva z 3 hlavných častí: teoretická čast, implementácia a nakoniec pracovné prostredie.

Teoretická časť je napísaná chronologisckou formou, čo znamená od základných primitív a po tie komplexné pre plné pochopenie hromadných podpisov. Ako prvým sa teoretická časť zaoberá digitálnym podpisom a jeho hlavnými vlastnosťami v kapitole 1.1, úvodom do samotnej všeobecnej schémy až po najznámejšie používané algoritmy. Taktiež táto kapitola predstavuje hash funkcie ako neodmysliteľnú súčasť digitálnych podpisov, s bliším zameraním na vlastnosti bezpenčných hash funkcii a následním porovnaním najznámejších použivaných hash funkcii z pohľadu bezpečnosti.

V kapitole 1.2 sa popisuje oblasť kryptografie založená na elitpických krivkách, z dôvodu neskoršej implementácie v tejto práci. Hlavným dôvodom využitia je ich efektívnosť, rýchlosť a väčšia bezpečnosť v porovnaní s klasickými kryptografickými primitívami. Kapitola popisuje funkciu eliptickej krivky, jej štruktúru a základné vlastnosti. Blišie je vysvetlené funkčnosť a využitie eliptických kriviek v praxi najmä vďaka násobeniu skalárnej hodnoty a bodu na eliptickej krivke. V neposlednom rade kapitola sa zameriava na možné útoky ale aj bezpečnosť a ich porovnanie k symetrickým algoritmom alebo RSA algoritmu. Nakoniec sú porovnané 2 najpoužívanejšie eliptické krivky a to secp256k1 a secp256r1. Porovnanie bolo zamerané najmä vplyv rýchlosti podpisovania a overnia na základe využitej knižnice. Z výsledkov bol vyvedený záver v prospech eliptickej krivky secp256r1, ktorá je neskôr použitá pre implementáciu.

V nasledúcej kapitole 1.3 je popísaná digitálny podpis s názvom Schnorrov podpis, ktorý som sebou nesie veľa výhod, ktoré esenciálne v hromadných podpisoch. Jednými z nich sú práve jednnoduchosť a linearita, ktorá umožnuje spočítať viacero podpisov dokopy, bez toho aby to bolo možné rozoznať. V kapitole je blizšie opísaný algoritmus pre podpis a overenie. Kedže práca je založená na eliptických krivkách, kapitola porovnává rozdiely v schémach založelných na eliptických krivkách a taktiež popisuje algoritmy podpisu a overenia, ktoré sú použité neskôr v implementácii.

Digitálne podpisy sú kľúčovou súčasťou zabezpečenia IoT zariadení, pretože umožňujú autentifikáciu a overovanie správ a dát zdieľaných medzi zariadeniami.

Avšak môžu nastať situácie, kde jediný podpis nie je dostatočný a je potrebných viacero podpisov na zabezpečenie platnosti správy. Hromadné podpisy (multipodpisy) poskytujú riešenie tohto problému, umožňujúc skupine podpisových osôb spoločne podpísať správu a zabezpečiť, že bude uznávaná len v prípade, že ju podpíše dostatočný počet osôb. V tejto práci špecificky kapitola 1.4 sa zaoberá základmi multipodpisov, ich klasifikáciou a výhodami a nevýhodami. Multipodpisy pozostávajú z troch algoritmov: generovania kľúčov, podpisovania a overovania. Proces generovania kľúčov má najväčší vplyv na rýchlosť a bezpečnosť multipodpisových schém, pretože účastníci musia súhlasiť s privátnymi/verejnými kľúčmi. Po vygenerovaní kľúčov účastníci používajú svoje súkromné kľúče na generovanie podpisov na správe a overovač kontroluje platnosť podpisu.

Ako bolo spomenuté vyššie, generovanie kľúčov je najzložitejšou časťou multipodpisovných schém. Fakt, že viaceré nedôveryhodné strany musia spoločne prijať a distribuovať verejné a súkromné kľúče bez odhalenia tajných informácií, robí generovanie kľúčov problematickým. Okrem toho musí byť verejný kľúč konečnou funkciou súkromného kľúča. Inými slovami, generovanie kľúčov musí splniť požiadavky na súkromie a korektnosť. Existujú dve hlavné techniky, Shamirovo Bezpečné Zdieľanie a Distribuvované Generacia Kľúčov, ktoré sú bizšie popísané v podkapitolách 1.5.4 a 1.5.5. Kedže Shamirovo Bezpečné Zdieľanie má nevýhodu v potrebu distribútora čiastočných súkromných kľúčov, ktorý vie rekonštruovať hlavný súkromný kľúč, čo je považované za bod zraniteľnmosti algoritmu, preto neskoršia implementácia je založená na Distribuvovanej Generacii Kľúčov. Táto schéma generovania kľúčov pozostáva z inicializácie, distribúcie dielov "shares", ich overnie a nakoniec generácia kľúčov. Pre rekonštrukciu súkromného kľúča je použitá Lagrangova polynomiálna interpolácia, ktorá je popísaná v kapitole 1.6.

Kapitola 1.7 sa zaoberá Paillierovým kryptosystémom, ktorý je známy ako pravdepodobnostná asymetrická metóda používaná v kryptografii s verejným kľúčom, založená na probléme distrkétneho logaritmu. Táto kryptografická metóda má vlastnosť aditívnej homomorfie, čo umožňuje kombináciu dvoch šifrovaných textov bez poškodenia výsledku. Dekryptácia nie je potrebná, pretože výpočet funguje tak, akoby príslušné otvorené texty boli jednoducho sčítané. Avšak účinnosť Paillierovho kryptosystému ako homomorfnej šifry je stálym problémom. Na riešenie tohto problému bolo predložených niekoľko optimalizačných nápadov.

Predosledná teoretická kapitola 1.8 sa zaoberá samotným flexibýlnym rundovo optimalizovaným prahovým podpisom založený na Schnorrovej schéme známy ako FROST podpis. Prahové podpisy sú špecifická obnož multipodpisov, pri ktorých je potrebná minimalná účasť podpisujúcich správu z celkovej množiny možných podpisujúcích. v kapitole sú blizšie zmienené výhody tohoto podpisu. V podkapitolách je blizšie zmeniená distribucá kľúčov, ktorá je založená na Pedersonovej schéme Distribuovanej Generácie kľúčov. Tá pozostáva z 2 rúnd, ktoré sú matematickz popísané v podkapitole 1.8.2. Nakoniec je popisaná schéma samotného podpisu v podkapitole 1.8.3, ktorá pozostáva z 3 fázy: vytvorenie záväzku, výzvy a nakoniec samotného podpisu.

Posledná kapitola 1.9 sa zaoberá prahovým podpisom pre zachovanie súkromia v blockchaine. Daný podpis je založený na Paillerovej schéme, Shnorrovom podpise a Shamirovom Bezpečnom Zdieľaní. Daný podpis má dvojité využitie buď pre jedného používateľa, čo zvyšuje bezpečnosť tým, že vyžaduje podpísanie transakcií v blockchaine z viacerých zariadení používateľov, alebo pre celú skupinu používateľov, ktorí spolupracujú, čo podporuje súkromie tým, že umožňuje anonymné podpisy v mene spoločnej peňaženky v blockchaine.

Druhá časť práce sa zaoberá samotnou implementáciou a dôvodom výberu práve FROST podpisu. V Kapitole 2.1 je porovnanie najznamejších multipodpisov. Ich porovnanie je založené z pohľadu bezpečnosti, potrebných interácii pri generovaní kľúčov a podpisovaní a nakoniec samotnej náročnosti algroritmu na výpočet. Z výsledkov bol nakoniec usúdený záver v prospech FROST podpisu, ktorý je ideány kandidát pre implementáciu na IoT zariadenia z pohľadu dostatočnej bezpečnosti, rýchlosti a nenáročnosti na výpočtovú techniku.

Pre samotnú implementáciu je potrebné splňovať určité bezpečnostné kritéria, aby sme ju mohli považovať za bezpečnú. Bezpečnosť FROST podpisu ako schémy, využitie bezpečnej knižnice a následným bezpečných implementovaním funkcii. Taktiež je potrebné použiť dostatočne bezpečne kryptografické primitíva a na záver je potrebné bezpečne alokovať a následné dealokovať pamäť v samotnej implementácii. Túto širokú časť popisuje práve druhá čast tejto práce.

Implementácia je napísaná v programovacom jazyku C s využitím knižnice OpenSLL. Kapitola 2.2 blizšie špecifikuje dôvod výberu s obhajobou bezpečnosti knižnice pre použitie. Pre implementáciu je využitá verzia 3.0, z ktorej sú následne využité potrebné kryptografické primitíva ako hash funkcia SHA-256, eliptická krivka SECP-256r1 a generátor náhodných čísel. Implementácia využíva najnovších funkcii podporované knižnicou OpenSSL 3.0.

Samotná implementácia FROST podpisu je naprogramovaná formou knižnice, ktorá pozostáva z .c súborou *setup.c*, *signing.c*, *globals.c* and *macros.c*, ktoré sú nalikované na hlavičkové súbory z priečinku *../headers* a na hlavičkové súbory knižnice OpenSSL, ktorá je potrebná v OS pre spustenie projektu. *Main.c* slúži ako API na testovanie samotnej knižnice. Hlavičkové súbory v *../header* sú nasledovné: *setup.h*, *signing.h* a *globals.h*. Celý projekt je spustený pomocou *Makefile*. Knižnica má na staroti chod celého podpisu v zmysle , inicializáciu elitptickej krivky a jej paremetrou, matematických výpočtou až po alokáciu a dealokáciu pamäte. Z pohľadu API má uživateľ na staroti volanie funkcii, ktoré slúžia na inicialižaciu potrebných dát a následnú komunikáciu medzi užívateľmi. Pre bližšie pochopenie potrebnej komunikácie a samotnej schémy podkapitola 2.3.1 obsahuje diagram pre generáciu kľúčov a podkapitola 2.3.2 vysvetľuje komunikáciu pri podpisovaní v danom diagrame. V neposlednom rade podkapitola 2.3.5 zhrňuje celkovú bezpečnosť implementácie a bližšie opisuje správu pamäte a jej čistenie. Tá bola testovaná open-source nástrojom Valgrind. V podkapitole 2.3.6 je opísané testovanie FROST implementácie z pohľadu rýchlosti a vplyvu množstva učinkujúcich pri generovaní kľúčov a samotného podpisu. Boli testované schémy $(2, 3)$, $(3, 5)$ a $(4, 6)$ pomocou knižnice $time.h$.

Na záver kapitola 2.4 opisuje samotnú implementáciu prahového podpisu pre zachovanie súkromia v blockchaine. Implementácia je napísaná v programovacom jazyku C s využitím knižnice OpenSLL a cJSON. Implementácia z časti naväzuje na dimplomovú prácu [1], kde bol poskytnutý kód za cieľom využitia počiatočného nastavenia Zmluvy o autentizačnom kľúči na základe Shamirovho zdieľania tajomstva. Implementácia funguje pre schému $(3, 3)$ no pokračujúca práca je potrebná najmä v oblasti generovania kľúčov.

JARINA, Jakub. *Lightweight Multi-signature schemes for IoT*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2023, 62 p. Bachelor's Thesis. Advised by M.Sc Sarra Ricci, Ph.D.

Author's Declaration

Author:	Jakub Jarina
Author's ID:	230086
Paper type:	Bachelor's Thesis
Academic year:	2022/23
Topic:	Lightweight Multi-signature schemes for IoT

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, bachelor's thesis I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to express my deepest gratitude and appreciation to all the individuals who have supported me throughout the process of completing my bachelor's thesis. I would like to thank my supervisor M.Sc Sarra Ricci Ph.D. for professional guidance, patience and suggestions. Many thanks to my colleague Dmitrii Bulashevich for sharing his valuable knowledge and leading me in programming.

Contents

List of Figures

List of Tables

Listings

Introduction

Over the last few decades, the Internet has reached its full potential and has had a significant impact on our everyday lives. Only recently has the Internet unlocked a new feature of digital ownership that transforms our current economy. Nowadays, a trend of owning physical assets is tending to be replaced by the online world, where people can hold different types of ownership.[7] A good example of this can be seen in digital rights or art, commonly associated with Non-Fungible Tokens (NFTs), and any type of cryptocurrency. They are all based on the principles of cryptography, more specifically digital signatures.

Digital signatures employ asymmetric cryptography, which operates with private and public keys and hash functions.[6] The main goal is to securely conduct data with a proof of integrity, authenticity, and non-repudiation over the Internet. Signing schemes, where a single user issues signatures, may suffer from potential threats as only one signer is considered a point of failure. A solution can be found in a specific type of digital signature known as multi-signatures, where two or more people can sign documents together as a group.

This thesis is concerned with multi-signature schemes with a major focus on multi-sig and threshold signatures. They caught public attention after blockchain technology was invented and implemented in the cryptocurrency such as Bitcoin.[8] Blockchain is used by Bitcoin as a ledger to keep track of all network transactions that are made primarily on-chain by multi-sig. However, it brings some drawbacks that threshold signatures are able to solve.

The main goal of the thesis, therefore, is to analyze different types of current multi-sig and threshold schemes and compare them in the area of security and computational complexity. Moreover, the thesis covers the implementation of the currently best-known multi-signature called Flexible Round-Optimized Schnorr Threshold signatures (FROST) [2] and threshold signature [9] in the C programming language. These implementations are focused on the suitable usage of multi-signatures in the area of the Internet of Things (IoTs).

1 Background

This chapter focuses on the theoretical aspects necessary for understanding the topic and its subsequent implementation. While some general knowledge of cryptography is welcomed, it is not required, as all the necessary information is presented in chronological order to provide a deeper understanding.

1.1 Elliptic Curve Cryptography

1.1.1 Context

The classical era and the modern era can be used to divide the history of cryptography. The Diffie-Hellman key exchange algorithm and the introduction of the Rivest–Shamir–Adleman (RSA) algorithm in 1977 mark the turning point between the two. The principle of modern cryptography is that the key is used to encrypt data can be made public, while the key you need to decrypt data can be kept secret. Therefore, these systems are called public-key cryptography, which is also known as asymmetric cryptography. RSA is the first system of this type and is still widely used publicly. It is equipped with strict security proofs based on effective trap-door functions that make the algorithm powerful. In general, trap-door functions are algorithms that are easy in one direction and difficult in the other. The easy technique in the case of RSA multiplies two prime numbers [10]. If multiplication is the easy algorithm, then decomposing the multiplication product into its two prime components is the difficult pair algorithm without a private key [10]. This principle is based on a mathematical statement about the difficulty of factorizing large prime numbers. After the development of RSA and Diffie-Hellman, researchers investigated other similar mathematically based cryptographic methods, which led to the study of Elliptic Curves (ECs) and the development of a new branch of asymmetric cryptography known as Elliptic Curve Cryptography (ECC). This is a public key encryption method based on EC theory that can be an alternative to RSA for instance. This method allows cryptographic keys to be generated faster, more efficiently, and in smaller sizes with equivalent level of security with respect to traditional cryptography[11]. Due to its advantages, it is most often used for digital signatures and in cryptocurrencies such as Bitcoin.

1.1.2 Principle

As opposed to the RSA conventional method of generation as the product of big prime numbers, ECC creates keys through the characteristics of an EC equation.

The points on the graph, used in later generation of private/public keys, can be expressed using the following equation from a cryptographic perspective:

$$y^2 = x^3 + ax + b \tag{1.1}$$

Equation 1.1 is simplified version of EC also known as Short Weierstras Curve [11]. ECC uses this form of curve with two possibilities: EC over the finite field F_p *or* F_{2m}, where p is a prime number and p > 3, respectively. For F_2m 'p' is size of 2_m_ that indicates the EC's points can only have integer coordinates within the field, which is a square matrix of size $p * p$ [12]. Every algebraic operation performed on the field, such as point addition and scalar multiplication, yields a new point. All points belonging to EC can be expressed by cyclic algebraic group or non-overlapping cyclic subgroups (each including a portion of the EC points on the curve). All EC's points are described by equation $n = h * r$ where; n is order of the curve, h is number of subgroups (known as co-factor) and lastly r is the number of points in each subgroup (called order of the subgroups) [11]. By detail examination of the EC displayed on the figure 1.1, it is possible to notice some of remarkable aspects. First of all, curves are horizontally symmetric. Secondly, any non-vertical line will only cross the curve three times, which is a more intriguing characteristic. As a feature can be noticed from the figure 1.1, addition of two point A, B creates a new point that reflects over x-axes resulting in C point. The feature is known as EC point addition that idea can be enhanced. By adding A point k times, also known as EC multiplication,new point is created in really quick way [11]. It is good theoretical example, but in practice point A is replaced with generator point G that bring useful properties. Since EC over finite field form cyclic algebraic group or non-overlapping cyclic subgroups, generator G is used for generating any other point from group/subgroup by multiplying with integer in range of [0...r], where r is order of the cyclic subgroup. This leads to creating public key describes by equation:

$$P = k * G \tag{1.2}$$

where P is public key, k is secret key and G is generator.

1.1.3 Security

EC's security is based on mathematical principle called The Elliptic Curve Discrete Logarithm Problem (ECDLP). Definition is derived from generalized discrete logarithm problem as follows: [13];

Definition 1 *Given a finite cyclic group of points G of order n of an EC over a finite field, a generator A of G, and an element $B \in G$. Find the integer $x, 0 \leq x \leq n - 1 : [x]A = B$*

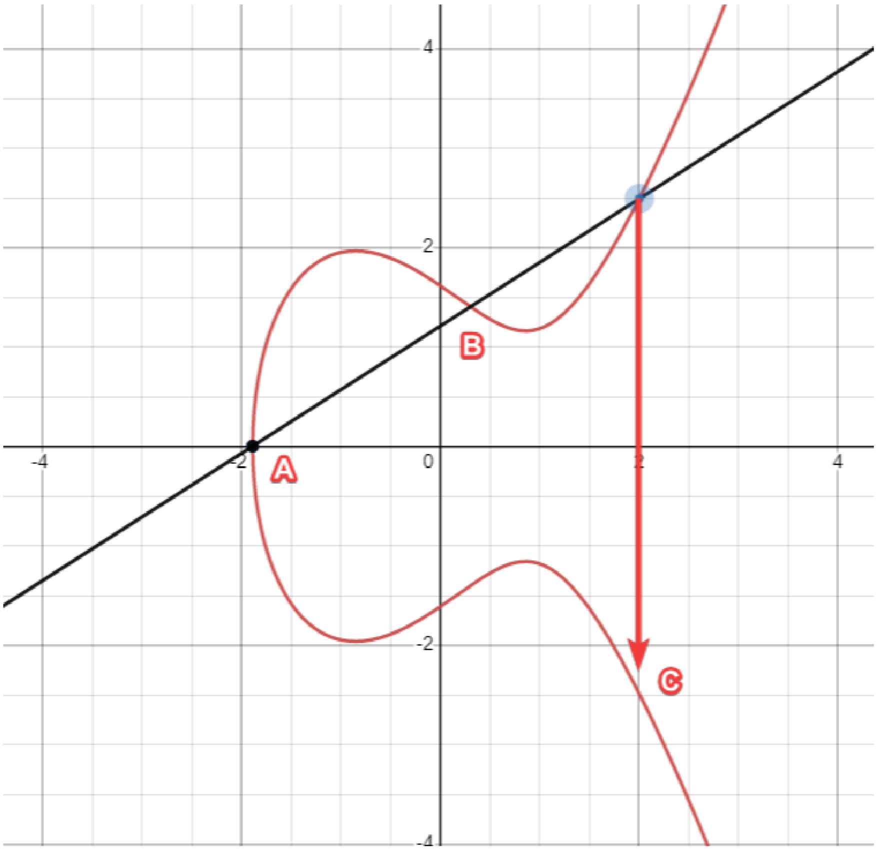


Fig. 1.1: Visualisation of EC

The ECDLP problem lacks an effective solution for carefully selected finite fields and ECs, according to cryptographers, although there is no exact mathematical proof that EC is secure [12]. System security is of the utmost importance. A minimum of 128 bits of security should be provided by contemporary systems, according to the majority of cryptographic specialists. This is not the key length, though. The unique algorithm and its key length work together to provide security. This means that at least a 2*k-bit curve is required in order to obtain a k-bit security strength because the quickest known technique to solve the ECDLP for a key of size 'k' requires steps [12]. Because of this, 256-bit ECs typically offer security strength of about 128 bits. One common example is the belief that AES-128, EC-256 and RSA-3072 bits can provide 128 bits of protection. As the amount of computation accessible to attackers continues to grow, keys typically get longer over time. Table 1.1 clearly shows an efficiency of EC security regards of key length.

		Minimum Size of Public Keys (Bits)	
Security Bits	Symmetric Algorithm	RSA	ECC
80	Skipjack	1024	160
112	3DES	2048	224
128	AES-128	3072	256
192	AES-192	7680	384
256	AES-256	15360	512

Tab. 1.1: Algorithm Size Comparisons for Security [3]

1.1.4 Comparison of Secp256r1 and Secp256k1

As signature is computed by IoT devices in this work, a decision of choosing appropriate computing method of signing and verifying is crucial. Limited computing capacity has to be taken into consideration. Therefore, balance between sufficient security and fast computing is required. Because of its effectiveness and solid security guarantees, ECC is widely employed in commercial environments. Curves, which are established by the National Institute of Standards and Technology (NIST)[14], are the most often utilized ECs in ECC. In numerous cryptographic protocols and systems, these curves are commonly used and approved. Based on their sizes, the NIST curves are separated into three groups: P-256, P-384, and P-521. The key sizes for these curves are 256 bits, 384 bits, and 521 bits, respectively. The most used curve, the P-256 curve, strikes a fair mix between security and effectiveness. Our requirements sorted possible candidates into two options: EC secp256r1 also known as prime256v1 or its sibling secp256k1. Secp256r1 heavily used publicly as it is standardized by NIST. On the other hand, secp256k1 is created and standardized by Standards for Efficient Cryptography Group (SECG) [15]. However, this curve was not added as NIST standard in the last publication FIPS 186-5 yet [16].That led to public comment raised by block-chain community included ETH Foundation [17]. In spite of this fact, secp256k1 has some benefits over secp256r1. The benefits comes from its fundamental structure of curve that is defined by equation 1.3:

$$y^2 = x^3 + 7 \tag{1.3}$$

As The Weierstrass coefficients (a,b) are static with set values of (0,7), it results in using interesting features that resulted secp256k1 to be curve with fast scalar multiplication. Table 1.2 compares secp256k1 and secp256r1 in terms of signing and verification in Elliptic Curve Digital Signature Algorithm (ECDSA) provided by OpenSSL and Libsecp256k1 libraries.

According to benchmark in table 1.2, speed of the ECs are critical on selection of library as Openssl library is optimized for secp256r1 and libsecp256k1 was primaly

	OpenSSL secp256r1	OpenSSL secp256k1	Libsecp256k1 secp256k1 excl. endomorphism	Libsecp256k1 secp256k1 incl. endomorphism
sign	33000/s	2000/s	22000/s	22000/s
verify	12000/s	2300/s	15000/s	21000/s

Tab. 1.2: Elliptic Curve Speed Comparison [4]

created for secp256k1 curve. When it comes to security, next table 1.3 shows security difference that has currently no effect as both curves are 256 bits.

Curve	secp256k1	secp256r1
Security	127.03	127.83
Automorphism Order	6	2
Parameters a	0	3
Cost for a combine attack	2 ^109,5	2 ^120,3

Tab. 1.3: Security Comparison of secp256k1 and secp256r1 [5]

In terms of security, both ECs are considered secure, with small lead of secp256r1. Since then, secp256r1 has been advocated by standards organizations like the National Security Agency (NSA) and NIST and is increasingly frequently adopted and used in cryptographic protocols and systems. secp256r1 was chosen for implementation of this work since secp256k1 is predominantly used in bitcoin and blockchain applications and may not be as extensively accepted in different environments.

1.2 Digital Signature

In the digital realm, there is a need for an equivalent representation of a hand-written signature with all its properties. A digital signature serves as a unique behavioral biometric that enables easy authentication and guards against signature alteration or accurate falsification. Additionally, it encompasses a valuable attribute known as non-repudiation, which prevents the signer from denying their own signature in the future. Digital signature, a mathematical scheme used for verifying digital messages [6], satisfies the requirements of authentication, integrity, and non-repudiation. It is generally regarded as a signature created using cryptographic methods, specifically employing asymmetric cryptography and hash functions.

1.2.1 Hash function

A hash function is a mathematical procedure that converts a numeric input value into another value.[18] Hash values, or simply hashes, can be understood as representations of fixed-length messages of any length. To create hashes, a hash function operates on two fixed-size blocks of data, typically ranging between 128 bits and 512 bits. The entire process functions as a chain with rounds, where the output of one hash function becomes the input of another. This results in a desirable effect known as the avalanche effect [19]. Nearly identical messages that differ by only one bit produce completely different hashes.

A reliable hash algorithm must satisfy several criteria, as they are commonly used in digital signatures, authentication systems, and databases. One essential property is speed. Additionally, a hash function should possess the following properties. Pre-image resistance ensures that it is difficult to reverse the hash algorithm and deduce the original input from the output. Second Pre-image resistance refers to the concept that given an input and its hash value, finding a different input with the same hash value should be challenging [20]. Since a hash function is a compression function with a specific hash length, collisions are unavoidable. Therefore, comparing two inputs of different lengths that result in the same hash value, commonly known as collision resistance, should be challenging.

The NIST compares the security of the most popular hash functions in the SHA-3 Standard publication [21]. The comparison is presented in the table 1.4. In definition 1 the security strength against second pre-image attacks on a message M is stated as:

Definition 1 *$log_2(len(M)/B)$, where B is the block length of the function in bits, i.e., $B = 512$ for SHA-1, SHA-224, and SHA-256, and $B = 1024$ for SHA-512 [21].*

1.2.2 Secure Hash Algorithm 256-bit

Secure Hash Algorithm 256-bit (SHA-256) is a widely used cryptographic hash function that belongs to the SHA-2 family. It was designed by NSA and later on published by NIST in 2001. In order for SHA-256 to work, the input message must be divided into 512-bit blocks. Next, each block must undergo a series of cryptographic operations that combine and alter the bits. These procedures include conditional assignments, modular arithmetic, bitwise logical operations, and message expansion [22]. These operations provide a 256-bit hash value that is unique to the input message as the end result.

One of SHA-256's major characteristics is its resistance to collisions, which means that finding two separate input messages that give the same output hash value

		Security Strengths in Bits		
Function	Output Size	Collision	Preimage	2nd Preimage
SHA-1	160	<80	160	160–L(M)
SHA-224	224	112	224	min(224, 256–L(M))
SHA-512/224	224	112	224	224
SHA-256	256	128	256	256–L(M)
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	512–L(M)
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512

Tab. 1.4: Security Comparison of Hash Functions [6]

is computationally impossible. Moreover, SHA-256 is considered as deterministic which results by same input same output. Because of these characteristics, SHA-256 may be used for a variety of cryptographic tasks, including password storage, digital signatures, and message authentication codes.

1.2.3 Digital Signature Scheme

The most commonly a digital signature scheme consists of three-stage process [6]; a key generation, signing and a signature verification. The key generation is an algorithm that creates a pair of private and public key. The pair of keys is tied to one entity, who signs a document by private key that is kept in secret. Then, public key is sent to other entity for verification. Signing is generated by hashing document. A outcome of hash function is signed by private key in order to create digital signature and then to be transmitted with document to a receiver. Finally, the signature verification is done by decryption using public key. Algorithm states acceptance or rejection of signature authenticity. Problem can be found in distribution of public key from one entity to another as that is no evidence of key authenticity. Moreover, public key can be tampered by third party. Public Key Infrastructure (PKI) solves this problem by creating third party organization, trusted by both entities, also known as Certificate Authority (CA). they issue certificate on entity and its legitimate key pairs.

1.2.4 Digital Signature Algorithms

Due to the presence of various types of digital signatures, NIST specifies approved digital signature algorithms in the publication Digital Signature Standards (DSS) [14]. This includes the Digital Signature Algorithm (DSA) developed by NIST, as well as the recently added RSA [10] and ECDSA. All of these algorithms work in conjunction with approved hash functions specified in the Secure Hash Standard (SHS) [23] or the SHA-3 Standard [21].

Moreover, alternative digital signatures gains popularity these days. The Schnorr Signature [24] is a relatively old algorithm created by Claus Schnorr, which was under patent protection until 2008. In the same year, Bitcoin was created, and its creator decided to implement ECDSA using the EC secp256k1 due to its optimization and public awareness [25]. However, ECDSA has some drawbacks that the Schnorr Signature solves. Therefore, developers decided to implement the Schnorr signature in the taproot upgrade that took place in November 2021.

1.3 Schnorr Signature

Public key signature techniques are essential for authenticating sensitive messages such as electronic funds transfers and managing access to communication networks. Since the development of RSA, research has focused on improving the effectiveness of these techniques. In 1991, Claus Schnorr introduced a new signature scheme, known as Schnorr signature, with main purpose of minimizing computation for smart cards due to a lack of computing power [26]. The Schnorr signature is a digital signature protocol known for its ease of use, effectiveness, and concise signatures. It is based on the idea of public keys and is widely used in many different cryptosystems. The Discrete Logarithm Problem (DLP), extracted from the Schnorr identification method using the Fiat-Shamir heuristic, serves as the basis for the scheme [27]. The scheme's security has been investigated and proven in the Random Oracle Model (ROM). It has been demonstrated to be strongly resistant to forging during adaptively chosen-message attacks[27]. One of the advantages, that has noticeable effect is linearity that is desirable in multi-party computation. This useful attribute enables the creation of another valid Schnorr signature by combining two Schnorr signatures, resulting of possible algebraic operations in signatures. Following algorithms describe signing and verification process in Schnorr signature [26]. Let H be a cryptographic hash function that maps to $\mathbb{Z}_q^*$ and let $\mathbb{G}$ be a group with generator g and prime order q. The following actions are taken to generate a Schnorr signature over a message m:

Algorithm 1 Signing Algorithm for Schnorr Signature [26]

```
1. Select random nonce $k \in_R \mathbb{Z}_q$
2. Calculate the commitment $R \leftarrow g^k \in \mathbb{G}$
3. Calculate the challenge $c = H(m, R)$
4. Calculate the response with secret key $S_k$ $z = k + S_k * c \in \mathbb{Z}_q$
5. Signature is defined as $\sigma = (z, c)$
```

Algorithm 2 Verification Algorithm for Schnorr Signature [26]

```
1. $Parse\ \sigma\ into\ (z, c)$
2. Calculate $R' = g^z * P_k^{-c} \mid P_k = g^{S_k}\ (Public\ key)$
3. Calculate $z' = H(m, R')$
if $c = z'$ then
    output is 1; Valid
else
    output is 0; Rejected
end if
```

1.3.1 EC-Schnorr Signature

Since introduction of basic Schnorr signature by Claus Schnorr, another versions of this type were proposed. Significant improvement is provided by collaboration with ECC, where EC are used for calculating parameters in result of faster computing. As a consequence, small changes are done in signing and verifying algorithm. Following table compares some types of Schnorr signatures.

Scheme	Schnorr Sig.	EC-SDSA	EC-FSDSA	Schnorr Sig. BIP 340
1. Component	$H(m, R)$	$H(R_x \| \| R_y \| \| m)$	$R_x \| \| R_y$	R_x
2. Component	$k + S_k * h$	$k + S_k * h$	$k + H(R_x \| \| R_y \| \| m) * S_k$	$k + H(R_x \| \| P_{Kx} \| \| m) * S_k$
Sign. Size	b+2b	2b+2b	4b+2b	2b+2b
Public Key	g^{S_k}	$-S_k * G$	$-S_k * G$	$S_k * G$
Reference	[26]	[28]	[29]	[30]

Tab. 1.5: Comparison of Different Types of Schnorr Signature

Nowadays, one of the most breaking news in cryptography is implementation of EC-Schnorr signature into Bitcoin that took place in November 2021. It was well-grounded by Bitcoin Improvement Proposal (BIP), more specifically BIP 340 [30]. It is considered as standard for 64-byte EC-Schnorr signature algorithm that is performed over secp256k1 EC. Following algorithms show how signing and verifying is done in this work:

Algorithm 3 Signing Algorithm for EC-Schnorr Signature

```
1. Select random nonce $k \in_R \mathbb{Z}_q$
2. Calculate the point on curve $R = k * G$
3. Calculate hash $c = H(R||m)$
4. Calculate the challenge with secret key $S_k$ $s = k + c * S_k$
5. Signature is defined as $\sigma = (s, c)$
```

Algorithm 4 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.

1.4 Multi-signatures

In order to secure communication and data sharing on IoT 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.

1.4.1 Classification

Threshold multisignature schemes and distributed multisignature schemes are the two basic kinds of multisignature schemes.

1. For **Threshold signature** to be legitimate, a certain minimum number of signers must take part in the signing process according to this method. Two more classes may be added to the classification of threshold multisignature schemes:
   - **Secret sharing-based multisignature** methods proposed by Shamir Using Shamir's secret sharing technique, the message is first divided into shares, and each side creates a partial signature on their corresponding share. A reconstruction procedure is used to combine the partial signatures to create the whole signature. With this strategy, the signing job is divided among the signers, but the shares must be created and distributed by a reputable dealer.
   - **Schnorr's threshold signature-based multisignature** schemes: In this method, each party signs the message partially using their private key, and the partial signatures are then merged with a threshold signature algorithm to create the final signature. While Shamir's secret sharing-based schemes have a lower computational overhead, this strategy does away with the requirement for a trusted dealer.
2. **Distributed multisignature** techniques permit any subset of signers to cooperatively create a signature on the message rather than requiring a minimum number of signers to participate in the signing process. Two more classes may be added to the classification of distributed multisignature schemes:
   - Multisignature techniques based on **ring signatures** use a public key ring that contains the public keys of all signers, each party creates a ring signature on the message in this method. Using a verification technique, the ring signatures are combined to get the final signature. Although this method offers signers anonymity, it has a higher computational cost when compared to other multisignature schemes.
   - **Aggregate signature-based multisignature** schemes depends on each participant as they create a unique signature on the message using their private key, and these signatures are then merged to make the final signa-

ture using an aggregate signature algorithm. This method has a minimal computational cost and is appropriate for devices with limited resources, but it needs a reliable third party to combine the signatures.

1.5 Multi-signature Key Generation

As in previous section was mentioned, key generation is the most complicated part in multisignatures. The fact, that multiple untrusted parties have to jointly accept and distribute public and private key without leaking secret information makes key generation problematic. Moreover, final public key has to be a function of secret key. In other words, key generation has to fulfill terms of privacy and correctness. There are 2 main techniques that are used in Multi-signature key generation.

1.5.1 Shamir's Secret Sharing

Shamir's Secret Sharing (SSS) is a cryptographic procedure that divides a secret into shares, with the result that the original secret can only be recreated if enough shares are joined. Adi Shamir created it in 1979 [31], and today it is extensively used for many different purposes, such as secure communications and multi-party calculations.

The key concept underlying SSS is to create shares of a secret via polynomial interpolation. To be more precise, we can produce N points on a random polynomial of degree $N-1$, where the secret value S is the constant term, given a secret value S and a positive integer N [31]. In order to prevent any party from learning anything about the secret value from only their share, these N points can be divided to N separate parties. Any T or more parties are able to combine their shares using polynomial interpolation in order to recreate the secret value. In other words, we can calculate the special degree $T-1$ polynomial that goes over any T shares (where T is less than or equal to N). The secret value S serves as the polynomial's constant term.

SSS is secure since an attacker cannot discover any information about the secret value from less than T shares. The reason for this is that any polynomial with degree less than $T-1$ may be constructed to pass through an endless number of points, hence the shares by themselves are meaningless in revealing the secret[31]. It has many advantages and useful properties such as: scalability, flexibility, robustness and efficiency as only simple arithmetic algorithms are required.

SSS has the drawback of requiring a reliable dealer to create and distribute the secret shares. In order to maintain the confidentiality of the shared secret, the dealer must be trustworthy and cannot collaborate with any of the parties. If the dealer is dishonest or malevolent, they may distribute the wrong shares or disclose information that compromises the secrecy of the secret. Therefore, in some schemes is not desired to have centralized power in hand of one entity that brings us to point of failure.

1.5.2 Distributed Key Generation

A cryptographic technique called Distributed Key Generation (DKG) is used to create cryptographic keys in a distributed and safe way without depending on a reliable dealer. DKG enables a group of participants to collectively produce a shared secret that may be used as a cryptographic key for a variety of purposes, including encrypted communication or digital signatures.

The Pedersen DKG protocol, first forward by Torben Pedersen in 1991 [32], serves as a prime example of DKG. SSS and ECC are used in the Pedersen DKG protocol to provide a shared secret key that is safely generated. Generaly any DKG scheme, Pedersen DKG included, consist of following stages:

1. **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 [32].
2. **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 [32].
3. **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.
4. **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.

1.6 Lagrange Polynomial Interpolation

A mathematical method called Lagrange polynomial interpolation is used to identify a polynomial function that traverses a collection of known points on graph [33]. In various fields, including cryptography and its cryptographic systems like SSS, which is used to safely divide a secret into numerous shares, the Lagrange polynomial interpolation is utilized.

The general form of the Lagrange polynomial interpolation is [33]:

$$P(x) = \sum_{i=0}^{n} y_i \, l_i(x) \tag{1.4}$$

where $l_i(x)$ is the ith Lagrange basis polynomial, defined as [33]:

$$l_i(x) = \prod_{j=0 j\neq i}^{n} \frac{x - x_j}{x_i - x_j} \tag{1.5}$$

The characteristic of the Lagrange basis polynomials is that for any j i, $\quad l_i(x_i) = 1$ and $l_i(x_j) = 0$. This indicates that the point (x_i, y_i) is the only point through which the polynomial function $P(x)$ passes and not any other points [33].

1.7 Pailler Cryptosystem

The Paillier cryptosystem is a probabilistic asymmetric method used in public-key cryptography, and it was developed by Pascal Paillier in 1999 [34]. A trapdoor mechanism developed from the family of trapdoors based on the DLP is shown in Paillier's work [34] with an emphasis on composite residuosity classes.

This cryptographic method has the additive homomorphic characteristic, which allows two ciphertexts to be combined without impairing the outcome. Decryption is not required since the computation operates as if the appropriate plaintexts were simply added.

However, for any homomorphic encryption technique, the effectiveness of the Paillier cryptosystem is an an ongoing concern. To solve this problem, several optimization ideas have been presented up. Paillier himself, for instance, suggested Scheme 3 as a modification of the initial Scheme 1. Moreover, another way of improvement includes computing certain values in advance, such as exponentiating either the message g^m or the noise r^n, as suggested in the article [35].

1.7.1 Homomorphic Properties

The additive and multiplicative homomorphic qualities of the Paillier scheme are its major features. These characteristics make it possible to simulate addition and multiplication operations on ciphertexts while computing addition and multiplication operations on plaintexts. This is stated mathematically as follows [1]:

- **adding homomorphically** the two related plain-texts together is identical to decryption the product of two ciphers (or the product of a cipher and a generator g raised to the power of the plain-text):
  - $D(E(m_1, r_1) \cdot E(m_2, r_2) n^2) = (m_1 + m_2) \pmod{n}$
  - $D(E(m_1, r_1) \cdot g^{m_2} \pmod{n^2}) = (m_1 + m_2) \pmod{n}$

- plaintexts are **multiplied homomorphically**. Decrypting the result in this instance reveals the multiplication of the two plaintexts when a ciphertext is raised to the power of a plaintext:
  - $D(E(m_1, r_1)^{m_2} \pmod{n^2}) = (m_1 m_2) \pmod{n}$
  - $D(E(m_2, r_2)_1^{m} \pmod{n^2}) = (m_1 m_2) \pmod{n}$

1.7.2 Pailler Scheme 1

The following procedure is used to create the public key *pk* and secret key *sk* in Scheme 1. P and Q, two large prime numbers, are first chosen at random to ensure their independence from one another. Both primes must be of similar length in order to ensure a certain property $(GCD(PQ, (p-1)(q-1)) = 1)$. The parameter is then calculated as the least common multiple of $(p-1)$ and $(q-1)$ and the parameter n is then calculated as the product of P and Q [1]. The parameter g is then drawn at random from the set $\mathbb{Z}^*_{n^2}$ after that. It's crucial to confirm that n divides g by looking for a modular multiplicative inverse, given by the symbol. At this stage, it is necessary to ensure that n divides the parameter g by verifying the existence of a modular multiplicative inverse denoted as μ showed in equation 1.4 [1].

$$\mu = (L(g^{\lambda} \pmod{n^2}))^{-1} \pmod{n} \tag{1.6}$$

where $L(x) = \frac{x-1}{n}$. The public key *pk* is defined as the ordered set consisting of the parameters n and g. On the other hand, the secret key *sk* is defined as the ordered set containing the parameters λ and μ. The **encryption** of message m is done as in equation 1.5 [1].

$$c = g^m \cdot r^n \pmod{n^2} \tag{1.7}$$

On the other hand, the **decryption** of cipher-text is done as in equation 1.6 [1].

$$m = (L(c^{\lambda} \pmod{n^2})) \cdot \mu \pmod{n} \tag{1.8}$$

1.8 FROST signature

1.8.1 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].

1.8.2 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. Each participant in this procedure creates a random polynomial whose degree is equal to the threshold value $t - 1$. The other coefficients are selected at random, and the polynomial's constant term is set to their secret share. Following that, each participant broadcasts their polynomial to the entire group. After receiving polynomial from all participants, polynomials are verified with previously shared commitments that consists of random values chosen by participant. If verification holds, participant moves on for key generation, otherwise protocol is aborted. Secret key of participant is generated by sum of all received polynomial, while public key is computed as a linear combination of the individual public keys using Lagrange polynomial interpolation. Specifically, each participant evaluates their polynomial at a designated point, and then computes their individual public key as a scalar multiple of the group generator raised to the participant's secret share. Pedersen DKG is done in 2 rounds. Since FROST in this thesis is based on EC computations, DKG is done as following (Please, consider a 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: $Y = \sum_{j=1}^{n} \phi_{j,0}$

1.8.3 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 prerequisite to participate in signing operation. Overall, both versions are based on same computations.

Secondly, FROST has options of signing in terms of aggregator role [2]. Without an aggregator, each signer contributes their own signature to the message. The total of the individual signatures is then calculated to create the signature. This indicates that the signing procedure requires the presence of all participants, and the signature cannot be calculated if any member is unavailable or unresponsive.

An aggregator is a single participant who gathers the partial signatures from the other participants while using this method of signing. The incomplete signatures are then combined by the aggregator to create a complete signature. The benefit of this strategy is that just the aggregator has to be present when signing documents. The signature procedure may be postponed until the aggregator is back online or responsive if it is unavailable. It has also disadvantage that, aggregator has to be honest, but to ensure security aggregator can be made randomly and changed for each signing process.

Signing part can be divided into 3 phases (Please, consider a 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 $z = \sum\limits_{i=1}^{t} z_i \pmod{Q}$
   - Release signature $\sigma = (z, c)$ along with message m [2]

Released signature σ is verified as standard EC-Schnorr signature by public key Y with algorithm 4 in subsection 1.3.1.

1.9 Threshold Signature for Privacy-preserving Blockchain

Threshold signature presented in [9] abbrev. (TSPB) is focused on increasing security and privacy in blockchain technology. [9] provides a method for distributing a Blockchain wallet across several devices safely. It is possible to implement this divide for either a single user, which increases security by requiring multiple user's devices to sign Blockchain transactions, or for an entire group of users that collaborate, which promotes privacy by allowing anonymous signature on behalf of the shared Blockchain wallet [9]. The signature is based on cryptographic primitives that have been demonstrated to be secure, including the Schnorr signature, Pailler cryptosystem, and SSS.

1.9.1 Setup Algorithm

The proposed approach necessitates the collaboration of a subset of registered devices, specifically t out of n, in order to retrieve the secret key. To achieve this, an utilization of SSS scheme along with the Paillier cryptographic scheme is essential in setup part. This combination ensures a secure distribution of the client authentication secret key, which is computed as the sum of individual secret keys sk belonging to the devices and the client. The resulting share is then employed as the secret key for the respective device.

A polynomial made up of randomly generated values (d_i, t) is created throughout the distribution process. Here, i stands for the device number, while t stands for the threshold value, which is related to the degree of the polynomial. The polynomial is defined in accordance with a system of N devices by following equation 1.7.

$$f(x) = (d_{1,1} + \ldots + d_{N,t})x^t + \ldots + (d_{N,1} + \ldots + d_{N,t})x + \sum_{i=1}^{N} \kappa_i \tag{1.9}$$

where $\sum_{i=1}^{N} \kappa_i$ the client device as well as other devices' secret keys are added up. By adding the terms of the polynomial $d_{i,1}x^t + \ldots + d_{i,t}x + \kappa_i$, where each value of x is encrypted using the Paillier scheme, one can obtain the summations of $d_{1,t}, \ldots d_{N,t}$ and κ_i. The authentication sk calculation may be partially executed on each device thanks to this encryption, guaranteeing that none of the secret keys ever leave their respective devices.

Setup algorithm can be divided into 2 rounds computed by number of n participants:

1. **Parameter Generation**
   - generate random values $d_{1,t}, \ldots d_{N,t}$
   - generate the Pailler's key pair $(pk_{p,j}, sk_{p,j})$
   - generate random secret k_j
   - calculate $pk_j = g^{k_j}$

   The entire secret distribution process proceeds as follows for the calculation of each $f(x_j)$ where j ranges from 1 to n. Consider the variable h, which is defined as $j + 1$:
2. **Polynomial Evaluation**
   - D_h generates random value $r_{j,h}$ and compute $\chi = Enc_{pk_{p,j}}(x_j, r_h)$
   - D_h generates random value $v_{j,h}$ and compute

   $$c_h = \chi_{j,h}^{x_j^{t-2} * d_{t-1}^{(h)}} \chi_{j,h}^{x_j^{t-3} * d_{t-2}^{(h)}} * \ldots * \chi_{j,h}^{d_1^{(h)}} * Enc(k_j, v_{j,h}) \tag{1.10}$$

   - if $h = j + 1$, then D_h sends c_h to D_{h+1}
   - if $h \neq j$, then $h = h + 1 \pmod{n}$ and go to first step of the polynomial evaluation
   - if $h = j$, then D_j computes:

   $$f(x_j) = Dec(c_{j-1}) + d_{t-1}^{(j)} x_j^{t-1} + \ldots + d_1^{(j)} x_j + k_j \tag{1.11}$$

1.9.2 Signing Algorithm

Signing part and later on verification of final signature is identical with FROST scheme in this article. First of all, t out of n participants need to be agreed to issue signature σ. In proposed work [9] participants are divided into **Main Device** (MD), who has enabled signing mode and **Secondary Device** (SD) with co-signing mode. Therefore, for optimization and better understanding MD can be considered as participant and aggregator in one entity as MD participates in setup and signing part. Then, signing algorithm is followed in subsection 1.8.3 consisted of 3 phases: **Commitment, Challenge** and **Signature phase**.

Released signature σ is verified as standard EC-Schnorr signature by public key Y with algorithm 4 in subsection 1.3.1.

2 Implementation

This chapter is dedicated to the practical part of the thesis, focusing on the comparison of different types of multi-signature schemes in terms of computational complexity and efficiency. Based on the obtained results, the most suitable signature scheme is selected and implemented for IoT. The next section focuses on the implementation of multi-signature in the programming language C, with carefully chosen and included essential libraries.

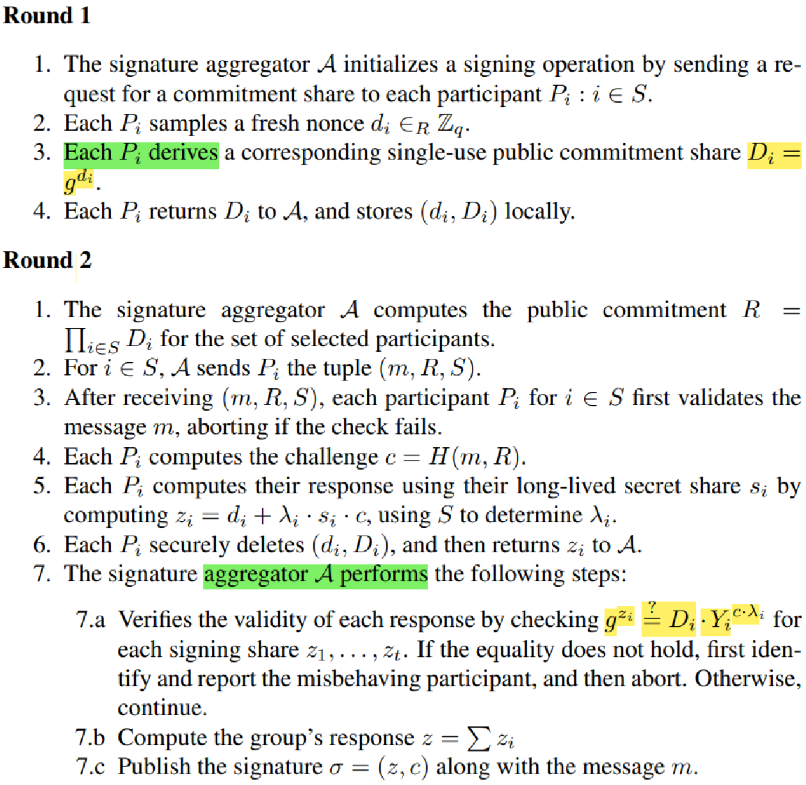
2.1 Multi-signature Comparison

During the research on the defined problem, five potential candidates were selected for further implementation of multi-signature for IoT. 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. 2.1: Comparison of Multi-signatures

Table compares 5 different types of securely-proven multi-signatures. All signatures are compatible with Schnorr signature except mBCJ. They can be divided into 2 main types of Threshold and Multi-Signature (Mu-Sig). Result of that is the different number of parties involved for signature as Mu-Sig requires all number of participants n and on the other hand, threshold requires only group of participant t from all participants n. A representative of threshold is FROST that is precisely presented in Komlo and Goldberg [2] with 2 variants; FROST requiring 1 (FROST ver. 1) or 2 (FROST ver. 2) iterations for signing. FROST security is based on One-More Discrete Logarithm (OMDL) and Programmable Random Oracle Model (PROM) assumptions [36]. A valuable tool for demonstrating cryptography methods and highlighting probable assumptions that might or might not hold true in practice is the security model. According to PROM, the execution environment (which executes the adversary and simulates answers to the adversary's oracle queries) is permitted to program the random oracle, but only if the programming is identical to all other truly random responses [36]. In FROST key generation is done by DKG with protocol called Pedersen's DKG that takes 2 iterations [2]. Difference between FROST with 1 round and 2 rounds is that 1 round misses generation of public share commitments that is done in preprocess stage autonomously operated as a requirement to take part in next signing processes. Therefore, all participants must have access to the commitment server role since it manages and stores the participant's commitment shares [2]. Moreover, it changes and reduce signing rounds with slightly different structure in FROST 2 rounds. As an example can be shown picture of FROST 2 rounds signing algorithm with steps for calculating of exponentiation:



Round 1

1. The signature aggregator $\mathcal{A}$ initializes a signing operation by sending a request for a commitment share to each participant $P_i : i \in S$.
2. Each P_i samples a fresh nonce $d_i \in_R \mathbb{Z}_q$.
3. Each P_i derives a corresponding single-use public commitment share $D_i = g^{d_i}$.
4. Each P_i returns D_i to $\mathcal{A}$, and stores (d_i, D_i) locally.

Round 2

1. The signature aggregator $\mathcal{A}$ computes the public commitment $R = \prod_{i \in S} D_i$ for the set of selected participants.
2. For $i \in S$, $\mathcal{A}$ sends P_i the tuple (m, R, S).
3. After receiving (m, R, S), each participant P_i for $i \in S$ first validates the message m, aborting if the check fails.
4. Each P_i computes the challenge $c = H(m, R)$.
5. Each P_i computes their response using their long-lived secret share s_i by computing $z_i = d_i + \lambda_i \cdot s_i \cdot c$, using S to determine λ_i.
6. Each P_i securely deletes (d_i, D_i), and then returns z_i to $\mathcal{A}$.
7. The signature aggregator $\mathcal{A}$ performs the following steps:

   7.a Verifies the validity of each response by checking $g^{z_i} \stackrel{?}{=} D_i \cdot Y_i^{c \cdot \lambda_i}$ for each signing share $z_1, \ldots, z_t$. If the equality does not hold, first identify and report the misbehaving participant, and then abort. Otherwise, continue.

   7.b Compute the group's response $z = \sum z_i$

   7.c Publish the signature $\sigma = (z, c)$ along with the message m.

Fig. 2.1: Signing Algorithm for 2-round FROST [2]

Round 1 is mainly set up of public commitment that in FROST ver. 1 preprocess stage deals with it. As every single participant t has to calculates single-use public commitment first exponentiation is equal to t participants. Round 2 is practically similar for both FROST versions. Public group commitment is computed by the signature aggregator who also verifies response validity of participants that contains 2 exponentiation. This is also done in FROST ver. 1, but single-use commitments are taken commitment server.

The best representative of Mu-sig group is MuSig2, introduced in Nick et al. article [37], with sufficient security based on OMDL assumption. MuSig2 is fast and robust, but requires all participants that in some occasions is not the ideal option. Another Multi-signature scheme is BN06 (brings in Bellare and Neven article [38]) based on DLP, regarded as a "standard assumption" in the field of cryptography, supported with ROM, presuming that hash function outputs are identical to random values. Lastly, multi-signature mBCJ, firstly mentioned by M. Drijvers et al. [39]. It is in many ways similar to BN06, but does not support Schnorr signature, thus for this implementation is not appropriate.

2.2 Implementation of OpenSSL 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.

OpenSSL [40] is a strong and well-liked software library that offers developers useful cryptographic functionalities. Therefore, the implementation of FROST signature is based on this library. More information about the library itself can be found in chapter 3. Used version of OpenSSL library is following: OpenSSL 3.0.7 1; Nov 2022. OpenSSL version 3 has some new updated functions that replace older deprecated function.

2.2.1 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. Hash function implementation satisfy the latest changes in OpenSSL library as functions replacing deprecated functions are implemented as in the listing 2.1:

```
/*declares an array of unsigned characters
to store the value of 256 bits*/
unsigned char hash[SHA256_DIGEST_LENGTH];

/*struc holding the context for a message digest operation*/
  EVP_MD_CTX* mdctx;

/*struct, that represents the message digest algorithm*/
  const EVP_MD* md;

/*select the SHA-256 algorithm*/
  md = EVP_sha256();

/*allocates and initializes a new struct*/
  mdctx = EVP_MD_CTX_new();
/*initializes the message digest context*/
  EVP_DigestInit_ex(mdctx, md, NULL);

/*updates the message digest context with the input data*/
  EVP_DigestUpdate(mdctx, concat_string, hash_len);
/*finalizes computation and stores hash value into array*/
  EVP_DigestFinal_ex(mdctx, hash, NULL);

/*free allocated memory*/
  EVP_MD_CTX_free(mdctx);
```

Listing 2.1: Implementaion code of SHA-256

2.2.2 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 [15]:

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

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 [15]. Generator G, order Q and modulus p are initialized as in the listing 2.2. The point is serialized into a byte array since function a $EC\_POINT\_point2bn$ has been deprecated since OpenSSL 3.0.

```
void initialize_curve_parameters() {
  // initialize curve
  ec_group =
  EC_GROUP_new_by_curve_name(NID_X9_62_prime256v1);

  // retrieves point of the EC group
  p_generator = EC_GROUP_get0_generator(ec_group);
  // serialize the point into a byte array
  size_t buf_len = EC_POINT_point2oct(
      ec_group, p_generator, POINT_CONVERSION_UNCOMPRESSED,
      NULL, 0, NULL);

  unsigned char* buf = OPENSSL_malloc(buf_len);
  EC_POINT_point2oct(ec_group, p_generator,
  POINT_CONVERSION_UNCOMPRESSED, buf,
                     buf_len, NULL);

  // create a BIGNUM from the byte array
  b_generator = BN_bin2bn(buf, buf_len, NULL);

  order = EC_GROUP_get0_order(ec_group);
  modulo = EC_GROUP_get0_field(ec_group);

  // free the memory allocated for buffer
  OPENSSL_free(buf);}
```

Listing 2.2: Implementaion code of Secp256r1

The implementation uses uncompressed generator G in form [15]:

$G = 04\ 6B17D1F2\ E12C4247\ F8BCE6E5\ 63A440F2\ 77037D81\ 2DEB33A0$
$F4A13945\ D898C296\ 4FE342E2\ FE1A7F9B\ 8EE7EB4A\ 7C0F9E16\ 2BCE33$
$57\ 6B315ECE\ CBB64068\ 37BF51F5$

Finally, the order Q is defined as following [15]:
$Q = FFFFFFFF\ 00000000\ FFFFFFFF\ FFFFFFFF\ BCE6FAAD$
$A7179E84\ F3B9CAC2\ FC632551$

2.2.3 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 OpenSSL.

The specified RAND method, a collection of instructions for producing random numbers, determines the algorithm used by RAND_bytes() [41]. The Deterministic Random Bit Generator (DRBG) technique is used as the default RAND method in OpenSSL 3.0 from NIST SP 800-90A [42]. DRBG belongs to a group of Pseudo-Random Number Generators (PRNGs) that are cryptographically safe. To produce pseudo-random output, a number of cryptographic primitives are used, such as hash functions, block ciphers, and Message Authentication Codes (MACs) [41]. The process is referred to be deterministic since only the seed value and any other inputs, such customization or entropy, have any bearing on the final result. Even with a compromised internal state, the DRBG algorithm is built to offer a high level of security and predictability. However, it is important to implement randomization securely with later memory cleaning. Function $generate\_rand()$ is called every time, 32 byte random number is needed. Function is implemented as in listing 2.3:

```
BIGNUM* generate_rand() {
  unsigned char buffer[NUM_BYTES];
  BN_CTX* ctx = BN_CTX_new();
  BIGNUM* result = BN_new();
  // generate random bytes
  if (RAND_bytes(buffer, NUM_BYTES) != 1) {
    printf("Error␣generating␣random␣bytes\n");
    exit(EXIT_FAILURE);}
  // convert buffer to a bignum mod Q
  BIGNUM* rand_num = BN_bin2bn(buffer, NUM_BYTES, NULL);
  BN_mod(result, rand_num, order, ctx);

  OPENSSL_cleanse(buffer,sizeof(buffer));
  BN_CTX_free(ctx);
  BN_clear_free(rand_num);
  return result;}
```

Listing 2.3: Generation of 32-byte Random Number

2.3 FROST Implementation

FROST signature is implemented in C, since it is low level program language that is resulting high performance and possibility for IoT device implementation. Implementation consists of following .c files: *main.c*, *setup.c*, *signing.c*, *globals.c* and *macros.c*. .c files are then linked with header files located $../headers$ and with headers of OpenSSL library that has to be downloaded to OS. Header files in $../header$ are following: *setup.h*, *signing.h* and *globals.h*. Whole project is built by $Makefile$.

The project is meant to be programmed as library with API that is run by *main.c* which tests whole library with a result of released signature and its verification. Structure of API and communication between participant is showed in following subsection. Please, be aware of $(2, 3)$ FROST is showed in the thesis for simplicity, but implementation is defaulted set as $(3, 5)$. Since communication between participants is needed link list algorithm is used for storage and later use of packets.

2.3.1 FROST Setup

Setup of FROST is basically implementation of Pedersen-DKG that is done withing 2 rounds. At the beginning function *init_pub_commit*() is called by every participant P_i. Public commit packet is created by this function holding index of sender, length of public commit array and finally public commit array. Within the function $init\_coeff\_list()$ is trigged which results in creation of array of random numbers that has length of n. 32 bytes random numbers are used in later public commitment. As every P_i created own public commitment, broadcast to all participants P_j is done. After the packets are received by all the participants, every participant calls function *init_sec_share*() for each n participant for creating polynomial. Function takes parameters such as address of participant P_i that sends secret share and index of participant P_j that will receive the share. Please, be aware of participant P_i sending secret share itself as it is not done internally. When participant P_i created secret share for participant P_j, secret share is verified and accepted/denied by P_j with function *accept_pub_commit*(). Mathematical function mentioned in section 1.7.2 is used for verification of secret share by function *accept_pub_commit*(). If verification hold participant P_i secret share is accepted and stored, otherwise the protocol is aborted. When all participants P_i finished verification of all secret shares, key generation is done with function *gen_keys*(). 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:

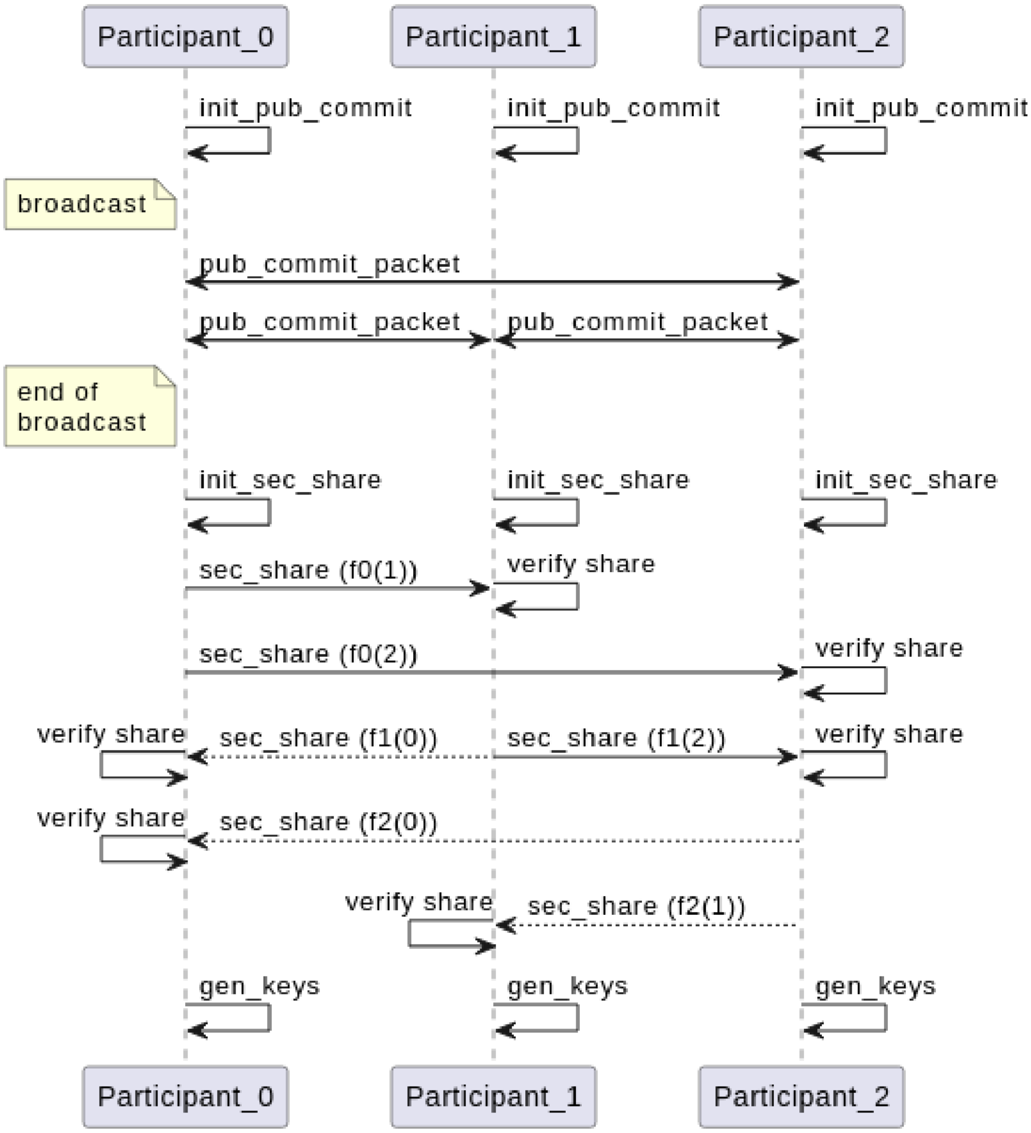


Fig. 2.2: FROST Setup Diagram

2.3.2 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 $threshold\_set$[]. Every participant P_i calls function $init\_pub\_share()$ that results in creation of public share packet. The packet consist of index of sender P_i, P_i's verify share, single use commitment share and public key. The packets are then received by aggregator with function $accept\_pub_share()$. Received data are stored and next step is to generate tuple with function $init\_tuple\_packet()$ by aggregator. Within this function other 2 functions are trigged $R\_pub\_commit\_compute()$ and $pub\_share\_mul()$ that provide check if all public share packets were received from $treshold\_set$[] participants.

If yes, R is computed by function $pub\_shares\_mul()$, otherwise protocol is aborted. Tuple packet consists of message m, size of m, public commitment R, set of participants t and size of set. Tuple is received by every participant P_i with function $accept\_tuple()$. As long as tuple data are stored, function $init\_sig\_share()$ is called by every participant P_i. Function is in charge of creating partial signature for every P_i. $Hash\_func()$ and $lagrange\_coefficient()$ functions are called by the function as essential part for partial signature. When all partial signatures are created, they are sent to aggregator by participants P_i. For verification and acceptance of partial signature $accept\_sig\_share()$ is called. Verifying is done by function described in subsection 1.7.3 followed by storing the partial signature if verification holds or by aborting the protocol if function is broken. 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.

2.3.3 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 in subsection 1.3.1 by Algorithm 4. 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.

2.3.4 Implementation of Link List

For storing and organizing data, a fundamental data structure was chosen in the implementation, called singly linked list. In this kind of linked list, there is only one way in which the linked list may be traversed, where each node's next pointer links to a different node, but the last node's next pointer points to $NULL$. Two main operation are applied in the implementation: **insertion** with time complexity $O(1)$ and **search** with time complexity $O(n)$.

During setup part link list is applied as participants communicates between each other as they need to share $pub\_commit\_packets$ and later sec_shares. For accepting $pub\_commit\_packets$ each participant P_i stores the last node of packet. By function $accept\_pub\_commit()$ one of 2 functions is trigged; $create\_node\_commit()$

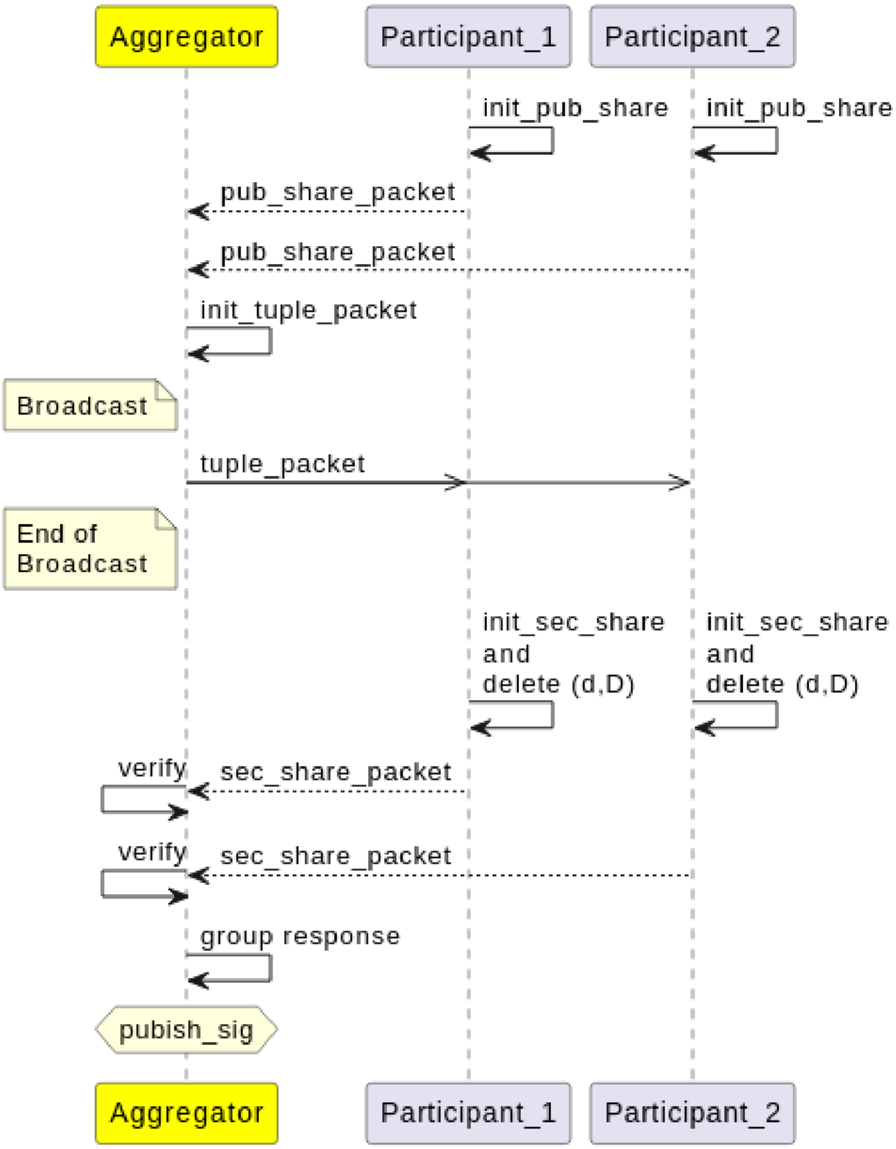


Fig. 2.3: FROST Signing Diagram

function, if participant did not accept any *pub_commit_packet*, otherwise *insert_node_commit*() is used. Every participant P_i after all *pub_commit_packets* are stored is searching in its list in *accept_sec_share*() function, as the verification of *sec_shares* from participants P_j is validated towards earlier accepted *pub_commit_packet* from participant P_j. Similar methodology is used for storing *sec_shares* by function *accept_sec_share*() that calls once function *create_node_share*() and then every time *insert_node_share*() is used. Searching in this list is not necessary. Finally, traversing of both lists is necessary during generation of public key and participant's secret share in functions *gen_sec_share*()

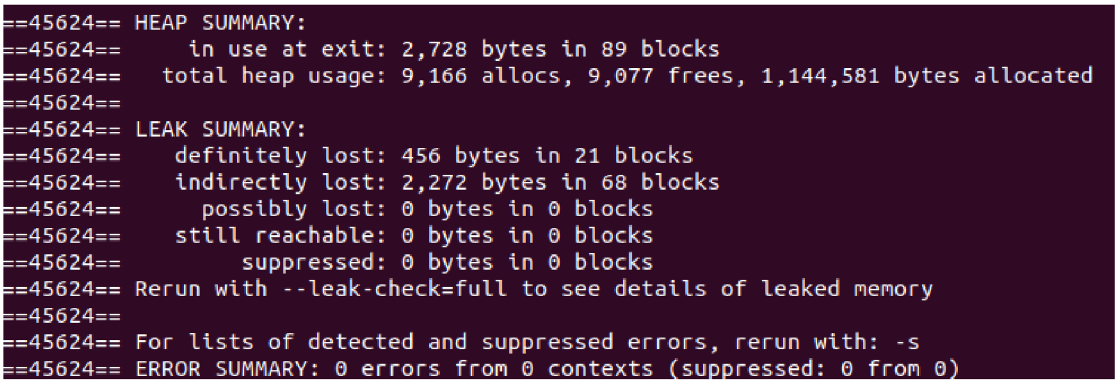
and $gen\_pub\_key()$.

In signing part, link list algorithm is used during communication of participants with aggregator as he need to store $pub\_share\_packets$ and later on sig_shares. $pub\_share\_packets$ are stored by function $accept\_pub\_share()$ with help of functions $create\_node\_pub_share()$ and then $insert\_node\_pub\_share()$. Each pub_share_packet from participants P_i has to be search in the list for later verification of sig_share_packet of participants P_i. Finally, aggregator stores each sig_shares by function $accept\_sig\_share()$ with similar steps of calling function $create\_node\_sig\_share()$ and then $insert\_node\_sig\_share()$. At the end, aggregator traverses this list for sum of all received sig_shares to create final signature.

2.3.5 Security of the Implementation

For overall security of the implementation, security of protocol, used secure cryptographic primitives, used secure library with proper functions and lastly secure allocation/de-allocation of memory has to be taken into account. Since FROST is considered as secure, in previous sections were defensed secure cryptographic primitives such as EC with proper parameters, secure hash function and random generator. Also the implementations is based on secure OpenSSL library where functions with the highest precision were chosen to satisfy updated library of version 3.0. Therefore, the last thing for considering the implementation as secure, allocation/deallocation of memory has to be proven. Library is designed to free memory for used, therefore no action is needed in API from user. Memory is allocating and then freed simultaneously within the functions in the most cases. However, library and also participants operates and store sensitive data that in case of some leak or attack would be destructive for protocol. Therefore, 3 main clearing are done by library, despite of simultaneous freeing of unnecessary variables. At the end of setup, after generation of keys, all secret shares, commitments, coefficient lists and polynomials are securely freed. Lastly, after initializing of partial signature every participant deletes all data followed by aggregator after publishing the final group signature. Lastly, parameters of EC are freed after verification of the signature. For memory free of sensitive data function $BN\_clear\_free()$ is used. The big number (BN) library in OpenSSL has the $BN\_clear\_free()$ function, which is used to deallocate memory allocated to a large number once it is no longer required. This function is written as a macro that first uses the $BN\_clear()$ function to clear the contents of the big number and then uses the $OPENSSL\_free()$ function to release the memory. While the $OPENSSL\_free()$ method deallocates the memory allocated to the big number itself, the $BN\_clear()$ function resets the value of the large number to zero and releases any memory allocated to retain the value [43].

For memory testing Valgrind open-source tool was used [44]. It is tool used for debugging and profiling programs including memory leaks based on Linux and other Unix-based operating systems. The results are followed in the next figure:



```
==45624== HEAP SUMMARY:
==45624==     in use at exit: 2,728 bytes in 89 blocks
==45624==   total heap usage: 9,166 allocs, 9,077 frees, 1,144,581 bytes allocated
==45624==
==45624== LEAK SUMMARY:
==45624==    definitely lost: 456 bytes in 21 blocks
==45624==    indirectly lost: 2,272 bytes in 68 blocks
==45624==      possibly lost: 0 bytes in 0 blocks
==45624==    still reachable: 0 bytes in 0 blocks
==45624==         suppressed: 0 bytes in 0 blocks
==45624== Rerun with --leak-check=full to see details of leaked memory
==45624==
==45624== For lists of detected and suppressed errors, rerun with: -s
==45624== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fig. 2.4: Summary of Memory Leak

It can be noticed that library does keep 30 blocks per participant after ending of the protocol. These blocks are mainly allocated memory of final signature and public attributes such as, public key and verify key. Finally, each computed secret share of participant can be kept for potential later use in another signature. Overall, all data that could lead to reconstruction of keys or final signature are securely freed during the protocol. Therefore, the implementation can be considered as secure, depending under deeper circumstances of later use in production.

2.3.6 FROST Benchmark

Benchmark was done by library *time.h* on setup and then signing part separately. Each participant had index set to size of up to 2^{30} bit number. Main purpose is to notice an impact of number of participants in each operation. 3 different schemes are compared in following table in seconds:

It can be noticed that all iterations were very consist with small difference in time by each scheme. Setup is done by all participants n due to construction of keys. Therefore, setup by 3,5 and 6 participants is compared by table. However, it can be noticed as more participants are involved in setup average time is slightly increased. After computation, an average increment of adding one participant to setup is 1.3850583 milliseconds. Setup part of FROST signature is the most complicated part with high consumption of device performance. On the other hand, the signing part is much faster in every scheme then the setup part due to lower mathematical difficulty as it can be noticed on following table 2.3. The average increment per participant is calculated with the result of 0.351 milliseconds.

Itr.\(t,n)	(2,3) [ms]	(3,5) [ms]	(4,6) [ms]
1	6.874	2.452	6.147
2	3.385	14.967	17.827
3	6.232	13.609	16.278
4	4.63	13.115	17.585
5	6.865	4.349	11.094
6	7.668	4.77	3.56
7	7.287	9.466	9.513
8	3.23	12.291	3.851
9	18.552	12.412	10.508
10	3.704	13.466	20.307
11	2.55	12.095	12.803
Avg.	**6.45245**	**10.272**	**11.77027**

Tab. 2.2: Frost Setup Benchmark

Itr.\(t,n)	(2,3) [ms]	(3,5) [ms]	(4,6) [ms]
1	0.561	0.398	1.555
2	0.794	1.044	0.726
3	0.949	1.093	0.524
4	0.355	0.585	2.804
5	0.591	0.569	0.775
6	0.565	0.816	1.288
7	0.587	1.557	1.856
8	0.64	1.583	2.458
9	0.652	0.595	0.565
10	0.871	1.658	1.011
11	0.392	1.62	2.109
Avg.	**0.723**	**1.047**	**1.425**

Tab. 2.3: Frost Signing Benchmark

2.4 Implementation of TSPB

The TSPB signature is implemented in C, since it is low level program language that is resulting high performance and possibility for IoT device implementation. Implementation consists of following .c files: *main.c*, *setup.c*, *signing.c*, *globals.c*, *support_functions.c* and *macros.c*. .c files are then linked with header files located *../headers* and with headers of OpenSSL library and JSON library that have to be downloaded to OS. Header files in *../header* are following: *setup.h*, *signing.h*, *support_functions.h* and *globals.h*. Lastly, the project contains a folder $precomputed_values$. In the folder can be found 2 .json files $precomputation_message$ and $precomputation_noise$.The whole project is built by $Makefile$.

The implementation partially is followed up master thesis [1] as a code of the thesis was provided with goal to use an setup part of Secret Sharing Authentication Key Agreement and optimize it for TSPB.

The project is meant to be programmed as library with API that is run by *main.c* which tests whole library with a result of released signature and its verification. Structure of API and communication between participant is very similar to the FROST implementation as singing part is identical with the communication during the signing process. Therefore, signing communication is pictured in figure 2.3. Moreover, verification of the final signature is done by the same algorithm 4 from subsection 1.3.1.

2.4.1 Results

After investing significant time and effort, I have managed to make progress with the C code obtained from the master thesis [1]. Although the code lacked clear comments and was challenging to comprehend, I successfully executed it, albeit only partially, by identifying and rectifying errors in obtain in public key. As verification was always aborted by public key from function *_get_pk_c*() even after point was changed from point to uncompressed bignum form, the changes were made in this part. New public key is computed as sum of all verification shares that are computed as $PK_i = G * s_i$, where s_i is the session key of participant P_i. Moreover, global variables were moved to *macros.c* as provided code was not able to build due to errors of multiple definitions. It is worth noting that the original code exhibited several undesirable characteristics, such as excessive use of global variables and the presence of goto statements, which are generally discouraged in C programming. Moreover, the code is rigid in terms of adding participants the setup is using only for loop structure for storage and communication between fictive participants.

Despite these challenges, the implementation is partially working in scheme $(3, 3)$

as the final signature is successfully verified. However, the implementation is aborted as threshold signature as computation of SSS has to have unresolved error or is just incompatible with overall API as the setup using random generated number instead of ID of participant in SSS.

To enhance the implementation of TSPB, it is important to consider the following recommendations:

- To improve code readability, add clear comments throughout the codebase. This will make the code easier to understand and facilitate future maintenance and collaboration.
- Reduce the reliance on global variables by encapsulating data within appropriate data structures. Instead of using global variables, pass data as parameters to functions as needed. This will make the code more modular and organized.
- Refactor the control flow by replacing goto statements with structured control flow mechanisms, such as loops and conditional statements. This will improve the overall code structure and make it easier to maintain.
- Modify the participant setup to allow for dynamic addition or removal of participants. Currently, the code relies on a fixed for loop structure, which limits scalability and adaptability. By making the setup more flexible, the codebase can accommodate varying numbers of participants more effectively.
- Investigate and resolve any errors or incompatibilities in the computation of the threshold signature using SSS. Ensure that the proper participant IDs are used instead of randomly generated numbers. This will result in a more accurate and reliable implementation of the threshold signature functionality.

By implementing these recommendations and continuously refining the codebase, the overall implementation of TSPB can be significantly improved. These improvements will make the system more robust and efficient, enhancing its reliability and usability.

2.5 Working Environment

The implementations were performed on a device with the following parameters: Processor Intel(R) Core(TM) $i5-8250U\ CPU\ 1.60GHz$, $4Core(s)$, $8GB\ RAM$, Windows 11 Home x64 based Operate (Host) System. Virtual machine system: $Ubuntu\ 22.04.2\ LTS$ with kernel version $5.19.0-38-generic$. The Virtual Machine was set to 4 GB RAM and 4 CPU processors. Following commands are required for installing Openssl, cJSON libraries into system and other dependencies for building the projects:

```
/*Install OpenSSL into system*/
sudo apt-get install openssl
/*Install the OpenSSL development headers*/
sudo apt-get install libssl-dev
/*Install make for building the project*/
sudo apt install make
/*Install gcc compiler*/
sudo apt install gcc
/*Install cJson into system*/
sudo apt install libcjson-dev
```

Listing 2.4: Installing Project Dependentcies

3 Practical Background

3.1 Programming Language

The implementation is written and programmed in C programming language. Since its the first introduction to public, it has developed into one of the most popular and important programming languages, acting as the basis for several other programming languages and operating systems.

The advantages of C include its effectiveness, adaptability, and intimate connection to the underlying hardware. Because it is a low-level language with direct access to memory and system resources, it is appropriate for embedded systems and systems programming. It also provides high-level structures that enable organized and modular programming at the same time.

For the implementation was chosen mainly because of its portability and efficiency: C programs can be compiled to run on a wide range of platforms and architectures. The language itself is designed to be highly portable, allowing developers to write code that can be easily ported and executed on different systems. Moreover, as C is low-level language, it provides low-level control over memory and hardware resources which results in code that is highly optimized for performance. Thus, execution speed and memory usage is exceptional.

The **C/C++ Extension Pack** (Version 1.3.0) created by Microsoft was used in conjunction with the Microsoft Visual Studio Code editor (Version 1.78.2) for the development [45]. The list of extensions included in this package and other used are following:

- C/C++ by Microsoft (Version 1.15.4) for IntelliSense, debugging and code browsing,
- C/C++ Themes by Microsoft (Version 2.0.0) for User Interface (UI) themes,
- CMake by twxs (Version 0.0.17) for CMake language support,
- CMake Tools by Microsoft (Version 1.14.31) for extended CMake support in the VS Code,
- GitHub Pull Requests and Issues (version 0.64.0) For editing and managing pull requests and issues on the GitHub platform,
- Clang-Format by Xaver Hellauer to format C/C++ code (version 1.9.0),
- PlantUML by Jebbs (version 2.17.5) to create sequence diagrams.

3.2 Libraries

3.2.1 OpenSSL Library

OpenSSL is an open-source project [40] and is maintained by a team of volunteer developers. It has a long history and has been widely adopted by the industry as a standard cryptographic library. Furthermore, OpenSSL has a well-established process for managing vulnerabilities, including coordinated disclosure, CVE assignment, and regular security releases. Its quick response to security vulnerabilities and patching has established trust with developers for being a reliable and secure software library.

The software library OpenSSL is frequently used to give applications access to cryptographic utilities and functionalities. It is a popular choice for developers that need to include cryptography in their applications because of its robust security features and wide variety of functionality.

AES, RSA, and SHA are only a few of the many cryptographic algorithms that are supported by OpenSSL. These algorithms, which are among the strongest currently in use, are used to encrypt data, produce digital signatures, and validate the legitimacy of certificates. Strong random number generators, which are necessary for many cryptographic operations, are among the security-enhancing features included in OpenSSL [40].

The library is written in the C programming language, and is available for various operating systems, including Linux, Unix, macOS, and Windows [40]. It provides a comprehensive set of APIs for developers to incorporate cryptographic functions into their applications.

All things considered, OpenSSL is a strong and well-liked software library that offers developers useful cryptographic functionalities. Its vast functionality and robust security features make it a popular option for applications that need cryptography, and its track record for promptly patching security flaws has made it a reliable option for security-conscious apps. Therefore, the implementation of FROST signature is based on this library. Used version of OpenSSL library is following: OpenSSL 3.0.7 1; Nov 2022.

3.2.2 JSON Library

The cJSON library[46] is a popular JavaScript Object Notation (JSON) library specifically designed for C programming. It provides a lightweight and efficient solution for parsing, generating, and manipulating JSON data within C code.

The cJSON library is known for its simplicity and ease of use, making it a popular choice among C developers for working with JSON data. It is distributed as a single header file and source file, allowing for easy integration into existing projects. cJSON is designed to be lightweight and efficient, with a small footprint and minimal dependencies. It aims to provide fast JSON parsing and generation capabilities, making it suitable for resource-constrained environments or performance-critical applications. JSON data may be parsed using the cJSON library to create a hierarchical structure that is simple to explore and retrieve. It offers tools for extracting values, navigating the JSON hierarchy, and working with different data kinds including objects, arrays, characters, integers, and booleans.

3.3 Source Code Dictionary Tree

For better navigation and organization directory trees of the implementations with brief information of each file and folder are provided in this section.

```
../frost ................................................ root folder
  headers ................................. folder with headers of library
    globals.h
    setup.h ..................... linking and defining objects participant etc.
    signing.h .................... linking and defining objects aggregator etc.
  src ........................................... folder with source files
    globals.c ........................... initialization of EC and DRBG
    macros.c ..................... source file for defining of global variables
    main.c ...................................... API for testing library
    setup.c ........................... source file of setup computations
    signing.c ......................... source file of signing computations
  build ................................... folder where project is built
  Makefile ................................ file for project compilation
```

```
../TSPPB ................................................ root folder
  headers ................................. folder with headers of library
    globals.h
    setup.h ..................... linking and defining objects participant etc.
    signing.h .................... linking and defining objects aggregator etc.
    support_functions.h
  src ........................................... folder with source files
    globals.c ........................... initialization of EC and DRBG
    macros.c ..................... source file for defining of global variables
    main.c ...................................... API for testing library
    setup.c ........................... source file of setup computations
    signing.c ......................... source file of signing computations
    support_functions.c ............. source file of Pailler’s computations
  build ................................... folder where project is built
  precomputed_values ............................ folder with .json files
    precomputation_message.json
    precomputation_noise.json
  Makefile ................................ file for project compilation
```

Conclusion

A increasing need for safe and effective cryptographic protocols has been generated by the rise of the Internet of Things (IoT), particularly in the context of portable devices with constrained resources. This thesis looked at the usage of multisignatures for IoT, with a particular emphasis on the implementation of the FROST signature scheme based on elliptic curves (more particularly, secp256r1) in the C programming language with the OpenSSL library.

Our research has demonstrated that the FROST signature scheme, which has various benefits over other current schemes, is a potential option for lightweight multisignatures in IoT applications. The first feature of the concept is distributed key generation, which enables the creation of public and private keys without the need for a single, trusted authority. Second, it uses elliptic curve encryption instead of more conventional RSA-based techniques, which provides high levels of security while using less resources. Additionally, the FROST signature technique outperforms other comparable systems in terms of efficiency, notably with regard to signature size and verification speed.

Moreover, the work includes the implementation of Threshold Signature for Privacy-preserving Blockchain. This implementation is partially working in scheme $(3, 3)$. Further work is essential namely in setup part as the code is rigid with possible inconsistencies or incompatibilities with overall library and API.

We have also investigated different elements of elliptic curve cryptography throughout our implementation, including the mathematical foundations of elliptic curves and their use in cryptographic applications. We have also given security in IoT systems some thought, especially the necessity for compact solutions that can effectively fend off intrusions.

In conclusion, the use of lightweight multisignatures for IoT applications has been shown to be feasible and potentially useful through the implementation of the FROST signature scheme based on elliptic curves in the C programming language using the OpenSSL library. Our research has emphasized the need of efficiency and security in these systems, and we think the FROST signature scheme offers a potential way to satisfy these needs.

Bibliography

[1] Pavla Ryšavá. Secret sharing authentication key agreement. Master's thesis, VUT Brno, 2022. URL: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=241101.

[2] Chelsea Komlo and Ian Goldberg. Frost: flexible round-optimized schnorr threshold signatures. In *International Conference on Selected Areas in Cryptography*, pages 34–65. Springer, 2020.

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

[4] CoinEx Chain Team. Acceleration of ecdsa verification with endomorphism mapping of secp256k1. *Medium*, January 2, 2020. URL: https://coinexsmartchain.medium.com/acceleration-of-ecdsa-verification-with-endomorphism-mapping-of-secp256k1-12

[5] Azine Houria, Bencherif Mohamed Abdelkader, and Guessoum Abderezzak. A comparison between the secp256r1 and the koblitz secp256k1 bitcoin curves. *Indonesian Journal of Electrical Engineering and Computer Science*, 13(3):910–918, 2019.

[6] Jonathan Katz. Digital signatures: Background and definitions. In *Digital Signatures*, pages 3–33. Springer, 2010.

[7] Dominic Chalmers, Christian Fisch, Russell Matthews, William Quinn, and Jan Recker. Beyond the bubble: Will nfts and digital proof of ownership empower creative industry entrepreneurs? *Journal of Business Venturing Insights*, 17:e00309, 2022.

[8] Bitcoin Core. Technology roadmap-schnorr signatures and signature aggregation. *URL: https://bitcoincore. org/en/2017/03/23/schnorrsignature-aggregation/(visited on 06/07/2020)*, 2017.

[9] Sara Ricci, Petr Dzurenda, Raúl Casanova-Marqués, and Petr Cika. Threshold signature for privacy-preserving blockchain. In *Business Process Management: Blockchain, Robotic Process Automation, and Central and Eastern Europe Forum: BPM 2022 Blockchain, RPA, and CEE Forum, Münster, Germany, September 11–16, 2022, Proceedings*, pages 100–115. Springer, 2022.

[10] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[11] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[12] Ján Jančár. Security considerations for elliptic curve domain parameters selection.

[13] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[14] National Institute of Standards and Technology. Digital signature standard (dss), 2013-07-19 2013. doi:https://doi.org/10.6028/NIST.FIPS.186-4.

[15] Daniel RL Brown. Sec 2: Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography*, 2010.

[16] 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.

[17] Shailee Adinolfi. Public comments received on draft fips 186-5: Digital signature standards (dss). January 27, 2020. URL: https://csrc.nist.gov/CSRC/media/Publications/fips/186/5/draft/documents/fips-186-5-draft-comments-received.pdf.

[18] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, 1977.

[19] AF Webster and Stafford E Tavares. On the design of s-boxes. In *Conference on the theory and application of cryptographic techniques*, pages 523–534. Springer, 1985.

[20] Bart Preneel. *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit te Leuven Leuven, 1993.

[21] Morris J Dworkin et al. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

[22] FIPS Pub. Secure hash standard (shs). *Fips pub*, 180(4), 2012.

[23] Quynh Dang. Secure hash standard, 2015-08-04 2015. doi:https://doi.org/10.6028/NIST.FIPS.180-4.

[24] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[26] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[27] Bc Antonín Dufka. *Schnorr Signatures with Application to Bitcoin*. PhD thesis, Master's thesis, Masaryk University Faculty of Informatics, Czech Republic, 2020.

[28] British Standards Institution. Elliptic curve cryptography. *Technical Guideline BSI TR-03111*, 2018. URL: `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_V-2-1_pdf.pdf?__blob=publicationFile&v=1`.

[29] ISO/IEC 4888-3:2018. *IT Security techniques — Digital signatures with appendix — Part 3: Discrete logarithm based mechanisms*. International Organization for Standardization, 2018.

[30] Tim Ruffing Pieter Wuille, Jonas Nick. Schnorr signatures for secp256k1, last update on aug 23, 2022. last commit is 3998dbbc8a3ab3bfabb1b2e90a4840ad93a84adb. URL: `https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki`.

[31] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979. `doi:10.1145/359168.359176`.

[32] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10*, pages 522–526. Springer, 1991.

[33] JL Lagrange. Leçon cinquième: sur l'usage des courbes dans la solution des problèmes. *Séances des Écoles Normales recueillies par les sténographes et revues par les professeurs, Reynier, Paris*, 1795.

[34] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. [cited 10-12-2021].

[35] Christine Jost et al. Encryption performance improvements of the paillier cryptosystem. *Cryptology ePrint Archive*, Report 2015/864, 2015. [cited 17-05-2022]. URL: `https://ia.cr/2015/864`.

[36] Chelsea Komlo. On security assumptions underpinning recent schnorr threshold schemes. *Ethereum Foundation*, 2022. URL: `https://crypto.ethereum.org/blog/schnorr-threshold-blogpost#user-content-fn-2`.

[37] Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: simple two-round schnorr multi-signatures. In *Annual International Cryptology Conference*, pages 189–221. Springer, 2021.

[38] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399, 2006.

[39] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1084–1101. IEEE, 2019.

[40] OpenSSL. *OpenSSL Project*, 2021. Accessed: May 16, 2023.

[41] OpenSSL. *OpenSSL `RAND_DRBG`*. OpenSSL, 2021. Accessed: May 16, 2023.

[42] National Institute of Standards and Technology. *NIST Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. National Institute of Standards and Technology, 2012. Accessed: May 16, 2023.

[43] OpenSSL. *OpenSSL `OPENSSL_malloc`*. OpenSSL, 2021. Accessed: May 16, 2023.

[44] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[45] Microsoft. C/C++ Extension Pack - Visual Studio Marketplace. Visual Studio Marketplace. URL: `https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools-extension-pack`.

[46] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

Symbols and abbreviations

NFT	Non-Fungible Token
FROST	Flexible Round-Optimized Schnorr Threshold signature
IoT	Internet of Thing
RSA	Rivest–Shamir–Adleman
EC	Elliptic Curves
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
ECDLP	Elliptic Curve Discrete Logarithm Problem
NIST	National Institute of Standards and Technology
SECG	Standards for Efficient Cryptography Group
NSA	National Security Agency
SHA-256	Secure Hash Algorithm 256-bit
PKI	Public Key Infrastructure
CA	Certificate Authority
DSS	Digital Signature Standards
DSA	Digital Signature Algorithm
SHS	Secure Hash Standard
ROM	Random Oracle Model
DPL	Discrete Logarithm Problem
BIP	Bitcoin Improvement Proposal
SSS	Shamir's Secret Sharing
DKG	Distributed Key Generation
Mu-Sig	Multi-Signature
OMDL	One-More Discrete Logarithm

PROM Programmable Random Oracle Model

DRBG Deterministic Random Bit Generator

PRNG Pseudo-Random Number Generators

MAC Message Authentication Code

TSPB Threshold Signature for Privacy-preserving Blockchain

TSPB Threshold Signature for Privacy-preserving Blockchain

API Application Programming Interface

JSON JavaScript Object Notation