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

GROUP SIGNATURE BASED ON SECURE MULTI-PARTY COMPUTATION

SKUPINOVÝ PODPIS ZALOŽENÝ NA BEZPEČNÉM VÝPOČTU VÍCE STRAN

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR — Bc. Kristián Klasovitý

AUTOR PRÁCE

SUPERVISOR — M.Sc. Sara Ricci, Ph.D.

VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Kristián Klasovitý ***ID:*** 196068

Year of study: 2 ***Academic year:*** 2022/23

TITLE OF THESIS:

Group signature based on Secure Multi-party Computation

INSTRUCTION:

The work is focused on the implementation of a group signature based on secure multi-party computation and Weak-Boneh Boyen signature. The thesis aims to implement and analyze (through experimental results) a new group signature generated by the merging of the two-party computation scheme proposed in [1] and the Weak-Boneh Boyen signature [2]. The signature implementation is expected to be run on different devices.

RECOMMENDED LITERATURE:

[1] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham, ``Randomizable Proofs and Delegatable Anonymous Credentials", Advances in Cryptology - CRYPTO 2009, vol.5677, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 108-125.

[2] Boneh D, Boyen X, Shacham H. Short group signatures. In Annual international cryptology conference 2004 Aug 15 (pp. 41-55). Springer, Berlin, Heidelberg.

Date of project specification: 6.2.2023 ***Deadline for submission:*** 19.5.2023

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

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

WARNING:

The author of the Master's Thesis claims that by creating this thesis he/she did not infringe the rights of third persons and the personal and/or property rights of third persons were not subjected to derogatory treatment. The author is fully aware of the legal consequences of an infringement of provisions as per Section 11 and following of Act No 121/2000 Coll. on copyright and rights related to copyright and on amendments to some other laws (the Copyright Act) in the wording of subsequent directives including the possible criminal consequences as resulting from provisions of Part 2, Chapter VI, Article 4 of Criminal Code 40/2009 Coll.

ABSTRACT

This thesis aims at implementing a group signature scheme that uses two-party computation to jointly compute a signing value used in the signature. In this way, the user's secret key is hidden from the manager and it cannot be used to impersonate the user. The signature also supports revocation and opening algorithms. Moreover, a blind issuance attribute-based credential is also presented, where the credential issued by the issuer remains private to the user. Both schemes were run on different devices and the performances were benchmarked. At last, the group signature was used to implement an application allowing one to sign a document on behalf of a group. The implementation is run on multiple devices that use NFC to communicate.

KEYWORDS

Android, Elliptic curves, Group signature, Homomorphic encryption, NFC, Non-interactive zero-knowledge proof of knowledge, Secure two-party computation, Schnorr protocol

ABSTRAKT

Práce se věnuje implementaci skupinového podpisu, který využívá společný výpočet dvou stran k vypočítání tajné hodnoty využité k podpisu. Díky tomu zůstává soukromý klíč člena skupiny skrytý před manažerem skupiny, a nemůže být manažerem zneužit. Podpisové schéma podporuje revokaci a otevírání podpisů manažerem. Také byl představen způsob slepého vydávání atributového pověření, kde je pověření skryto před vydavatelem. Obě schémata byla spuštěna na více zařízeních a byl změřen čas jejich vykonání. Schéma pro skupinové podpisy bylo využito pro vytvoření aplikace, pomocí které je možné podepisovat dokumenty jménem skupiny. Implementace je spuštěna na více zařízeních, která komunikují pomocí NFC.

KLÍČOVÁ SLOVA

Android, eliptické křivky, skupinový podpis, homomorfní šifrování, NFC, neinteraktivní důkaz s nulovou znalostní, bezpečný společný výpočet dvou stran, Schnorrův protokol

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

ROZŠÍŘENÝ ABSTRAKT

Tato práce se zaměřuje na implementaci skupinového podpisu s použitím bezpečného výpočtu dvou stran. Skupinový podpis je podpis, kdy členové určité skupiny, kterou spravuje manažer, mohou vytvářet podpisy jménem celé skupiny. Ověřovatel při ověření takových podpisů nezjistí žádné informace o podepisujícím, ale může pouze zjistit, jaká skupina podpis vytvořila a zda je platný. Jedním z problémů těchto podpisů je vydávání tajných hodnot použitých v podpisu jednotlivých členů skupiny. Jelikož je tajná hodnota vypočítána ze soukromého klíče manažera skupiny a člena skupiny, dochází tak k prozrazení jak soukromého klíče, tak této vypočítané hodnoty manažerovi. Toto je možné vyřešit přidáním společného výpočtu obou stran za použití homomorfního šifrování. Díky tomu zůstane soukromý klíč manažera skryt před uživatelem a zároveň soukromý klíč uživatele i výsledná tajná hodnota používaná v podpisu zůstane skryta před manažerem.

Schéma implementované v této práci využívá skupinový podpis založený na weak Boneh-Boyen podpisu a na společném výpočtu manažera a člena skupina za využití neinteraktivního důkazu s nulovou znalostí (angl. *Non-Interactive Zero-Knowledge Proof of Knowledge*, zkráceně NIZKPK). Pomocí zmíněného schématu je vytvořen systém pro podepisování dokumentů. Tento výpočet více stran je poté použit i pro implementaci slepého vydávání atributů ve schématu klíčově ověřitelných anonymních pověření (angl. *Keyed-Verification Anonymous Credentials*, zkráceně KVAC).

V první části se práce věnuje představení kryptografických primitiv, které jsou v protokolu použity, jako jsou kryptografie eliptických křivek a operace bilineárního párování na křivkách, podpis Weak Boneh-Boyen, Schnorrův protokol, teorie skupinových podpisů a anonymních atributových pověření. Dále je zde popsána technologie NFC (*Near-Field Communication*) i programovací jazyky a knihovny zvažované pro implementaci.

V druhé části jsou podrobně popsány implementované kryptografické protokoly. Prvním je již zmíněný NIZKPK protokol, který byl doplněn o výpočet důkazů znalosti, které dosud nebyly nikde implementovány. Jedná se tedy o první kompletní implementaci tohoto NIZKPK protokolu. Tyto důkazy jsou v protokolu použity k dokázání znalosti jednotlivých hodnot, se kterými strany, ve výpočtu pomocí homomorfního šifrování, počítají. Dále je zde popsáno schéma samotného podpisu. Je také ukázáno, jak lze dokázat, že je schéma kompletní a tedy, že pro platný podpis bude verifikace úspěšná. Dále se práce zabývá tím, jak lze ve zmíněném schématu implementovat i revokaci a otevírání podpisů. Revokací uživatele je míněno odstranění uživatele ze skupiny tak, aby jeho podpis již nebyl platný. Toho se dá docílit zveřejněním blacklistu takto vyloučených uživatelů. Tento seznam zveřejňuje právě manažer skupiny, který má také schopnost otevřít podpis. Otevřením podpisu

se zde rozumí odhalení identity podepisujícího. Nakonec jsou v této části popsány změny provedené v KVAC protokolu pro implementaci slepého vydávání pověření, kdy je pověření skryto během vydávacího protokolu před vydavatelem.

Výstupem praktické části je aplikace, která umožňuje podepisování dokumentů pomocí skupinového podpisu. Tato implementace obsahuje aplikaci pro PC, která je rozdělena na tři menší aplikace. První je manažerská aplikace, která umožňuje spravování skupiny, včetně přidávání členů za pomoci NIZKPK protokolu. Druhou aplikací je aplikace člena skupiny, která umožňuje vybrání PDF souborů k podpisu a uložení podpisu do jejich metadat. Poslední je aplikace ověřovatele, která slouží k ověření podpisů vytvořených pomocí této aplikace. Vedle PC aplikace byla také vytvořena aplikace mobilní telefony s operačním systémem Android. Jedná se o aplikaci člena skupiny, kterou uživatel používá pro generování jednotlivých podpisů a pro výpočet NIZKPK protokolu během přidávání uživatele do skupiny.

Při reálné simulaci systému je tedy manažer zastoupen PC aplikací, která pro přidání uživatele do skupiny komunikuje skrze NFC s aplikací v mobilním telefonu, ta simuluje uživatele. Tato dvě zařízení spolu spočítají NIZKPK protokol a uživatel obdrží svou tajnou hodnotu, kterou poté může použít pro podepisování souborů. Při podpisu si uživatel otevře PC aplikaci pro podpis, která poté mobilnímu telefonu skrze NFC posílá hash souboru k podepsání. Podpis je poté spočítán v telefonu a odeslán zpět do PC, kde je uložen.

Pro implementaci byl zvolen jazyk Java, a to z důvodu jeho přenositelnosti a využití v Androidu, pro který byla mobilní aplikace zamýšlena. Aby byl protokol co nejrychlejší, byly na začátku provedeny rychlostní testy několika knihoven. Jednalo se o knihovny Javy a jazyka C. I když byla zvolena Java jako hlavní jazyk, nativní rozhraní Javy (angl. *Java Native Interface*, zkráceně JNI), umožňuje používání funkcí jazyka C v Javě. Srovnávané knihovny pro bilineární párování na eliptických křivkách byly MCL (jazyk C) a AMCL (Java). Ze srovnání vyšla mnohonásobně lépe knihovna MCL, a proto byla využita ve finální implementaci. Pro modulární aritmetiku byly porovnávány knihovny GMP (jazyk C) a BigInteger (Java). V tomto případě na mobilním zařízení a na virtuálním PC s operačním systémem Ubuntu Linux byla rychlejší knihovna GMP, avšak na PC s operačním systémem Windows byla rychlejší knihovna BigInteger. Na obou zařízeních byla přidána možnost manuálně přepínat mezi možnostmi použití těchto knihoven ve výpočtech, a to především pro podporu jiných Unixových operačních systémů, kde může být GMP rychlejší. V základním nastavení byla ale na mobilním telefonu použita knihovna GMP a na PC BigInteger.

Dále práce popisuje postup při implementaci systému. První součástí je kryptografické jádro aplikace. Bylo implementováno tak, aby nebylo závislé na jiných částech aplikace a případně se dalo přesunout a použít v dalších implementacích.

Toto jádro obsahuje jednotlivé kroky protokolů. V této části je také naznačeno, jak lze vybrané knihovny jazyka C použít i v programu Javy.

Další částí během tvorby systému byla implementace komunikačního rozhraní mezi PC a mobilním telefonem. K tomu bylo zvoleno NFC. V práci je podrobněji popsán způsob, jak lze toto rozhraní používat. Skládá se z částí implementace terminálu, který používá čtečku karet, a implementace emulace čipových karet (angl. *Host Card Emulation*, zkráceně HCE) na mobilním telefonu s operačním systémem Android.

Následně jsou popsány kroky pro vytvoření výsledných aplikací, které budou uživatelé používat. Postup sestává z řešení ukládání souborů v obou zařízeních, z tvorby grafického uživatelského rozhraní a způsobu jeho propojení se zbývajícími částmi aplikace.

Celý systém pro skupinové podpisy byl poté otestován a byly provedeny rychlostní testy všech důležitých částí protokolu a systému. Hlavním cílem bylo zjistit, jak se liší rychlost protokolu na různých zařízeních a zda je protokol spočitatelný na těchto zařízeních v rozumném čase. První testovanou částí byl NIZKPK protokol použitý pro společný výpočet manažera a uživatele, připojujícího se do skupiny. Zde bylo zjištěno, že nejrychlejší konfigurací byl počítač s operačním systémem Ubuntu Linux používající knihovnu GMP v kombinaci s mobilním telefonem. Průměrný čas protokolu dosahoval v tomto případě 9 sekund. Jelikož byl protokol rozdělen na dvě komunikační fáze a část byla počítána bez aktivního spojení, telefon musel být nejprve na čtečce držen v průměru 1,2 sekundy a poté 2 sekundy. Druhou stále použitelnou kombinace byl počítač s operačním systémem Windows využívající knihovnu BigInteger spolu s telefonem. Naopak zařízení jako Raspberry Pi 4 B či chytré hodinky se ukázala jako nevhodná v tomto protokolu, a to z důvodu nedostatečné rychlosti. Při použití Raspberry Pi bylo nutné udržet aktivní NFC spojení i 10 sekund. I když nebyl NIZKPK protokol příliš rychlý, protokol se musí provést pouze jednou pro každého uživatele, takže se nejedná o velkou překážku.

Rychlost podepisování pomocí mobilního zařízení byla také měřena. Samotný podpis byl zde velmi rychlý, v průměru trval 7 ms na telefonu a 44 ms na chytrých hodinkách. Nejvíce času při podepisování zabrala komunikace přes NFC. Celkově podepisování, které se skládalo ze zaslání hashe od PC do telefonu, spočítání podpisu a jeho následné odeslání z telefonu do PC, trvalo průměrně 125 ms. Algoritmus používaný pro verifikaci podpisu byl také velmi rychlý, sám o sobě trval na nejrychlejším testovaném zařízení přibližně 8 ms. Čas ověření podpisu také závisel na počtu revokovaných uživatelů, kdy na stejném systému vzrostl o 3,7 ms pro každého revokovaného uživatele. Celkově ověření, i s načtením souboru a jeho hashováním trvalo na nejrychlejším z testovaných zařízení průměrně 150 ms.

Poslední část praktické implementace byla věnována použití NIZKPK protokolu

pro slepé vydávání atributů pro KVAC schéma. Do existující implementace KVAC protokolu byla přidána existující implementace NIZKPK protokolu. Ta ovšem neobsahovala všechny potřebné části protokolu, a musela být doplněna. Stejně tak musela být pozměněna původní implementace KVAC protokolu, aby bylo možné v něm použít slepé vydávání. Rychlost této implementace byla měřena na více zařízeních, ale bez komunikačního rozhraní. Takto modifikovaný algoritmus byl poté porovnán s původním protokolem. Hlavním cílem srovnání bylo zjistit, jak slepé vydávání pověření ovlivní rychlost prokazovacího (Show) a ověřovacího (Verify) algoritmu. Tyto algoritmy byly pouze mírně zpomaleny o maximálně 10 %. Největší nevýhodou byl ale pomalý NIZKPK protokol použitý při vydávání pověření, avšak podobně jako u skupinového podpisu je nutné jej spočítat pouze jednou pro každého uživatele.

KLASOVITÝ, Kristián. *Group signature based on Secure Multi-party Computation*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2023, 98 p. Master's Thesis. Advised by M.Sc. Sara Ricci, Ph.D.

Author's Declaration

Author:	Bc. Kristián Klasovitý
Author's ID:	196068
Paper type:	Master's Thesis
Academic year:	2022/23
Topic:	Group signature based on Secure Multi-party Computation

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, 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 thank the advisor of my thesis, M.Sc. Sara Ricci, Ph.D. for her valuable comments, patience and suggestions for improvements.

Contents

Introduction **16**

1 Cryptographic primitives, protocols, implementation background **17**

1.1 Basics of asymmetric cryptography 17
1.2 Elliptic curve cryptography 17
1.2.1 Operations on elliptic curve 18
1.2.2 Bilinear pairings on elliptic curves 20
1.3 Weak Boneh-Boyen signature 21
1.4 Schnorr's protocol 22
1.5 Homomorphic encryption 23
1.5.1 Paillier cryptosystem 24
1.6 Group Signatures 25
1.7 Attribute-based credentials 26
1.7.1 Keyed-Verification Anonymous Credentials 26
1.8 Implementation background 28
1.8.1 Java 28
1.8.2 C language 29
1.8.3 Android 30
1.8.4 Near-field communication 31

2 Group signature with two-party computation and blind issuance for attribute-based credentials **33**

2.1 NIZKPK protcol 33
2.1.1 Setup phase 34
2.1.2 Two-party signature 34
2.2 Group signature 37
2.2.1 Revocation and opening of the signature 38
2.3 Using the NIZKPK in KVAC 39
2.3.1 Modifying the issue algorithm 39
2.3.2 Modifying the Show and ShowVerify algorithms 40

3 Practical implementation **42**

3.1 Choice of a platform and libraries 42
3.1.1 Comparison of Java and C libraries on an Android device . . . 43
3.1.2 Comparasion of Java and C libraries on a desktop PC 46
3.2 Implementing the cryptographic core 49
3.2.1 Integrating the libraries into a project 49

3.2.2 Implementation of NIZKPK 52
3.2.3 Implementation of the group signature algorithms 54
3.3 Implementing communication between the devices 56
3.3.1 Terminal on PC 56
3.3.2 Implementation of HCE on Android 58
3.4 File managing on the devices 61
3.4.1 Files in the PC application 61
3.4.2 Storing application data on an Android device 64
3.5 Building the applications with GUI 65
3.5.1 GUI and final application on PC 65
3.5.2 GUI on Android 68
3.6 Benchmarks of the implemented group signature protocol and the applications 69
3.6.1 Benchamrks of the NIZKPK protocol 70
3.6.2 Benchmarks of the group signature 75
3.7 Implementing the NIZKPK into KVAC 77
3.7.1 Comparing the modified KVAC with the original version . . . 78

Conclusion **80**

Bibliography **82**

Symbols and abbreviations **88**

A Structure of the archive with the source files **90**

B Manual for the applications **91**
B.1 PC application 91
B.1.1 Installing the PC application on Windows 91
B.1.2 Installing the PC application on Linux 91
B.2 Installation of the mobile application 93
B.3 Using the PC application with the Android application 94

List of Figures

1.1 Point addition on an elliptic curve. 19
1.2 Scheme of Schnorr's interactive protocol. 22
1.3 Structure of an APDU command and APDU response. 32
3.1 State diagram of the NIZKPK functions calls. 53
3.2 Structure of the choose AID command. 56
3.3 Checking the password for the file. 63
3.4 The main window of the application. 66
3.5 The manager's application. 67
3.6 The signer's application. 67
3.7 The verifier's application. 68
3.8 GUI of the Android phone application. 69
3.9 NIZKPK protocol with NFC communication. 71
3.10 Times of manager's pre-computation on different devices. 72
3.11 Times of the NFC transfers and manager's computation during the active protocol on different devices. 72
3.12 Times of the mobile device's computations during the NIZKPK protocol for a phone and a smartwatch. 73
3.13 Total time of the interactive part of the protocol for different combinations of devices. 74
3.14 Time consumption during the signing algorithm in the application. . 75
3.15 Time needed to check the revocation depending on the number of revoked users in the group. 76
3.16 Time it takes to compute the Show algorithm on mobile devices. . . . 79
3.17 Time it takes to compute the Verify algorithm. 79
B.1 The NFC settings on a Xiaomi phone. 94
B.2 The main window of the application. 95
B.3 The register and login windows of the PC application. 95
B.4 The manager application. 96
B.5 The mobile application during the protocol. 97
B.6 The client's signing application. 98
B.7 Result of the verification process. 98

List of Tables

3.1 Comparison of MCL and AMCL computation times (in μ s) on an Android device. 44
3.2 Comparison of BigInteger and GMP computation times for addition and multiplication on an Android device (times in μs). 45
3.3 Comparison of BigInteger and GMP computation times for modular exponentiation and prime generation on an Android device (times in μs). 45
3.4 Comparison of MCL and AMCL computation times (in μ s) on a Windows PC. 46
3.5 Comparison of BigInteger and GMP computation times for modular exponentiation and prime generation on a Windows PC (times in ms). 47
3.6 Comparison of MCL and AMCL computation times (in μ s) on Linux VM. 48
3.7 Comparison of BigInteger and GMP computation times for modular exponentiation and prime generation on Linux VM (times in ms). . . 48

Listings

3.1 Android.mk file in jniLibs 50
3.2 Passing the Android.mk to Gradle 51
3.3 ModPow function in C for Java 51
3.4 Compile command for library using GMP in Java for Linux 52
3.5 The function to initialize connection with a mobile device 57
3.6 Modification to the AndroidManifest.xml 58
3.7 Content of apduservice.xml file 59
3.8 Creation of a receiver in the MyHostApduService class 60
3.9 Saving the group signature to the metadata 63
B.1 The setup.sh script to install the program with all the dependencies . 92

Introduction

In today's world, anonymity in cryptography is more and more desirable. There are several cryptographic tools that provide different kinds of anonymity while keeping the security features of the scheme. One of these tools is group signatures [1]. These signatures allow users from some group administered by a manager to create anonymous signatures, that cannot be easily traced back to them. The verifier can check the validity of the signature and the user's affiliation with the group, but will not know who generated the signature. These signatures can be used for signing documents on behalf of a group or for creating a room access system. In the latter, users cannot be tracked in their access [2].

The problem is that in several signatures, the group manager issues the secret values used in the signature to the users. Therefore having knowledge of the user's secret key allows a malicious manager to sign on behalf of a user. This problem can be fixed by adding a multi-party computation, that will keep the user's secret value secret and be not shared with the manager.

The aim of this diploma thesis is the implementation of a group signature scheme that uses a two-party computation algorithm based on a *Non-Interactive Zero-Knowledge Proof of Knowledge* (NIZKPK) [3], that uses the Paillier cryptosystem [4] to compute a secret, only known by the user. The secret will then be used to sign a message with a group signature based on the Weak Boneh-Boyen signature [5]. It will also be shown how the two-party computation can be used to implement blind issuance into an attribute-based credential system such as *Keyed-Verification Anonymous Credentials* (KVAC) [6]. With blind issuance, the issued credential is hidden from the issuer.

The practical applicability of the group signature is shown through an application allowing document signing. A library with the crypto core of the system has been developed in a way that it is easily managed and exported. Furthermore, both group signature and attribute base credential schemes have been benchmarked on multiple devices to show how much time it takes to run it in a real environment, for the group signature this also includes communication overhead.

1 Cryptographic primitives, protocols, implementation background

This chapter describes the basics of the cryptographic primitives, systems, and protocols, that are needed in this implementation of both the group signature based on secure multi-party computation and Weak Boneh-Boyen signature and the KVAC blind issuance implementation. This chapter also introduces the implementation background i.e., the programming languages, Android *Operating System* (OS), and libraries considered for the implementation.

1.1 Basics of asymmetric cryptography

Asymmetric cryptography [7] is the main building block of most of today's systems that use cryptography. The main reason we use cryptography in the first place is, that we want to make some data unreadable for a third side during transmission (confidentiality), so we have two main operations in cryptography - encryption and decryption.

While in symmetric cryptography, there is one key for encryption and decryption, in asymmetric, there are two keys. One is called the private key and is used for decryption and signing and the other is the public key, used for encrypting and verifying a signature. There is always some kind of relationship between these two keys, the private key should never be published or sent, however, the public key can be freely shared with everyone. Having these two keys instead of one shared secret key as in symmetric cryptography solves the issue of sharing a secret key [8].

Asymmetric cryptosystems are based on NP problems. These problems are easy to solve with the knowledge of some secret values but almost impossible to solve without them. The most widely used problems are the *Discrete Logarithm Problem* (DLP) and integer factorization problems. The factoring problem can be simply described as that, it is easy to multiply two prime numbers, but it is hard to find these two numbers if you only have the product. The discrete logarithm problem relies on that, it is easy to compute the power of a number in a multiplicative group, but it is hard to find a logarithm of a number in a multiplicative group. This of course depends on the group and the size of the number.

1.2 Elliptic curve cryptography

Elliptic curve cryptography [9] is part of asymmetric cryptography and it is used to create public key cryptosystems. This part of cryptography relies on the *Elliptic*

Curve Discrete Logarithm Problem (ECDLP), meaning it is nearly impossible to calculate the discrete logarithm of a random element on an elliptic curve if we only know the generating point of the curve. The complexity of this problem is given by the size of the curve.

Cryptography considers elliptic curves with points in field $\mathbb{F}_p$, where p is prime. While systems based on modular arithmetic usually work in multiplicative groups, elliptic curve cryptography works in additive groups. In practice, an elliptic curve can be seen as a set of points that satisfies a specific equation and a point at infinity. The most used equation is the short Weierstrass form, which is displayed below in Equation 1.1.

$$y^2 = x^3 + ax + b \ (mod\ p) \tag{1.1}$$

One of the biggest advantages of elliptic curves compared to modular arithmetic is, that much smaller keys can be used to achieve the same level of security. Also, the operations on elliptic curves are usually much faster. Fortunately, every algorithm that relies on DLP in a multiplicative group can be transformed to work on elliptic curves.

In practice, a curve is chosen to be used in an instance of a protocol, and this curve is defined by a set of public parameters, that should be known to all the participating parties [10]. Parameters defining an elliptic curve $E(\mathbb{F}_p)$ are: a number p specifying the finite field $\mathbb{F}_p$, elements a, b specifying Equation 1.1, a point G stating a generator of the cyclic subgroup, a prime n being the order of G, and a cofactor h, where $h = |E(\mathbb{F}_p)|/n$, with $|E(\mathbb{F}_p)|$ being the number of points of the curve. Changing the form of the curve, i.e. its equation, has an impact on the performance and on the security features [11].

1.2.1 Operations on elliptic curve

There are two main operations that can be done on elliptic curves, the addition of two points and multiplying a point by a scalar. Both these operations are used in cryptographic implementations [12].

Point addition

There are multiple possible cases that can occur during the addition of points P and Q on an elliptic curve. Either P and Q are different points, they can also be the same point ($P = Q$), or Q can be the opposite of P.

Addition of different points $P + Q = R$ can be seen in Figure 1.1 on the left side. Geometrically such addition is done by drawing a line through the points P, Q, finding the third point of intersection with the curve ($-R$), and then finding

the point R by reflecting $-R$ in the x-axis. In the case of addition where $P = Q$ (also called doubling), a tangent to the curve at point P is drawn, and the point R is found in the same way i.e. as a reflection of the point of intersection. Doubling can be seen in Figure 1.1 on the right side. In the case of adding a point to its opposite, the line between the points will intersect the curve at infinity.

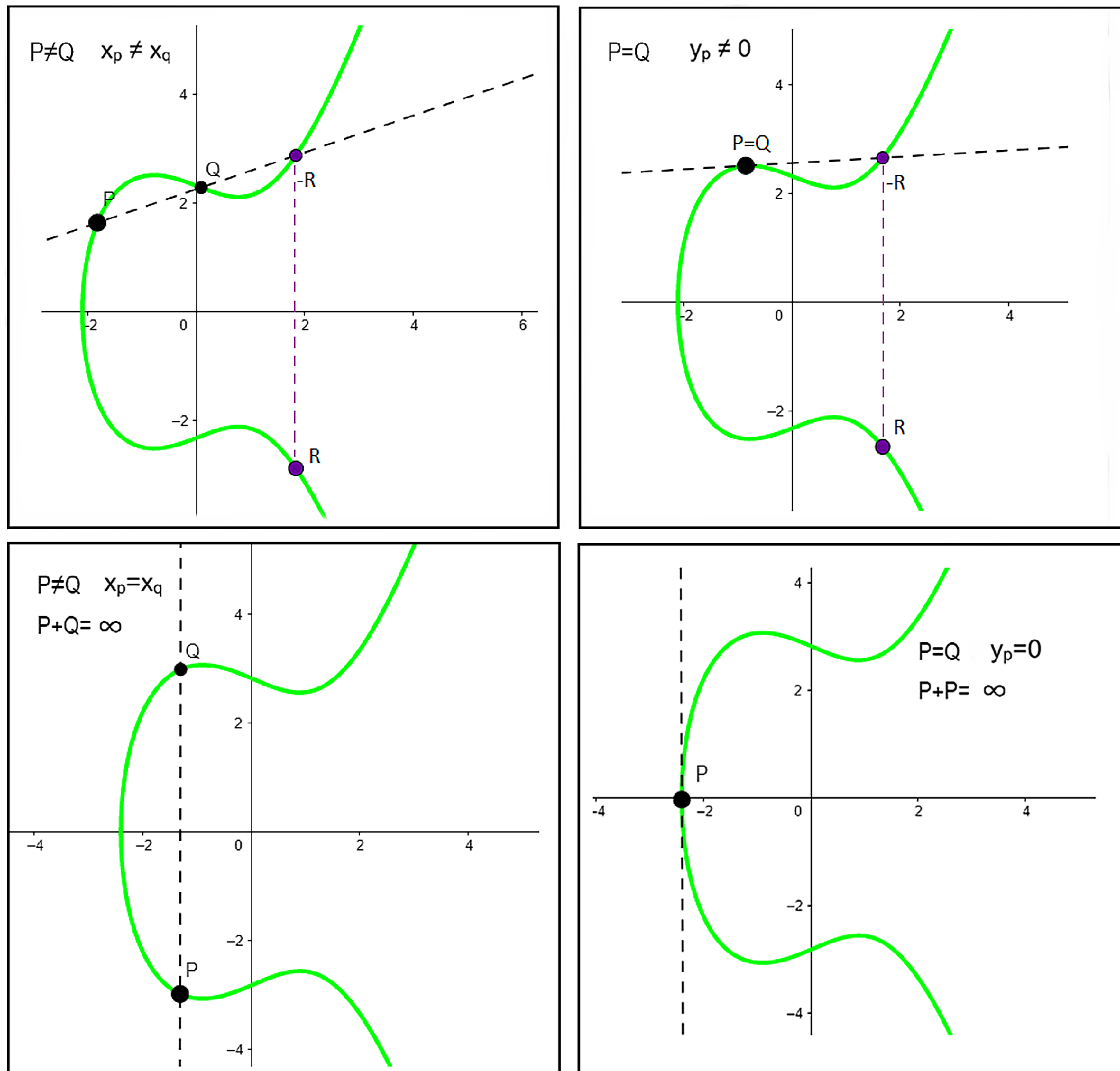


Fig. 1.1: Point addition on an elliptic curve.

In practical implementations, addition is done by calculating a gradient between the points. The calculation of $P(x_1, y_1) + Q(x_2, y_2) = R(x_3, y_3)$ can be seen in Equation 1.2 below.

$$
\begin{aligned}
s &= (y_1 - y_2)/(x_1 - x_2) \\
x_3 &= s^2 \cdot x_1 - x_2 \\
y_3 &= s \cdot (x_1 - x_2) - y_1
\end{aligned}
\tag{1.2}
$$

Multiplying a point with a scalar

Multiplying a point by scalar on an elliptic curve is defined as multiple additions of the point with itself. For example, if we would like to compute $4P$ we could compute it as $4P = P + P + P + P$. This approach would be very inefficient for big numbers. This computation can be sped up with the fast exponentiation algorithm. For instance, the multiplication of $8P$ could be done as $2P = P + P,\ 4P = 2P + 2P\ 8P = 4P + 4P$, this way only 3 additions are needed instead of 7.

In cryptography, multiplication is used to create cryptographic primitives utilizing the ECDLP. Because the multiplication is done modulo p, it is almost impossible to find k, such that $k \cdot G = P$, knowing only G and P (considering big enough k and p). This property can be used to create a pair of private and public keys, where P is the public key and k is the private key.

1.2.2 Bilinear pairings on elliptic curves

Elliptic curve bilinear pairing [13] is a bilinear map $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$, that allows us to map a pair of points (one in $\mathbb{G}_1$, one in $\mathbb{G}_2$) to a target group $\mathbb{G}_t$. Usually $\mathbb{G}_1$ is a subgroup of $E(\mathbb{F}_p)$, $\mathbb{G}_2$ is a subgroup of $E(\mathbb{F}_{p^k})$, and $\mathbb{G}_t$ is a multiplicative subgroup of $\mathbb{F}^*_{p^k}$, with k being the embedding degree of the curve. Basically, points in $\mathbb{G}_2$ are just different representations of the points in $\mathbb{G}_1$.

There are two properties that the pairing must satisfy, called bilinearity and non-degeneracy. The equations that ensure bilinearity are shown in Equation 1.3 (note that e stands for the paring function). The non-degeneracy, means that if $e(U, V) = 1$ for all V, then $U = \infty$, and if $e(U, V) = 1$ for all U then $V \in nE(\mathbf{F}_p)$ [14].

$$
\begin{aligned}
e(U_1 + U2, V) &= e(U_1, V) \cdot e(U_2, V) \\
e(aU, bV) &= e(U, V)^{ab} = e(abU, V) = e(bU, aV)
\end{aligned}
\tag{1.3}
$$

Not all curves are suitable for billinear pairings, the curves suitable for it are called pairing-friendly curves. Two widely used types of these curves are *Barreto-Naehrig* (BN) and *Barreto-Lynn-Scott* (BLS) curves. BN curves have embedding degree 12 and BLS curves either 12 or 24. There are also different kinds of pairings,

the main three are Weil, Tate, and Ate pairings. Each is a different way to compute the pairing, but Ate is the most commonly used one, as it has the best performance. There are many uses for pairings in cryptography [15], for example, identity-based encryption, signatures, attributed-based encryption, and zero-knowledge proofs.

1.3 Weak Boneh-Boyen signature

One of the schemes utilizing elliptic curve pairings is the weakly secure short signature scheme (or weak Boneh-Boyen signature) presented in [16]. This scheme is unforgeable under a weak chosen message attack.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be a bilinear group pair, where both groups have the same number of elements p, e a pairing that maps all the pairs to $\mathbb{G}_t$, and m the message (where m is an element of $\mathbb{Z}_p$). The scheme has three steps: key generation, signing, and verification.

The key generation is shown in Algorithm 1, the signature algorithm is then described with Algorithm 2 and the verification is shown in Algorithm 3. How the bilinear property can be used to prove that the equality $e(\sigma, \upsilon + m \cdot g_2) = z$ will hold for a legit signature can be seen in Equation 1.4.

Algorithm 1 Key generation

1: Consider $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$
2: Generate private key $x \in_R \mathbb{Z}_p$
3: Compute public key $\upsilon = x \cdot g_2 \in \mathbb{G}_2$
4: Compute $z = e(g_1, g_2) \in \mathbb{G}_t$
5: Store x, post public values $\mathbf{par}(g_1, g_2, z)$

Algorithm 2 Sign $(x, m, \mathbf{par})$

1: Compute signature of $m \in \mathbb{Z}_p$ as $\sigma = g_1 \cdot 1/(x + m) \in \mathbb{G}_1$
2: $1/(x + m)$ is computed as a multiplicative inverse of $(x + m)$ mod p.
3: Specify that 1/0=0, so that if $x + m \equiv 0 \ mod \ p$, we have $\sigma = 1 \in \mathbb{G}_1$.

Algorithm 3 Verify $(\sigma,\mathbf{par})$

1: Check the equality $e(\sigma, \upsilon + m \cdot g_2) = z$
2: If the equality holds or if $\sigma = 1$ and $\upsilon + m \cdot g_2 = 1$ the signature is valid.

$$
\begin{aligned}
&e(\sigma, \upsilon + m \cdot g_2) \\
&e(g_1 \cdot \frac{1}{x+m}, x \cdot g_2 + m \cdot g_2) \\
&e(g_1 \cdot \frac{1}{x+m}, g_2 \cdot (x+m)) \\
&e(g_1, g_2 \cdot \frac{x+m}{x+m}) \\
&e(g_1, g_2) = z
\end{aligned}
\tag{1.4}
$$

1.4 Schnorr's protocol

Shnorr's protocol is one of the so-called Sigma protocols [7]. Sigma protocols are based on zero-knowledge protocols. With these protocols, one (usually called a prover) can prove the knowledge of a secret value without revealing it to the verifier. This secret value is usually a private key. The output of such a protocol is just that the prover's statement is true or false, nothing else is revealed.

During Schnorr's protocol, the prover tries to prove the knowledge of the discrete logarithm x of an element $h = g^x \in E[\mathbb{Z}_q]$, where E is an elliptic curve of the system. In practice, h is the public key, and x is the private one. The protocol can be either interactive or non-interactive. In the interactive version, three messages must be exchanged between the prover and the verifier (shown in Figure 1.2). In the non-interactive proof, only one message is sent from the prover to the verifier [17].

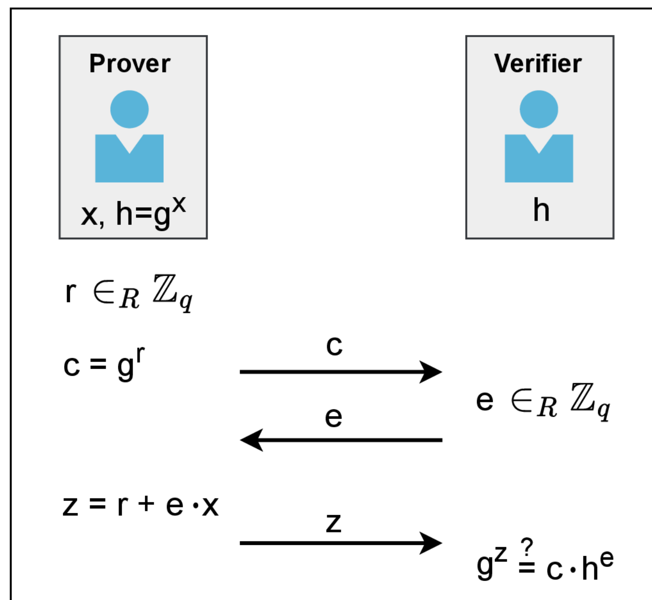


Fig. 1.2: Scheme of Schnorr's interactive protocol.

As mentioned, Schnorr's protocol can be modified to create a non-interactive zero-knowledge proof, where only one message is needed. For this, The Fiat-Shamir

heuristic technique is used. This technique allows us to convert an interactive protocol into a non-interactive one. That is done by the value of a hash function instead of the challenge generated by the verifier. Since there is no challenge this scheme could be susceptible to replay attacks, therefore a random value is usually generated by the prover and used in the hash function, these random values then might be stored on the verifier's side, so that the values can't be used again. Algorithm 4 shows how such a proof is computed, Algorithm 5 then shows the verification.

Algorithm 4 Non-interactive proof computation $(g, x, h = g^x)$

1: Generate $r \in_R \mathbb{Z}_q$
2: $u = g^r$
3: $e = H(g, h, u)$
4: $z = r + e \cdot x$
5: **return** $\pi = (u, e, z)$

Algorithm 5 Non-interactive proof verification $(g, h = g^x, \pi = (u, e, z))$

1: $e \stackrel{?}{=} H(g, h, u)$
2: $g^z \stackrel{?}{=} u \cdot h^e$

This non-interactive proof can be then also used as a signature scheme. The only modification is the addition of the message to the hashed values: $e = H(g, h, u, m)$. The message logically must be then sent to the verifier alongside the other values $(\pi = (u, e, z, m))$.

1.5 Homomorphic encryption

Homomorphic encryption [18] allows for specific types of computations to be done on encrypted data as if the data were not encrypted. These operations, usually being addition and multiplication will not reveal the data itself, but the result after decrypting will be the same as if the operation was done on plain text. This type of encryption can be used in cloud computing, for example when a third party is analyzing or doing computations with some private data of its users, and the users want the data to still be private [19].

There are three types of homomorphic encryption:

- **Partially homomorphic encryption:** An operation (either addition or multiplication) can be done on the ciphertext infinite times, while the other one cannot be done.
- **Somewhat homomorphic encryption:** Both operations of addition and multiplication can be done on the ciphertext, but only a limited number of times. Limited to evaluating low-degree polynomials over encrypted data.
- **Fully homomorphic encryption:** Both operations of addition and multiplication can be done an infinite number of times on the ciphertext.

In practice, partially homomorphic schemes are easy to design (compared to the other ones) but can be used only in some applications. These schemes can be used in some multi-party computations. Somewhat homomorphic schemes are harder to create, also it is usually slower. The most complicated to create but probably the most desirable are fully homomorphic schemes. There are some proposed schemes that are fully homomorphic (e.g., Craig Gentry's scheme based on lattices [20]), while these schemes theoretically work, they are too slow for use in the real world.

1.5.1 Paillier cryptosystem

Paillier cryptosystem [4] is an additive partially homomorphic cryptosystem, meaning it can be used to add ciphertexts an unlimited number of times, but not to multiply ciphertexts. However, it can be used to multiply a ciphertext with a plaintext.

It was invented by Pascal Paillier in 1999 and it is based on public key cryptography. It uses the *Decisional Composite Residuosity Assumption* (DCRA). The definition of this problem is that: given a composite n and an integer z, it is hard to decide whether there exists y such that: $z = y^n \, mod \, n^2$. The fact described in equation 1.5 is used in the cryptosystem [21].

$$\frac{(1+n)^x - 1}{n} \equiv x \, mod \, n^2 \tag{1.5}$$

For the purposes of the description of the cryptosystem usually, a function $L(x)$ is defined as $L(x) = \frac{x-1}{n}$, note that since n divides $x - 1$ this is an arithmetic division.

The Paillier cryptosystem consists of three phases: key generation, encryption, and decryption. The set-up of the Paiilier cryptosystem is depicted in Algorithm 6. When the parameters are generated a message can be encrypted as described in Algorithm 7. The ciphertext can then be decrypted as shown in Algorithm 8.

Algorithm 6 Paillier set-up

1: Generate random primes p, q, check that $gcd(p \cdot q, (p-1) \cdot (q-1)) = 1$ (gcd stands for greatest common divider)
2: $n = p \cdot q$
3: $\lambda = lcm((p-1) \cdot (q-1))$ (lcm stands for least common multiple)
4: Generate $g \in_R \mathbb{Z}_{n^2}$
5: Check that n divides the order of g by checking the existence of the multiplicative inverse $\mu = (L(g^\lambda \, mod \, n^2))^{-1} \, mod \, n$
6: Share public key (n, g)
7: Store private key (λ, μ)

Algorithm 7 Encryption (m, public key(n, g))

1: Consider a message $m \in \mathbb{Z}_n$
2: Generate $r \in_R \mathbb{Z}_n$
3: Compute the ciphertext as: $c = g^m \cdot r^n \, mod \, n^2$

Algorithm 8 Decryption (c, n, private key(λ, μ))

1: Consider a ciphertext $c \in \mathbb{Z}_{n^2}$
2: Compute the message as: $m = L(c^\lambda \, mod \, n^2) \cdot \mu \, mod \, n$

Homomorphic properties of Paillier cryptosystem

The first homomorphic property of the Paillier cryptosystem is the homomorphic addition of plaintexts. From Equation 1.6 it is evident that the product of two ciphertexts will decrypt to the sum of the messages. Equation 1.7 then shows how a ciphertext can be multiplied with a plaintext message to get the sum of the two messages when decrypted.

$$Dec(Enc(m_1, r_1) \cdot Enc(m_2, r_2) \, mod \, n^2) = m_1 + m_2 \, mod \, n \tag{1.6}$$

$$Dec(Enc(m_1, r_1) \cdot g^{m_2} \, mod \, n^2) = m_1 + m_2 \, mod \, n \tag{1.7}$$

The second homomorphic property of the Paillier cryptosystem is the homomorphic multiplication of plaintexts. Meaning that by raising a ciphertext to a plaintext message, we get the product of the two messages after decrypting, this is shown in Equation 1.8. However, there is no known way to compute the encrypted product of two encrypted messages.

$$Dec(Enc(m_1, r_1)^{m_2} \, mod \, n^2) = m_1 \cdot m_2 \, mod \, n \tag{1.8}$$

1.6 Group Signatures

Group signatures [1] are signatures that provide anonymity for signers. Usually, there is a group of signers and a manager or some kind of authority. The singers have their own secret used for signing, this secret is computed with some secret value of the authority. Then if a signer sings a message a verifier can verify if the message was signed by someone from the group, but he will not be able to tell by who. In some of the systems, the authority can trace who signed the message by using a special trapdoor. Some systems also support revocation, so if the authority wants to remove the ability to sign the messages from a signer it can be done without affecting the other signers [5].

There are four requirements each group signature scheme should fulfill [22]:

1. **Full Traceability:** Any malicious group of cheating signers and a cheating authority cannot be able to make a valid signature that will look like a signature created by a legit group member who did not actually produce it.
2. **Full Anonymity:** Any verifier without access to the group authority, who is provided with message m and two signatures from group members i and j, cannot tell who signed the message with better certainty than just guessing.
3. **Forward Security:** Members of a group that left the group can no longer sign messages on behalf of the group.
4. **Unlinkability:** Any verifier without access to the group authority, who is provided with messages m_1 and m_2, and two signatures s_1 and s_2, where the signatures are valid, cannot tell if the signatures were generated by the same signer or by two different signers with better certainty than just guessing.

1.7 Attribute-based credentials

Attribute-based credentials are a cryptographic mechanism that provides a way to authenticate using some issued attributes. The main advantage of such a scheme is the selective disclosure of attributes, meaning it is possible to only publish some of the attributes during a verification while keeping the other attributes hidden. These attributes can be for example personal information such as age, name, nationality, or something like a vaccine validity.

In these schemes, there are usually three to four main entities. Set of issuers, that has the power to issue credentials for the users, users that want to be able to selectively prove the ownership of their attributes, a set of verifiers that can verify the ownership proofs of the users, and in most cases a revocation authority that can revoke users' credentials [23].

1.7.1 Keyed-Verification Anonymous Credentials

KVAC scheme introduced in [6] is one of the implementations of attribute-based credentials. It focuses on speed and is aimed at smart cards. The scheme uses the fact that in systems such as public traffic the issuer and verifier are the same entity and therefore can share the private key used to issue the credentials. This key needs to be also used in the verification to make the computation on the client side as fast as possible and avoid using bilinear pairings. The scheme is faster than similar state-of-the-art implementations as shown in [6].

The scheme uses an algebraic MAC introduced by the authors. However, KVAC does not support blind issuance in the original design, meaning the issuer knows how

the credential he issues looks, and the issue algorithm must be done over a secure channel.

The scheme consists of 5 main algorithms: The `Setup` function outputs the system parameters *par* and is parameterized by security parameter 1^κ. The `CredKeygen` function generates the issuer private key *sk* and issuers parameters *ipar*. `Issue` takes as input the issuer's private key and attributes $(m_1, ...m_n)$ and outputs credential *cred* as shown in Algorithm 9.

Algorithm 9 `Issue` $(sk = (x_0, ...x_n), (m_1, ..., m_n))$

1: The issuer receives attributes $(m_1, ...m_n)$.
2: The issuer computes:
3: $\quad \sigma = g_1^{\frac{1}{x_0+m_1x_1+...+m_nx_n}}$
4: $\quad \sigma_{x_1} = \sigma^{x_1}, \sigma_{x_2} = \sigma^{x_2}, ..., \sigma_{x_n} = \sigma^{x_n}$
5: The issuer sends $cred = (\sigma, \sigma_{x_1}, ..., \sigma_{x_n})$ to the user.

The `Obtain` function of the scheme lets the user check the validity of the credential and the attribute value. The `Show` algorithm allows the user to prove ownership of his attributes with selective disclosure. The `ShowVerify` algorithm, is used by the verifier to check the validity of the proof provided by the user. The `Show` and `ShowVerify` algorithms are shown in Algorithms 10 and 11 respectively, the parameter D in the algorithms stands for the set of disclosed attributes.

Algorithm 10 `Show` $(\langle m_i \rangle_{i=1}^n, \sigma, \langle \sigma_{x_i} \rangle_{i=1}^n, D))$

1: The verifier generates and sends $nonce \xleftarrow{\$} \mathbb{Z}_q$ to the user.
2: The user computes:
3: $\quad r, \rho_r, \rho_{m_{i \notin D}} \xleftarrow{\$} \mathbb{Z}_q$
4: $\quad \hat{\sigma} = \sigma^r$
5: $\quad t = \prod_{i \notin D} \sigma_{x_i}^{\rho_m \cdot r} \cdot g^{\rho_r}$
6: $\quad c = \mathcal{H}(D, \langle m_i \rangle_{i \in D}, t, \hat{\sigma}, par, ipar, nonce)$
7: $\quad s_r = \rho_r + cr$
8: $\quad \langle s_{m_i} = \rho_{m_i} - cm_i \rangle_{i \notin D}$
9: The user sends $proof = (\hat{\sigma}, t, s_r, \langle s_{m_i} \rangle_{i \notin D}, \langle m_i \rangle_{i \in D}, D)$ to the verifier.

Algorithm 11 `ShowVerify` $(\langle x_i \rangle_{i=0}^n, proof)$

1: The verifier checks that:
2: $\hat{\sigma} \neq 1_{\mathbb{G}}$
3: $c = \mathcal{H}(D, \langle m_i \rangle_{i \in D}, t, \hat{\sigma}, par, ipar, nonce)$
4: $t \stackrel{?}{=} g^{s_r} \cdot \hat{\sigma}^{-c \cdot x_0 + \sum_{i \notin D}(x_i \cdot s_{m_i}) - \sum_{i \in D}(x_i \cdot m_i \cdot c)}$

1.8 Implementation background

This section describes the technical background of the implementation. That being the languages and libraries and technologies used as well as the Android OS.

1.8.1 Java

Java [30] is an object-oriented programming language first released in 1995 by Sun Microsystems. In 2009 Oracle Corporation bought the company and thus is now the main developer of Java. The newest version of Java is Java SE 19 released on September 20, 2022.

The language is designed to have fewer implementation dependencies than other languages. Java is widely used in computer applications, mobile development, data centers, etc. It is considered to be fast and secure, so it is a good match even for cryptographic applications. Since Java is so widely used, there are a lot of well-tested libraries for it.

Since the Java code is run with *Java Virtual Machine* (JVM), which converts Java bytecode into machine language, the code is platform-independent. Java also has automatic memory management provided by the so-called garbage collector. It also supports multi-thread programming, needed in more performance-heavy applications. *Java Development Kit* (JDK) is a software development environment used for creating Java applications, any programmer wanting to write in Java language needs to have the JDK installed. The JDK converts the written code into Java bytecode. *Java Runtime Environment* (JRE) is software that allows a device to run Java applications. It contains libraries, loader class, and the JVM. The JRE must be installed on a machine that wants to run Java applications. Note that the JRE does not contain any development tools [31].

BigInteger library

BigInteger [32] is an immutable arbitrary-precision integer library. The library allows the developer to use numbers much bigger than normal integers and provides methods for computations with these big numbers. These methods are the operations for modular arithmetic, prime generation, prime testing, comparison methods, and bit manipulation. The object of BigInteger is static, therefore during the operations, the result must always be assigned to a variable. Internally the class uses an array of integers for processing. For using this library in Java no extra library jar file is needed. Only a simple `import java.math.BigInteger;` call is needed in the class that will use its methods.

AMCL library

Apache Milagro Crypto Library (AMCL) [33] is a crypto library that provides functions for symmetric and asymmetric encryption, hashing, and elliptic curve cryptography including bilinear pairings. It also includes its own BIG type for storing big integers. The library can be used with multiple programming languages (Java, C, C++, Python, JavaScript, etc.), but the Java version will be considered in this work. Unlike the BigInteger library, this library must be added to a project with a jar file and added as a dependency, or the source code of the library must be added [34].

The library is not susceptible to the side-channel attack and is completely self-contained, meaning it does not need other dependencies to run, other than a random number generator. The library is available under the Apache-2.0 license. The AMCL supports multiple elliptic curves, these curves include BN curves, BLS curves, NIST curves, and many others.

1.8.2 C language

The C language [35] is a procedural programming language first introduced in 1972 by Ken Thompson and Dennis Ritchie. Initially, it was used for writing operating systems, it was used in the development of the Unix operating system. The main advantages of C are speed and low-level memory access, meaning the developer must take care of the memory management in contrast to Java's garbage collector. it uses a C compiler to compile the code into machine code that can be run on a processor.

Since this language is actively used for almost 50 years there are a lot of well-tested libraries and source codes of all kinds. Since the code supports low-level memory access with pointers it can be much better optimized compared to other high-level languages. Unlike Java, a C application is not platform-independent and must be compiled separately for each processor architecture.

GMP library

The *GNU Multiple Precision* (GMP) [36] arithmetic library is a library for arbitrary precision arithmetic. It works with signed integers, rational numbers, and floating-point numbers. The library aims at uses in cryptography, research, and internet security. The library was first released in 1991, it is primarily written in C and uses the GNU LGPL v3 and GNU GPL v2 licenses, meaning it cannot be used in proprietary software. The main emphasis of the library is on speed.

MCL library

MCL [37] is a library providing fast functions needed in paring-based cryptography. The library was created and is updated by Shigeo Mitsunari. The library supports 4 BN curves and one BLS curve. The library uses the BSD-3 License. The core of the library is written in C but the author provides instructions for compiling the library and using it with Java, JavaScript, or Python. For Java, the author even provides classes with bindings to the C functions.

1.8.3 Android

Android OS [38] is a Linux-based operating system. Android is primarily used in mobile phones, tablets, smart TVs, and cars, its modified version Wear OS is used in smartwatches. Android is open-source and uses the Apache v2 license, the main developer of Android is Google. The system is written in C and C++ and its *User Interface* (UI) is written in Java. Android was first released on September 23, 2008, its latest version Android 13 was released on August 15, 2022.

For the development of Android applications, Android Studio *Integrated Development Environment* (IDE) is usually used. The applications are primarily written in Java or Kotlin. Kotlin [39] is a programming language developed by programmers in IDE Jet Brains specifically for Android. Kotlin functions very similarly to Java internally, but externally it should be easier to use as it does not require type assignments, and the developers do not have to worry about catching exceptions. While Kotlin is very popular, many developers will use Java, since there are more code samples, the code is easily ported to other platforms, and learning a new language does not make sense to them.

Android NDK

Android *Native Development Kit* (NDK) [40] is a tool that allows for C or C++ code to be run on Android devices. The NDK is used in applications where fast calculations provided by C language are needed. This tool can be used in Android Studio since version 2.2. for compiling C code into a native library with the help of Gradle.

One way of running C code on Android is by creating a native C++ project in Android Studio and writing functions to an automatically created native-lib.cpp file and then calling the functions through *Java Native Interface* (JNI). This is a very complicated approach as the developer must take care of all type conversions and library dependencies. Some libraries take care of this for the developer by providing Java classes that call functions in the compiled C library. This way the developer

only adds the compiled library and the Java classes to the project and can use all the functions of the library in Java with the advantage of the speed of C language.

1.8.4 Near-field communication

Near-Field Communication (NFC) [24] is a wireless communication technology that allows two devices to communicate with each other at a very short range. The standards of the technology are specified by NFC Forum [25]. One of the devices is the master active device called the terminal, while the other is called a tag. The tag is usually a passive device that can be powered from the terminal, this can be for example a card. A passive tag communicates by modulating the electromagnetic field broadcasted by the terminal. The tag can also be an active device for example a mobile phone, in that case, it communicates actively by alternating the broadcasts with the terminal. The maximum transfer speed of NFC is about 424 kbit/s, which is rather slow, but the slow speed is balanced by a very fast and easy set-up.

There are three main modes the devices use when communicating through NFC [26]. The first mode is reader/writer mode, in this mode, the device is usually a terminal or a phone that powers an NFC tag and then reads the information transmitted from this tag. The second mode is peer-to-peer, in this mode, two devices communicate actively with each other using NFC and can transfer data of small size. The final mode is card emulation, this mode allows devices such as smartphones or smartwatches that support this technology to act like an NFC tag. But since these smart devices have much bigger computational power than a simple card, this technology can be used to construct more complex systems using NFC.

Host card emulation

Host Card Emulation (HCE)[27] is a technology that allows supported mobile devices to act as an NFC tag using a software solution. On Android devices use of HCE is allowed for Android OS version 4.4 and higher. The card can either be emulated on the CPU or with the use of a secure element. However, the secure element cannot be accessed by regular applications on a non-rooted phone and is only available to Google and its partners' applications [28].

The protocol a device using HCE uses to communicate with a terminal is called *Application Protocol Data Unit* (APDU) protocol. The protocol and the structure of the APDU messages are specified in ISO/IEC 7816-4. The terminal (usually a card reader) sends APDU commands to the phone and the phone can then respond with predefined responses. As a phone is much more powerful than a simple smart card, these defined responses can also contain complex computations. When using this protocol on an Android device, the application using it must have specified a

unique *Application Identifier* (AID). This AID is used by the terminal to construct a `Select AID` command to initiate communication with the desired application.

The structure of an APDU command is shown in Figure 1.3. The command is divided into multiple fields. The first CLA field specifies the class of instruction and is represented by one byte. The second field (INS) is used to determine the code of an instruction, this field can be specified for each application, the same as the next two bytes (P1, P2) specifying the parameters of the instruction. These four fields are mandatory in each APDU command. The Lc field contains information on how many bytes the DATA field holds. For messages shorter than 255 bytes this field is represented by one byte, but in implementations that support data size of up to 65 535 bytes, the field is represented by three bytes, where the first is 0x00. Next is the DATA field that carries the data from an application. The last Le field specifies the size of the data that is expected from the responding device in its data field. The Le field can either be one byte long, two bytes if the extended three-byte Lc field was used, or three bytes if no data were sent by the command but more than 255 bytes are expected back. The Lc, DATA, and Le fields are optional. The APDU response contains an optional DATA field for the application data and two bytes SW1 and SW2. These bytes contain the information about the processing of the command, for success bytes 0x90 0x00 are used [29].

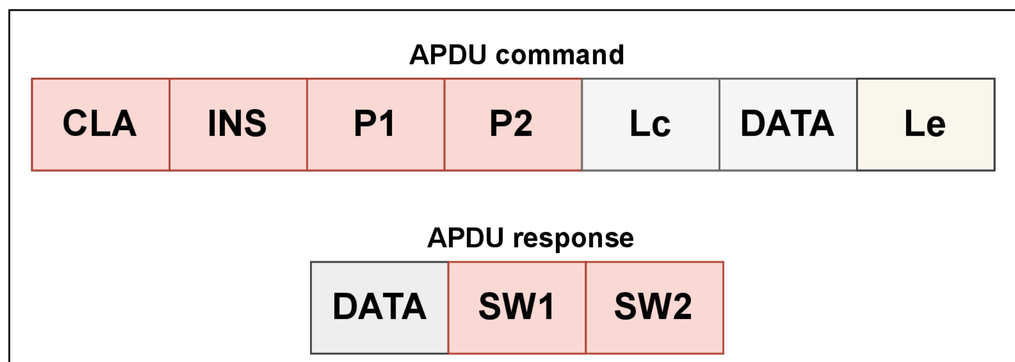


Fig. 1.3: Structure of an APDU command and APDU response.

2 Group signature with two-party computation and blind issuance for attribute-based credentials

This chapter describes individual parts of the group signature scheme that was implemented as well as the modified KVAC scheme that supports blind issuance. The group signature scheme can be used for creating signatures, without the need to share private keys between the entities. This scheme consists of two main parts. The first one is the NIZKPK protocol to compute a secret value that can be then used in a signature, the second one is the group signature scheme itself. The same NIZKPK protocol can also be used to implement blind issuance into the KVAC scheme.

In the group signature scheme, there are three main entities: manager, senders, and verifiers. The manager is a trusted party, and he can create a secret value with a sender, that the sender can then use to sign messages. This secret value is created from the manager's and the sender's private keys. Thanks to the scheme the manager does not know the signer's secret key and vice versa. The computed value is only known to the signer. A verifier can then check the signature with only the knowledge of the public key of the group manager. The verifier will not be able to tell which sender in the group signed the message, but he will be able to verify if the signature is legit for the group. The manager of the group can also open the signatures and determine which sender created the signature if the need occurs. It is also possible to revoke users from the group by publishing a black-list of revoked users.

In the following algorithms, the notation $a \xleftarrow{\$} \mathbb{Z}_n$ means that a is sampled uniformly at random from $\mathbb{Z}_n$. *Proof of Knowledge* (PK) protocols are described by using the notation introduced by Camenisch and Stadler (CS) [41]. The protocol for proving the knowledge of discrete logarithm of c with respect to g is denoted as $\mathrm{PK}\{\alpha : c = g^\alpha\}$. The function $\mathbf{e}(\cdot)$ stands for a pairing function, and $\mathcal{H}$ for a secure hash function. The notation $|a|$ means the bit length of a.

2.1 NIZKPK protcol

The group signature protocol needs a two-party algorithm to compute the value $\sigma = g^{1/(sk_i+sk_m)}$, without revealing the secret values sk_i (user's private key) and sk_m (manager's private key). This can be done by using a NIZKPK introduced in [3]. This algorithm uses Paillier encryption to jointly compute the value of σ while also

generating proofs of knowledge of the values used in the two-party computation by both sides. The NIZKPK algorithm can be divided into two main parts, the setup phase, and the two-party signature.

2.1.1 Setup phase

During the setup phase, the manager generates the parameters needed in the computations, alongside a secret value $\phi(n) = (p-1) \cdot (q-1)$, that is used for decryption on the manager's side. The setup phase can be seen in Algorithm 12. The first multiplicative group generated in Steps 2-4 is mainly for computations of the Paillier cryptosystem, and the second group of order $\mathfrak{n}$ is for the PK. Note that for safety reasons the second group modulo $\mathfrak{n}$ should not be directly generated by the manager. The generated public parameters should be shared with the sender.

Algorithm 12 Setup phase

1: Consider q_{EC} order of used elliptic curve.
2: Generate an RSA modulus $\mathbf{n}$ of size at least $|2^{3\kappa} \cdot q_{EC}^2|$, κ being a security parameter and $\mathbf{n} = p \cdot q$, where p and q are big prime numbers, $|p| = |q|$, $\phi(\mathbf{n}) = (p-1) \cdot (q-1)$.
3: Consider $\mathbf{h} = \mathbf{n} + 1 \in \mathbb{Z}_{n^2}$.
4: Generate $\mathbf{g}$ of order $\phi(\mathbf{n})$ in $\mathbb{Z}_{n^2}$.
5: Obtain another RSA-modulus $\mathfrak{n} = p_g \cdot q_g$, where p_g, q_g are big prime numbers and $|p_g| = |q_g|$.
6: Consider $\mathfrak{h} \xleftarrow{\$} \mathbb{Z}_{\mathfrak{n}}$ and $\mathfrak{g} \xleftarrow{\$} \langle \mathfrak{h} \rangle$.
7: Return public parameters $(\mathbf{h}, \mathbf{n}, \mathbf{g}, \mathfrak{h}, \mathfrak{n}, \mathfrak{g}, q_{EC})$, and store secret $\phi(\mathbf{n})$.

An important parameter of the setup phase is the size of n. As seen in Algorithm 12 on Line 2, this can be computed from the parameters κ and q_{EC}. If we consider parameter κ as 1350 (as specified in similar implementation in [42]), and q_{EC} is the order of the elliptic curve, that will be used (254), we get a bit length of at least 4557 bits. Therefore the generated prime numbers p and q should be half of this bit length. The setup phase is done by the manager and the value of $\phi(n)$ should not be shared but should be stored for later decryption.

2.1.2 Two-party signature

The second part of the NIZKPK protocol is the two-party signature, where the parameters generated in the setup phase are used alongside with manager's and user's private key to compute the value of $\sigma = g^{1/(sk_i + sk_m)}$. During this computation, each party also generates a PK to prove the knowledge of the values used in the

computation. The two-party algorithm is shown in Algorithm 13. Note that while computations in Steps 2-14 are done in a multiplicative group, computations in Steps 15 and 17 are done in an additive group, i.e., in an elliptic curve. After this algorithm, the sender owns the value $\delta_i = \sigma = g^{1/(sk_i+sk_m)}$, that he can use in signing.

Algorithm 13 Two-party signature

1: **Manager computes**:
2: $r \xleftarrow{\$} \mathbb{Z}_{\phi(\mathbf{n})}, r' \xleftarrow{\$} \mathbb{Z}_{\phi(\mathfrak{n})}$
3: $e_1 = \mathbf{h}^{\mathbf{n}/2+sk_m}\mathbf{g}^r \mod \mathbf{n}^2$
4: $\mathfrak{c} = \mathfrak{g}^{sk_m}\mathfrak{h}^{r'} \mod \mathfrak{n}$
5: $PK_m\{sk_m, r, r'\}$
6: **Sender computes**:
7: Check validity of PK_m.
8: $sk_i, r_1 \xleftarrow{\$} \mathbb{Z}_{q_{EC}}, r_2 \xleftarrow{\$} \{0...|2^\kappa q_{EC}|\}, \bar{r} \xleftarrow{\$} \{0...|2^\kappa \mathfrak{n}|\}$
9: $e_2 = (e_1/\mathbf{h}^{\mathbf{n}/2})^{r_1}\mathbf{h}^{\mathbf{n}/2+sk_i r_1+r_2 q_{EC}}\mathbf{g}^{\bar{r}} \mod \mathbf{n}^2$
10: $\mathfrak{c}' = \mathfrak{g}^{sk_i}\mathfrak{h}^{\bar{r}} \mod \mathfrak{n}$,
11: $PK_i\{sk_i, r_1, r_2, \bar{r}\}$
12: **Manager computes**:
13: Check validity of PK_i.
14: $x = (Dec(e_2) - n/2) \ mod \ q_{EC}$, where $Dec(e_2) = ((e_2^{\phi(n)} - 1)/n \ mod \ n^2) \cdot \phi(n)^{-1} \ mod \ n$
15: $\sigma_i = g^{1/x}$
16: **Sender computes**:
17: $\delta_i = \sigma_i^{r_1}$

PK in the two-party signature

This NIZKPK protocol should contain the computation of PK for the manager and for the sender as shown in Steps 5 and 11 of Algorithm 13. In these parts, the two parties need to show that they know the private key used in the computation and that they picked random numbers of the right size. The PKs were not implemented or described in detail in [3], therefore it was needed to construct them. In the manager PK it is required to prove the knowledge of three values at once (sk_m, r, r'), while the sender is proving the knowledge of four values $(sk_i, r_1, r_2, \overline{r})$. In Algorithm 14 it is shown how the manager's PK_m was constructed and in Algorithm 15 how the sender's PK_i is created. Both of these proofs of knowledge are based on Schnorr's non-interactive proof of knowledge.

Algorithm 14 Manager's $PK_m(sk_m, r, r')$

1: **Manager computes**:
2: $\rho_1 \xleftarrow{\$} \mathbb{Z}_{n^2} \cap \mathbb{Z}_{\mathfrak{n}}, \rho_2 \xleftarrow{\$} \mathbb{Z}_{n^2}, \rho_3 \xleftarrow{\$} \mathbb{Z}_{\mathfrak{n}}$
3: $c_1 = h^{\rho_1} \cdot g^{\rho_2} \mod n^2$
4: $c_2 = \mathfrak{g}^{\rho_1} \cdot \mathfrak{h}^{\rho_3} \mod \mathfrak{n}$
5: $e = \mathcal{H}(c_1, c_2)$
6: $z_1 = e \cdot sk_m + \rho_1$
7: $z_2 = e \cdot r + \rho_2$
8: $z_3 = e \cdot r' + \rho_3$
9: Send $PK_m = (z_1, z_2, z_3, e)$ to the sender.
10: **Sender computes**:
11: $c'_1 = h^{z_1} \cdot g^{z_2} \cdot (\frac{e_1}{h^{n/2}})^{-e} \mod n^2$
12: $c'_2 = \mathfrak{g}^{z_1} \cdot \mathfrak{h}^{z_3} \cdot \mathfrak{c}^{-e} \mod \mathfrak{n}$
13: $e \stackrel{?}{=} \mathcal{H}(c'_1, c'_2)$

Algorithm 15 Sender's $PK_i(sk_i, r_1, r_2, \overline{r})$

1: **Sender computes**:
2: Consider $\alpha = e_1/h^{n/2},\ \beta = \mathfrak{h}^{q_{EC}},\ sk'_i = sk_i \cdot r_1,\ u = -\overline{r} \cdot r_1$
3: $\rho_s \xleftarrow{\$} \mathbb{Z}_q,\ \rho_1, \rho', \rho_u, \overline{\rho} \xleftarrow{\$} \mathbb{Z}_{\mathfrak{n}},\ \rho_2 \xleftarrow{\$} \mathbb{Z}_{n^2}$
4: $c_1 = \alpha^{\rho_1} \cdot h^{\rho'} \cdot \beta^{\rho_2} \cdot g^{\overline{\rho}} \mod n^2$
5: $c_2 = \mathfrak{g}^{\rho_s} \cdot \mathfrak{h}^{\overline{\rho}} \mod \mathfrak{n}$
6: $c_3 = \mathfrak{c}'^{\rho_1} \cdot (1/\mathfrak{g})^{\rho'} \cdot \mathfrak{h}^{\rho_u} \mod \mathfrak{n}$
7: $c_4 = g_2^{\rho_s} \mod q_{EC}$
8: $e = \mathcal{H}(c_1, c_2, c_3, c_4)$
9: $z_s = e \cdot sk_i + \rho_s$
10: $z_1 = e \cdot r_1 + \rho_1$
11: $z_2 = e \cdot r_2 + \rho_2$
12: $z_u = e \cdot u + \rho_u$
13: $\overline{z} = e \cdot \overline{r} + \overline{\rho}$
14: $z' = e \cdot sk'_i + \rho'$
15: Send $PK_i = (z_s, z_1, z_2, z_u, \overline{z}, z', e)$ to the manager.
16: **Manager computes**
17: $c'_1 = \alpha^{z_1} \cdot h^{z'} \cdot \beta^{z_2} \cdot (e_2/h^{n/2}) - e \mod n^2$
18: $c'_2 = \mathfrak{g}^{z_s} \cdot \mathfrak{h}^{\overline{z}} \cdot (\mathfrak{c}')^{-e} \mod \mathfrak{n}$
19: $c'_3 = \mathfrak{c}'^{z_1} \cdot (1/\mathfrak{g})^{z'} \cdot \mathfrak{h}^{z_u} \mod \mathfrak{n}$
20: $c'_4 = g_2^{z_s} \cdot pk_i^{-e} \mod q_{EC}$
21: $e \stackrel{?}{=} \mathcal{H}(c'_1, c'_2, c'_3, c'_4)$

2.2 Group signature

The signature used in this scheme is a combination of the weak Boneh-Boyen signature and Schnorr's proof of knowledge protocol. It is based on the short group signature introduced in [5] and was first proposed in [43]. The standard version of the weak Boneh-Boyen signature uses the public key of the signer to check the validity of the signature, and this is not desired in a group signing scheme. The pairing property of the signature is used in the scheme to check if the signature is valid with respect to the manager's public key, and the Schnorr protocol is utilized to prove the knowledge of a signed message anonymously. The scheme is also used in [44], where the authors also provide all the proofs needed. The algorithm of the signature is displayed in Algorithm 16, and its verification is in Algorithm 17.

Algorithm 16 Sign (sk_i, δ_i, m)

1: Generate r $\in_R \mathbb{Z}_q$
2: $g' = g^r$
3: $\delta_i' = \delta_i^r$
4: $\overline{\delta_i} = (\delta_i')^{-sk_i}$
5: Generate $\rho_r, \rho_{sk_i} \in_R \mathbb{Z}_q$
6: $t = (\delta_i')^{\rho_{sk_i}} \cdot g^{\rho_r}$
7: $e = \mathcal{H}(g', \delta_i', \overline{\delta_i}, t, m)$
8: $S_r = \rho_r - e \cdot r$
9: $S_{sk_i} = \rho_{sk_i} + e \cdot sk_i$
10: $par = (g', \delta_i', \overline{\delta_i}, m, e, S_r, S_{sk_i})$

Algorithm 17 Verify (par, pk_m)

1: $\hat{t} = (\overline{\delta_i} \cdot g')^e \cdot \delta_i'^{S_{sk_i}} \cdot g^{S_r}$
2: $e' = \mathcal{H}(g', \delta_i', \overline{\delta_i}, \hat{t}, m)$
3: $\mathbf{e}(\overline{\delta_i} \cdot g', g_2) \stackrel{?}{=} \mathbf{e}(\delta_i', pk_m)$
4: $e \stackrel{?}{=} e'$

All the operations are done in an additive group of a pairing-friendly elliptic curve. The hash function $\mathcal{H}$ should be chosen in a way that $e \in \mathbb{Z}_q$. In the Verify algorithm, pk_m is a public key of the manager, where $pk_m = g_2^{sk_m}$.

Theorem 1. *The verification in Algorithm 17 is correct.*

Proof. In order to prove the correctness of the verify algorithm, we need to show the equality $t = \hat{t}$ in the first step.

$$\hat{t} = (\bar{\delta}_i' \cdot g')^e \cdot \delta'^{s_{sk_i}}_i \cdot g^{s_r}$$
$$\hat{t} = (\delta'^{-sk_i}_i \cdot g^r)^e \cdot \delta'^{s_{sk_i}}_i \cdot g^{s_r}$$
$$\hat{t} = \delta'^{-esk_i}_i \cdot g^{er} \cdot \delta'^{s_{sk_i}}_i \cdot g^{s_r}$$
$$\hat{t} = \delta'^{-esk_i}_i \cdot g^{er} \cdot \delta'^{\rho_{sk_i} - esk_i}_i \cdot g^{s_r}$$
$$\hat{t} = g^{er} \cdot \delta'^{\rho_{sk_i}}_i \cdot g^{s_r}$$
$$\hat{t} = g^{er} \cdot \delta'^{\rho_{sk_i}}_i \cdot g^{\rho_r - er}$$
$$\hat{t} = \delta'^{\rho_{sk_i}}_i \cdot g^{\rho_r} = t.$$

Since $t = \hat{t}$ the equation $\mathcal{H}(g', \delta'_i, \overline{\delta_i}, t, m) = \mathcal{H}(g', \delta'_i, \overline{\delta_i}, \hat{t}, m)$ also holds for valid signature. As the scheme uses cryptography based on bilinear pairings, the bilinear properties can be used to prove that the pairings will also be equal for a valid signature.

$$\begin{aligned}
\mathbf{e}(\overline{\delta_i} \cdot g', g_2) &= \mathbf{e}(\delta'_i, pk_m) \\
\mathbf{e}(\delta_i^{-sk_i \cdot r} \cdot g^r, g_2) &= \mathbf{e}(\delta_i^r, g_2^{sk_m}) \\
\mathbf{e}(g^{\frac{-sk_i \cdot r}{sk_m + sk_i}} \cdot g^r, g_2) &= \mathbf{e}(\delta_i^r, g_2^{sk_m}) \\
\mathbf{e}(g^{\frac{r \cdot (sk_m + sk_i - sk_i)}{sk_m + sk_i}}, g_2) &= \mathbf{e}(\delta_i^r, g_2^{sk_m}) \\
\mathbf{e}(\delta_i^{sk_m \cdot r}, g_2) &= \mathbf{e}(\delta_i^r, g_2^{sk_m}) \\
\mathbf{e}(\delta_i, g_2)^{sk_m \cdot r} &= \mathbf{e}(\delta_i, g_2)^{sk_m \cdot r}
\end{aligned}$$

2.2.1 Revocation and opening of the signature

While in a group signature scheme, the identity of the signer should be hidden, it is desirable for the manager to be able to open the signature and learn the identity of the signer, and to be able to revoke users from the group. In this group signature scheme that is possible.

The open function

For the opening, the manager of the group must keep a private list of inverted public keys $\overline{pk_i} = pk_i^{-1}$ of the senders in the group, ideally with the addition of their identifiers. The manager then needs to acquire values δ'_i and $\overline{\delta_i}$ of a signature and check the equality of the pairings $\mathbf{e}(\delta'_i, \overline{pk_i}) \stackrel{?}{=} \mathbf{e}(\overline{\delta_i}, g_2)$ for each of the $\overline{pk_i}$ he has saved. If the pairings are equal the user whose is the $\overline{pk_i}$ is the signer.

Theorem 2. *The open function is correct.*

Proof. In order to prove the correctness of the open function it must be shown that for sk_i used in the signature, such that $pk_i = g_2^{sk_i}$ the equation $\mathbf{e}(\delta'_i, \overline{pk_i}) = \mathbf{e}(\overline{\delta_i}, g_2)$ holds.

$$\begin{aligned}
\mathbf{e}(\delta'_i, \overline{pk_i}) &= \mathbf{e}(\overline{\delta_i}, g_2) \\
\mathbf{e}(\delta'_i, pk_i^{-1}) &= \mathbf{e}(\overline{\delta_i}, g_2) \\
\mathbf{e}(\delta'_i, g_2^{-sk_i}) &= \mathbf{e}(\delta_i^{-sk_i}, g_2) \\
\mathbf{e}(\delta'_i, g_2)^{-sk_i} &= \mathbf{e}(\delta_i, g_2)^{-sk_i}
\end{aligned}$$

Revocation

For revocation of the users (removing a user from the group), the creation of a blacklist can be used. The manager of the group will publish a blacklist with the values of $\overline{pk}_i = pk_i^{-1}$ of revoked users to the verifiers. The verifier must then check the same pairing $\mathbf{e}(\delta'_i, \overline{pk}_i) \stackrel{?}{=} \mathbf{e}(\overline{\delta}_i, g_2)$ for each entry in the blacklist during the verification. If for any of the entries in the blacklist, this pairing is equal, the signature is by a revoked user and therefore not valid.

The use of a blacklist requires extra computations of pairings during the verification of the signature. This can be computationally demanding for a big amount of users. But in a group signing scheme, the groups will usually not be bigger than 100 users and only a few of them will be revoked. If the blacklist is too long, the manager can regenerate the group, by creating a new manager (group) key pair and adding the current users to the new group.

Another important factor to consider for the deployment of this scheme is, that the public keys pk_i used by the users should be generated specifically for this group signature scheme. If for example a public key from a public certificate would be used, the verifier could obtain the public keys of the members of the group and find out who from the group created the signature. Practically this should not be a problem as the scheme uses different keys ($pk_i = g_2^{sk_i}$) than standard algorithms such as *Rivest–Shamir–Adleman* (RSA) and *Digital Signature Algorithm* (DSA).

2.3 Using the NIZKPK in KVAC

The NIZKPK protocol shown in Section 2.1 can also be used to introduce blind issuance into an attribute-based credential scheme like KVAC. Such a modification to the KVAC protocol makes the credential issued by the issuer private to the user, therefore hidden from the issuer. This way, the issuer will be not able to impersonate any user. In this modification to the KVAC protocol, the user is in possession of a private key that is not shared with the issuer. The protocol also protects the credential from an attack during the issue phase, as the real credential can only be extracted by the user.

2.3.1 Modifying the issue algorithm

In the standard `Issue` algorithm shown in Section 1.7 in Algorithm 9, the user sends his attributes to the issuer, and the issuer signs them with his private key, creating the credential. If we apply the blind issuance with the NIZKPK protocol, the final credential will be signed jointly by the issuer and the user as well. The modified

version of the KVAC issue algorithm is depicted in Algorithm 18, with the changes highlighted in red.

Algorithm 18 `Issue` $(sk = (x_0, ...x_n), (m_1, ...m_n), sk_i)$

1: The issuer receives attributes $(m_1, ...m_n)$.
2: The issuer computes: $d = x_0 + m_1x_1 + ... + m_nx_n$.
3: Run NIZKPK 2-party computation to compute: $k = (d + sk_i)r_1$
4: The issuer computes:
5: $\sigma^* = g_1^{1/k}$
6: $\sigma^*_{x_1} = \sigma^{*x_1}$,$\sigma^*_{x_2} = \sigma^{*x_2}$,..., $\sigma^*_{x_n} = \sigma^{*x_n}$
7: The issuer sends $\sigma^*, \sigma^*_{x_1}, ..., \sigma^*_{x_n}$ to the user.
8: The user computes $\sigma = \sigma^{*r_1}, ..., \sigma_n = \sigma_n^{*r1}$.

In Step 3 of Algorithm 18, the NIZKPK algorithm shown in Section 2.1 is executed. The change is that the manager (or the issuer in this case) uses the value of $d = x_0+m_1x_1+...+m_nx_n$ instead of the private key sk_m used in the group signature scheme. Since the credential sent by the issuer is now randomized by r_1, put in the computation by the user, no one, but the user is able to extract the final credential from the values $\sigma^*, \sigma^*_{x_1}, ..., \sigma^*_{x_n}$. So not only does the blind issuance enhance the privacy of the user by hiding his credential from the issuer, but it also makes it harder for an attacker to obtain the credential by intercepting the communication during the `Issue` algorithm.

2.3.2 Modifying the Show and ShowVerify algorithms

For the user to be able to prove the ownership of his credential in this modified scheme, changes to the other main algorithms of the KVAC scheme were needed. In the `Show` algorithm, the user must now also prove the knowledge of sk_i used in the issue algorithm. The modified version of the `Show` algorithm is displayed in Algorithm 19, and the altered `ShowVerify` algorithm is then shown in Algorithm 20.

Algorithm 19 `Show` $(\langle m_i \rangle_{i=1}^{n}, \sigma, \langle \sigma_{x_i} \rangle_{i=1}^{n}, D), sk_i)$

1: The verifier generates $nonce \xleftarrow{\$} \mathbb{Z}_q$ and sends it to the user.
2: The user computes:
3: $r, \rho_r, \rho_{m_{i \notin D}}, \rho_u \xleftarrow{\$} \mathbb{Z}_q$
4: $\hat{\sigma} = \sigma^r$
5: $t = \prod_{i \notin D} \sigma_{x_i}^{\rho_m \cdot r} \cdot g^{\rho_r} \cdot \sigma^{\rho_u \cdot r}$
6: $c = \mathcal{H}(D, \langle m_i \rangle_{i \in D}, t, \hat{\sigma}, par, ipar, nonce)$
7: $s_r = \rho_r + c \cdot r$
8: $s_u = \rho_u - c \cdot sk_i$
9: $\langle s_{m_i} = \rho_{m_i} - c m_i \rangle_{i \notin D}$
10: The user sends $proof = (\hat{\sigma}, t, s_r, \langle s_{m_i} \rangle_{i \notin D}, s_u, \langle m_i \rangle_{i \in D}, D)$ to the verifier.

Algorithm 20 `ShowVerify` $(\langle x_i \rangle_{i=0}^{n}, proof)$

1: The verifier checks that:
2: $\hat{\sigma} \neq 1_{\mathbb{G}}$
3: $c = \mathcal{H}(D, \langle m_i \rangle_{i \in D}, t, \hat{\sigma}, par, ipar, nonce)$
4: $t \stackrel{?}{=} g^{s_r} \cdot \hat{\sigma}^{-c \cdot x_0 + \sum_{i \notin D}(x_i \cdot s_{m_i}) - \sum_{i \in D}(x_i \cdot m_i \cdot c) + S_u}$

Theorem 3. *The verification in Algorithm 20 is correct.*

Proof. In order to prove the correctness of the modified KVAC verification algorithm we need to show the equality $t = g^{s_r} \cdot \hat{\sigma}^{-c \cdot x_0 + \sum_{i \notin D}(x_i \cdot s_{m_i}) - \sum_{i \in D}(x_i \cdot m_i \cdot c) + S_u}$.

$$
\begin{aligned}
& g^{s_r} \cdot \hat{\sigma}^{-c \cdot x_0 + \sum_{i \notin D}(x_i \cdot s_{m_i}) - \sum_{i \in D}(x_i \cdot m_i \cdot c) + S_u} \\
= \ & g^{\rho_r + c \cdot r} \cdot \hat{\sigma}^{-c \cdot x_0 + \sum_{i \notin D}(x_i \cdot s_{m_i}) - \sum_{i \in D}(x_i \cdot m_i \cdot c) + \rho_u - c \cdot sk_i} \\
= \ & g^{\rho_r + c \cdot r} \cdot \sigma^{r \cdot (-c \cdot x_0 + \sum_{i \notin D}(x_i \cdot (\rho_{m_i} - c \cdot m_i)) - \sum_{i \in D}(x_i \cdot m_i \cdot c) + \rho_u - c \cdot sk_i)} \\
= \ & g^{\rho_r + c \cdot r} \cdot \sigma^{r \cdot (\sum_{i \notin D}(\rho_{m_i} \cdot x_i) + \rho_u)} \cdot \sigma^{-c \cdot (x_0, + \sum_{i \in D}(x_i \cdot m_i + sk_i)) \cdot r} \\
= \ & g^{\rho_r + c \cdot r} \cdot \sigma^{r \cdot \sum_{i \notin D}(\rho_{m_i} \cdot x_i) + \rho_u} \cdot g^{-c \cdot r} \\
= \ & g^{\rho_r} \cdot \sigma^{r \cdot (\sum_{e \notin D}(\rho_{m_i} \cdot x_i) + \rho_u)} \\
= \ & \prod_{i \notin D} \sigma^{\rho_{m_i} \cdot r} \cdot g^{\rho_r} \cdot \sigma^{r \cdot \rho_u} = t
\end{aligned}
$$

3 Practical implementation

This chapter deals with the implementation of the group signature scheme with the NIZKPK protocol as described in Chapter 2. Speed tests of some libraries were conducted, to help choose a library for parts where the speed of computation is essential, i.e., mainly for the NIZKPK computations and for the signing and verification algorithm. The chapter also includes a part that deals with the implementation of blind issuance into the KVAC scheme.

The implemented group signature scheme was then used to build a small system for the demonstration of the protocol. This system contains a PC application with the ability to simulate the role of the group manager and a verifier and a user's Android application. The user uses a mobile device such as an Android mobile phone to take part in the 2-party computation and to generate signatures with the key saved in the mobile device. The implemented group signature scheme was used to create a system for digitally signing documents. The devices communicate through NFC.

The PC implementation is split into three smaller applications. The first is for a group manager and allows for the creation of a group and managing the members of the group. The second is the verifier's application which allows a verifier to check the validity of signatures. The third PC application is for the user, it allows him to choose a PDF file that he wants to sign, then he uses the key saved in his mobile application to generate a signature for the file. The hash of the file is sent from the PC application through NFC to the phone. The phone then sends a response with the generated signature. The signature is then saved to the metadata of the PDF file by the PC application.

3.1 Choice of a platform and libraries

First, it was necessary to choose the devices to use and consequently the language for the implementation. As the implementation is aimed at Android smartphones as the signer's main device and a PC as an issuer's or verifier's device, it was important to choose the right platform for the implementation.

Since we wanted to make our PC application portable to other devices without problems, Java on Windows was a good choice, as it is possible to create a simple jar file and distribute it with the compiled external libraries (be it Java library in .jar format or native C library in .dll format). This can be harder to do for C as it usually requires the libraries to be built for the system it runs on. Java also provides good tools to create *Graphical User Interface* (GUI) and is used in Android, and

with the help of JNI, C can be used in the background to make some computations faster.

This way we only need a Windows PC with JDK installed. The Java program can also be ported to a Linux system, however, the external JNI libraries for C should be built separately. This ensures compatibility of the libraries on different architectures, as they are compiled into machine code. The program includes a script that should be able to install all the dependencies and build the libraries on a fresh Linux system like Ubuntu.

The Android mobile phone is used to communicate through NFC with a terminal during the 2-party computation and for sending the signature to the computer. The choice of using Java on both devices also allows us to port the code between the devices, as some parts of the code are the same for the entities in the scheme. As most of the Android applications are developed in Java and Kotlin, the implementation is mainly in Java with the help of some JNI libraries that allow the calling of functions implemented in the native C language from Java. Kotlin is not the best suit for this application, as Java implementation is then easier to transfer to a computer [39].

3.1.1 Comparison of Java and C libraries on an Android device

There are two main ways how to use a C code in an Android application. The first one is implementing functions in C and using the NDK and JNI to pass from Java to C functions. But with this option, we must consider data type conversions and problems with using various C libraries that can be tricky to import. Then there is a second way. Having a C library prepared and compiled by the authors for Java or Android, where the bindings are already created. In this way, we have the functions of the library running in C with the help of JNI, but we just simply call Java functions and do not have to worry about any data conversions as those were implemented by the authors.

The main focus of this comparison was to find out if a C library called through JNI is faster than a similar Java library. Since this scheme uses homomorphic encryption in multi-party computation and bilinear pairings for signatures, we need a library for modular arithmetic with big numbers and a library for bilinear pairings. All tests were conducted on Xiaomi Redmi Note 8 Pro with 6 GB RAM, MediaTek Helio G90T 8 core 2,05GHz CPU, and Arm Mali-G76 3EEMC4 800MHz GPU. All tests were performed 10 times and the resulting time is their average.

Libraries providing bilinear pairings

There are not that many implementations of bilinear pairings for Java. One with big enough, i.e., with the right security level, pairing-friendly curves is the AMCL

library. This library provides a big variety of curves of different sizes, which is its main advantage. For C, the MCL library was chosen. The biggest advantage of MCL over other C libraries is, that it has a prepared JNI implementation that makes calling its functions easier from Java on an Android device. The biggest disadvantage of the MCL library is, that it only supports two curves, but since one is the BN-254 curve, which is suitable for the implementation, this library suits our implementation perfectly.

The main focus of comparing these two libraries is finding out which is faster on an Android device. Functions tested and timed were: adding two points in G1, scalar multiplication in G1 and G2, scalar multiplication by q-1 in G1, and pairing. Results are depicted in Table 3.1, where times are in μs.

	C - MCL	Java - AMCL
Adding two points in G1	5,4	26,8
Scalar Multiplication in G1	231	6 700
Scalar Multiplication in G2	504	22 836
Multiplication by q-1 in G1	172	9 779
Pairing	1 423	27 966

Tab. 3.1: Comparison of MCL and AMCL computation times (in μ s) on an Android device.

Table 3.1 shows that the MCL library is much faster than AMCL. In particular Scalar multiplication in G1 takes almost 30 times more time with the AMCL library, and in G2 it's even more. With this finding, the decision was easy and the MCL library was chosen for the implementation of the signature.

Libraries for modular arithmetic

For modular arithmetic, is needed a library that can operate with big numbers in a multiplicative group. In Java, the most widely used library for big numbers is BigInteger [32], it is perfectly suitable as it is well-tested, and contains all needed functions. For C language a GMP library [36] was chosen, since it can operate with big numbers, is targeted for cryptography, and has a general emphasis on speed. It was also used in the implementation of Secure Two-Party Computation by M. Sečkár [45].

Using the GMP library with Android is harder than the previously mentioned MCL library because while it can be compiled into a binary library, that can be used in Android, the creators don't provide Java bindings that would make calling the functions easier. Therefore, one must first implement functions in a cpp file,

that provide type conversions from Java to C and back through JNI. These needed conversions also add computational time, so the C functions must be faster by a lot to make it worth it.

The speed tests were conducted for addition, multiplication, modular exponentiation, and prime generation. While testing the speed of the libraries, it was found that function `mpz_powm` used by the GMP library can be susceptible to side-channel attacks. Authors themselves say, that in cryptographic applications `mpz_powm_sec` should be used instead [46]. Therefore, both of these functions were tested. While the `gmp_powm` function can be susceptible to side-channel attacks, this attack is not easily feasible against a mobile device such as a mobile phone. For the prime generation function `probablePrime` was used from the BigInteger library and from GMP it was `mpz_nextprime`. As GMP does not have a function for generating a random prime, and it only finds the next probable prime, we pass to it a random number of desired bit size. The results of the speed tests are depicted in Tables 3.2 and 3.3.

	BigInt Add	Gmp Add	BigInt Mul	Gmp Mul
1024 bits	29,7	148,2	27,5	125,1
2048 bits	35,1	425,7	30,5	387,3
4096 bits	38,9	1 428	37,5	1 450
8192 bits	61,5	5 298	85,3	5 320

Tab. 3.2: Comparison of BigInteger and GMP computation times for addition and multiplication on an Android device (times in μs).

	BInt powM	GMP powM	GMP powMsec	BInt prime	GMP prime
1024 bits	1 910	2 036	2 147	659 378	160 343
2048 bits	14 047	13 262	18 767	5 833 261	2 675 519
4096 bits	109 990	74 750	145 872	—	—
8192 bits	857 750	413 501	1 135 589	—	—

Tab. 3.3: Comparison of BigInteger and GMP computation times for modular exponentiation and prime generation on an Android device (times in μs).

Looking at the addition and multiplication operations results in Table 3.2, it seems like the GMP library is much slower for these operations. The reason is that the time is calculated with type conversions from Java to C and back since in a real program this time will influence the final time of a protocol because without these conversions it cannot be used on Android phones. The times of the operations of addition and multiplication with GMP without the conversions are comparable with BigInteger computation times.

If we focus on the most important and time-consuming operations, with modular exponentiation we can see in Table 3.3, that the normal `mpz_powm` function from GMP is faster, and the difference is more significant with bigger bit size. However, the secure version of GMP modular exponentiation is slower than BigInteger implementation, and here it's not only because of the type conversion, but the operation itself is slower too. As said before, we do not consider the attack against the `mpz_powm` function relevant on a mobile device, so it could be used to save some time in more demanding computations. Interesting are the times of generating primes. Here the C GMP library is significantly faster. This operation is used in the setup of the key agreement algorithm, but the setup will not be run on the mobile device.

3.1.2 Comparasion of Java and C libraries on a desktop PC

The issuer and verifier part of the scheme are implemented on a desktop PC. While for this kind of device, the performance is not as crucial as for the mobile device, it is still good to use faster implementations. The libraries tested are the same as for the mobile device. The device used in these benchmarks was HP Pavilion 15-bc5xxx, Intel i5-9300H (4cores, 2,4-4,1GHz), 16GB RAM, and Windows 10 Home 20H2.

Libraries providing bilinear pairings on Windows PC

Tested libraries providing bilinear were the same as for the Android mobile phone (AMCL and MCL). The most important operation on the PC part of the application is the pairing, as it is needed in the verification, for checking the revocation, and for opening the signatures by the manager. The curve used in the tests was the BN-254 curve and the functions benchmarked were: addition of two random points in G1, multiplying a point by a random scalar, and pairing. The results are shown in Table 3.4.

	C - MCL	Java - AMCL
Adding two points in G1	5,1	9,9
Scalar Multiplication in G1	417	2 472
Scalar Multiplication in G2	873	4 640
Pairing	2 621	8 358

Tab. 3.4: Comparison of MCL and AMCL computation times (in μ s) on a Windows PC.

While the MCL library is faster also on a desktop PC, it is interesting to see that the MCL library is actually slower than on an Android mobile device. This is probably due to the fact, that the library is not optimized for Windows. But it still

gives us the advantage of superior speed (compared to AMCL). Also on Windows, the compiled portable library can be used without the need to build and install the library on each new device. The MCL library is therefore also used in the desktop application.

Modular arithmetic libraries on Windows PC

The libraries considered for the modular arithmetic were the same as on the phone - GMP and BigInteger. The main tested functions were modular exponentiation and generating prime numbers, as those are the most demanding functions. Table 3.5 shows the difference between GMP and Java's Big Integer for prime generation and modular exponentiation on a Windows PC.

	BigInt prime	GMP Prime	BigInt powM	GMP powM
1024 bits	52	201	1,8	2,1
2048 bits	529	1 331	8,1	15,7
4096 bits	-	-	40	93
8192 bits	-	-	269	574

Tab. 3.5: Comparison of BigInteger and GMP computation times for modular exponentiation and prime generation on a Windows PC (times in ms).

While testing the GMP library on a Windows PC, it was found that the library is mainly aimed at Unix-type systems [36]. While it is possible to build GMP for Windows, it is not optimized and is much slower than it could be on a similar machine with a distribution of Linux OS. Table 3.5 shows that BigInteger has better results in all the tested functions on Windows. This makes the GMP library not suitable for the server implementation for Windows in Java. Note that the GMP is even called only from C without the Java in between and it is still slower. The faster version of `mpz_powm` was also used with GMP.

Libraries providing bilinear pairings on Linux VM

Because the MCL library was slower on the Windows PC than on the mobile phone, tests on a Linux-based system were also performed, to see if the optimized version for Linux would get better results. The tests were conducted on a *Virtual Machine* (VM) running on the same PC with OS: Ubuntu 64-bit Linux (VMWare, RAM: 8 GB, CPU: 4 cores). Table 3.6 shows the results of the benchmarks.

The comparison of the MCL and AMCL libraries turned out the same as on the other devices, where MCL has a big advantage in performance compared to AMCL.

	C - MCL	Java - AMCL
Adding two points in G1	4.6	10.3
Scalar Multiplication in G1	118	3 678
Scalar Multiplication in G2	170	6 232
Pairing	707	9 946

Tab. 3.6: Comparison of MCL and AMCL computation times (in μ s) on Linux VM.

The MCL library is also much faster on Linux than on Windows. It is probably because it is better optimized than the portable Windows version.

Modular arithmetic libraries on Linux VM

As the application will also be run on Linux, and GMP did not perform as expected on Windows, speed tests on a Linux-based system were also performed to compare GMP with BigInteger. This was mainly to determine if GMP support should be included in the PC application. The results of the benchmarks are in Table 3.7.

	BInt prime	GMP Prime	BInt powM	GMP powM	GMP powMsec
1024 bits	35	37	2.9	0.9	1.2
2048 bits	497	315	7.8	4.2	5.9
4096 bits	-	-	39	27	37.9
8192 bits	-	-	292	171	273

Tab. 3.7: Comparison of BigInteger and GMP computation times for modular exponentiation and prime generation on Linux VM (times in ms).

As seen in the table, GMP does perform better in a Linux environment compared to Windows. Since on Linux GMP was faster for modular exponentiation, the more secure function `mpz_powm_sec` was also tested. We can see that the modular exponentiation with 4096 and 8192-bit numbers is much faster with the `mpz_powm` function than BigInteger. However, the `mpz_powm_sec` function is almost the same speed as BigInteger. Therefore in the PC application, the faster version of the GMP function is used as an alternative, so the protocol performs better on Linux. If there is a possibility of a side-channel attack on the device, then the switch to BigInteger will solve that concern.

Since the results of the modular arithmetics were different for Windows and for Linux, the application has the option to switch between BigInteger and Linux. The recommendation is to use BigInteger on Windows and on Linux GMP, and if there is a concern about a side-channel attack it is better to use BigInteger on Linux also.

3.2 Implementing the cryptographic core

This section deals with the creation of the cryptographic core of the application. This core is shared for both the desktop and the mobile applications, although each uses only some of its functions. This core uses the BigInteger library for computations during the NIZKPK protocol (with the possibility to use GMP for some operations), and the MCL library in the group signature.

The classes of this core are located in the `cryptocore` package. The package should be self-contained, and usable in other projects with the need to import the MCL library, it is a good idea to turn off the GMP usage in other projects as it has to be implemented separately.

3.2.1 Integrating the libraries into a project

In this section, it is shown how the libraries can be integrated into the project on a Windows PC and on an Android device. It is also shown how the application can be run on Linux. This section should also help anyone that would like to create a similar implementation. Note that for Android NDK needs to be installed for Android Studio and JNI for Java on desktop.

Importing the MCL library to Java

The MCL library can be downloaded from [37]. For PC, first, the C library must be built, the steps on how to do it on Linux or on Windows with the help of Visual Studio are shown in the `readme.md` file of the library, and on the GitHub page. After compiling the C library, folder `ffi/java` provides files that can be used to build a library with bindings for Java. The directory also provides instructions on how to build it. On Linux make sure you have compiled `mcljava.so` in the `/usr/lib` directory. Another option is also using the MCL wrap [47] that includes scripts to build the library for Java with just one script, even though this might not be an optimal build on Windows, it creates a portable library and is easy to use. Since building the library on Windows can be troublesome, anyone that would like to use the library from this implementation can just download the compiled library file `mcljava-x64.dll` distributed with the application for group signatures. After the library is built it must be added to the project libraries and the Java classes from package `com.herumi.mcl` must be added to the project. After that, the program that will be using it must call `System.loadLibrary("mcljava-x64")` to load the library and initialize MCL with the curve it wants to use: `Mcl.SystemInit(Mcl.BN254)`. The load call will be different for Linux as the library might have a different name.

For Android, the library can be downloaded from [48], the page also provides steps on how to build the library using the NDK. The application created also provides these built libraries that can be used in an Android project. The built library must be then imported into a resource directory (for example `jniLibs`) into separate directories for different architectures. Then in the main function similar the library must be loaded by calling `System.loadLibrary("mcljava");` and initialized it in the same way as above with a curve. Also, the Java classes must be added to the project the same as before.

Using GMP on Android

GMP can be used on Android by importing an already build GMP library for each architecture from [49]. This GMP library can then be passed to the same `jniLibs` folder in the Android project. Then it is needed to create an `Android.mk` file in the `jniLibs` folder to be able to create functions that can be then called from Java, the contents of the file are shown in Listing 3.1. It specifies the file `test.cpp` that can then be used to implement functions with the use of GMP and the name of the module that will be used to load the library in Java with the `System.loadLibrary()` command. Finally, it is needed to pass to Gradle the location of the `Android.mk` file for the NDK build. This is done by adding the code shown in Listing 3.2 to the `build.graddle` file of the application.

Listing 3.1: Android.mk file in jniLibs

```
JNI_PATH := $(call my-dir)
include $(JNI_PATH)/gmp/Android.mk

LOCAL_PATH := $(JNI_PATH)
include $(CLEAR_VARS)

LOCAL_MODULE := gmp-tests
LOCAL_SRC_FILES := test.cpp

LOCAL_LDLIBS += -llog
LOCAL_SHARED_LIBRARIES := gmp
include $(BUILD_SHARED_LIBRARY)
```

Unlike the MCL library with GMP, it is needed to handle the passage of variables from Java to C, so it is more difficult to use in an Android project. That is why it is only used for the most demanding operation of modular exponentiation in this application. Listing 3.3 shows how such a function can be implemented in C so

it can be called through JNI from Java. The function must then be specified in the Java class that uses it as: `public static native String modPowC(String a, String b, String mod);`. Line 1 of Listing 3.3 shows the specific naming the function must have in order to be callable from the Java class. Lines 7-12 show how the strings from Java are passed to the `mpz_t` type of GMP. Line 16 is the operation itself, and lines 19-20 show how the result is passed back to Java.

Listing 3.2: Passing the Android.mk to Gradle

```
1 externalNativeBuild {
2     ndkBuild {
3         path 'src/main/jniLibs/Android.mk'
4     }
5 }
```

Listing 3.3: ModPow function in C for Java

```
1 extern "C" jstring
    Java_cz_vut_feec_xklaso00_groupsignature_cryptocore_
    NIZKPKFunctions_modPowC(JNIEnv *env, jclass thiz,
    jstring jst_a, jstring jst_b,jstring mod) {
2     mpz_t bn_a, bn_b, bn_mul, bn_mod;
3     const char *cc_a, *cc_b, *cc_mod;
4     jstring jst_mul;
5     char *c_mul;
6     mpz_inits(bn_a, bn_b, bn_mul,bn_mod, NULL);
7     cc_a = env->GetStringUTFChars(jst_a, NULL);
8     mpz_set_str(bn_a, cc_a, 10);
9     cc_b = env->GetStringUTFChars(jst_b,NULL);
10    mpz_set_str(bn_b, cc_b, 10);
11    cc_mod = env->GetStringUTFChars(mod,NULL);
12    mpz_set_str(bn_mod, cc_mod, 10);
13    cc_a=NULL;
14    cc_b=NULL;
15    cc_mod=NULL;
16    mpz_powm(bn_mul, bn_a, bn_b,bn_mod);
17    c_mul = mpz_get_str(NULL, 10, bn_mul);
18    mpz_clears(bn_mul, bn_a, bn_b,bn_mod, NULL);
19    jst_mul = env->NewStringUTF( c_mul);
20    return jst_mul;
21 }
```

Using GMP in Java on PC

It is possible to create a library that uses GMP and can be called from Java in the desktop environment too. This is usually called wrapping a library. Such a library was created mainly to support Linux systems since GMP is faster on Linux. It can be found in the thesis files in the `GroupSig_app` folder in the file called `gmp_forJava_linux.cpp`. The library implements operations of modular exponentiation and finding the next prime after a given number. In order to use GMP with Java, JNI must be installed on the system as well as the GMP library. This wrapper library must be compiled on the targeted system as shown in Listing 3.4 and then saved to a location such as `/usr/lib`. The library can also be compiled for Windows and used, however, it is slower than the BigInteger library, so it is not recommended.

Listing 3.4: Compile command for library using GMP in Java for Linux

```
1 g++ -fPIC -I /usr/lib/jvm/default-java/include/ -I /usr/
    lib/jvm/default-java/include/linux/ -shared -o
    libgmp_forJava.so gmp_forJava_linux.cpp -lgmp -lgmpxx
```

3.2.2 Implementation of NIZKPK

The two-party computation is based on the Paillier cryptosystem, so the first step was creating a working Paillier in Java, which was then modified to suit the NIZKPK protocol. Since modifications to the computations are needed, it was not possible to just use a finished library of Paillier and call its functions. The main classes in the package that deals with the NIZKPK implementation are: `PaillierKeyPair`, `PaillierPublicKey`, `PaillierPrivateKey` and `NIZKPKFunctions`. There are also other classes that mostly serve for the serialization of the variables used in the computations. The State diagram in Figure 3.1, shows the calls of the NIZKPK functions in a running protocol. It is shown when is each function called, and what function it passes data to.

The setup of the NIZKPK protocol is done in the class called `PaillierKeyPair` and is only run in the PC manager's application. The constructor of this class basically runs the setup of the NIZKPK protocol, it takes an integer value of desired bit size of n as a parameter (4561 in this implementation), and an instance of class `GothGroup` that holds the parameters of the gothic group (as these should not be generated by the manager). Firstly prime numbers p and q are generated with the help of the SecureRandom class for random seed. After this n is computed as $p \cdot q$, n^2 is also computed. Next secret value λ is computed as the least common multiple of *p-1 and q-1*. After this, it is needed to find a generator g in the set $\mathbb{Z}^*_{n^2}$, and make sure the modular multiplicative inverse μ exists. This can be done by generating a

random number from $\mathbb{Z}_n^*$, raising it to n in modulo n^2 (getting a possible generator), and checking if $g^\lambda \, mod \, n^2 = 1$, if this equation holds, we have found a generator. After this public values $n, g, n^2, \mathfrak{n}, \mathfrak{h}, \mathfrak{g}$, needed for computations on both sides are saved to a `PaillierPublicKey` object, while secret value λ alongside with values μ, n, n^2 are saved to a `PaillierPrivateKey` object. Java's `Serializable` interface can then be used to serialize the `PaillierPublicKey` object and send it through a communication channel, but it is important that this class is the same and located in the same package in the other application.

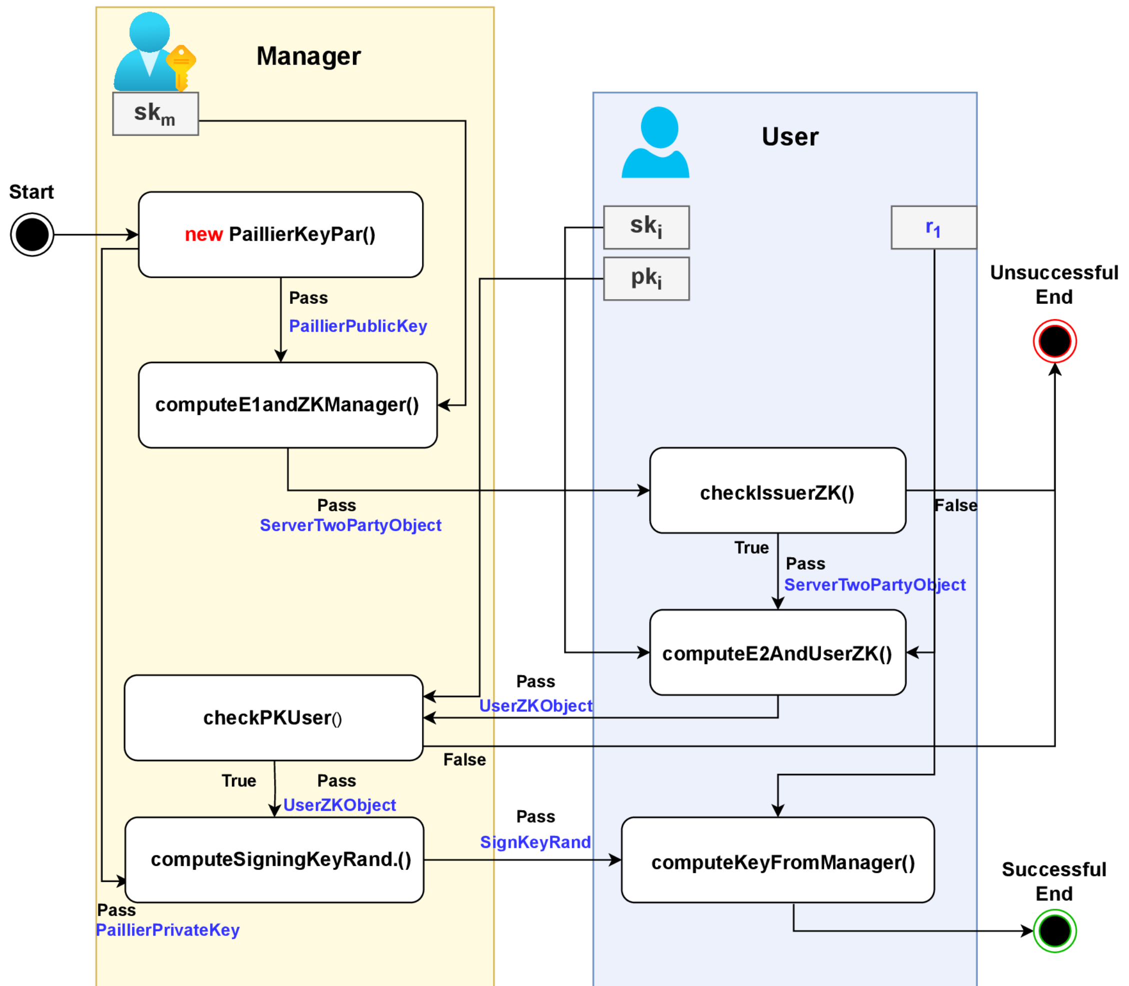


Fig. 3.1: State diagram of the NIZKPK functions calls.

The class `NIZKPKFunctions` contains all the functions needed during the run of the NIZKPK protocol. The methods are all defined as static. It consists of functions for both the manager and the user. There are many functions in this class used for the computations, so only the main ones that are then called from the outside are described.

The first function called after the setup is `computeE1andZKManager`. It has three arguments, an instance of `PaillierKeyPair`, the manager's private key sk_i, and the ID of the group. The function then computes values of $e_1, \mathfrak{c}$, and the PK_m with the use of other functions of this class. The values needed by the user to check the PK_m and compute e_2 alongside the public parameters in the `PaillierKeyPair` are saved to an instance of a `ServerTwoPartyObject` that can be then serialized and sent to the user. This function can be pre-computed, as no input from the user is needed.

The second function that is called during the running protocol is the `check-IssuerZK` by the user. It takes the parameters outputted by the previous function and checks the validity of the PK_m. It returns true for valid proof and false if the verification of the proof fails.

The next function called is `computeE2AndUserZK` on the user's side. This function's parameters are the public parameters saved in `PaillierPublicKey`, n of the curve used in the group signature later, the user's private and public keys, e_1, the user's ID, and a random number r_1 that is used in this computation and later to obtain the key used for signing. The reason this r_1 is not generated in this function is that the user should store it, so it can be used in a later function. The function computes $e_2, \mathfrak{c}'$, and the PK_i, it returns an instance of `UserZKObject` class with the values needed for the other manager's computations in it.

The manager can then use the `checkPKUser` function to check the validity of the user's PK_i, and if the proof is valid function `computeSigningKeyRandomized` can be called. This function computes the value of $x = (sk_m + sk_i) \cdot r_1$ and then returns σ_i. The user then only needs to run `computeKeyFromManager` function with parameters σ_i and the r_1 to obtain the value of δ_i used in the signature.

The use of GMP as an alternative is done by calling a specific function to perform the operation of modular exponentiation, as it is the most demanding operation during the protocol on the user's side. This function is called `myModPow` and if the use of GMP is disabled by a static boolean variable it will just call the standard `modPow` function from the BigInteger library. But if GMP use is enabled it converts the BigInteger values to Strings and passes it to a native C function `modPowC`. Similarly, on the manager's side, GMP can be used for generating the primes.

3.2.3 Implementation of the group signature algorithms

The functions used for signing and verification of the group signature are located in the class called `GroupSignatureFunctions` in the same package. It consists of functions for the user and for the verifier. All the functions are defined as static, so they can be called without creating an instance of the class. These functions mostly compute with points on the elliptic curve, so parameters passed to the functions

are usually either of type `G1` or `G2` for points and of `Fr` for scalar numbers, so the MCL library is needed for these methods. It is also important to initialize the library with `Mcl.SystemInit(Mcl.BN254);` in the main function of the program that wants to use this package. The curve used in the implementation is the BN254 curve, however, it should be possible to change this curve with other curves supported by MCL. However, the MCL library does not provide functions to get the generators G1 and G2 or the order of the curve, so these getters were implemented separately. The values used to construct generator points G1 and G2 were taken from the implementation of the AMCL library. So in case of changing the curve, the getters for G1 G2, and N in the class must be changed as well.

In the whole application for hashing the messages and for hashing during the protocols the program uses SHA-256. When the hash is used with the elliptic curve, operation modulo n of the curve is then applied to the hash, this is done with BigIntegers as the type `Fr` cannot store bigger values than the order of the curve. That is also the reason why the generation of random numbers in done with BigIntegers and only then passed to the `Fr` type.

The function for signing with the group signature is called `computeGroupSignature`. As arguments it takes the hash of a message to sign, n of the curve, the computed signing value δ_i, the User's private key, and the group's ID. The function runs the algorithm described in Section 2.2 and saves the values needed for the verification to a `SignatureProof` object, this object is then returned by the function. The values must be saved to this object as `byte[]` because the types from the MCL library are not Serializable by Java's `Serializable` interface. Fortunately, these types can be easily converted to `byte[]` by calling the `serialize()` function.

The function used for validating the signature by the verifier is called `checkProof`, as parameters it takes the `SignatureProof` generated by the user, the hash of the message, and the public key of the group. The function first checks the equality of the pairings (as shown in Algorithm 17 in Section 2.2), if the pairings are not equal the function will return false, if they are equal it continues by computing the $\hat{t}$ and the hash e'. The computed hash is then compared with the hash from the client, if they are equal the signature is legit and the function will return true.

The basic verifying function does not include the revocation check, this has to be done after this verification by another function. The reason for that is that other implementations might choose different formats for saving revocation lists. The class provides function `checkSignatureWithPK` that checks the pairing $\mathbf{e}(\delta_i', \overline{pk}_i) = \mathbf{e}(\overline{\delta_i}, g_2)$ for given $\overline{pk}_i$ and values of $\delta_i', \overline{\delta_i}$ used in the signature. For the final implementation, this function is called for all the values of $\overline{pk}_i$ in the revocation list, if the function returns 0 for any of the entries in the revocation list, the user is revoked and the signature is not valid. This function is also used for opening the

signature. The manager can cycle through his map of user's IDs and values of $\overline{pk}_i$, if the function returns 0 for some of the users, he knows that is the user that signed the message.

3.3 Implementing communication between the devices

The communication channel that was used for this demonstration was NFC. The main devices used were a PC and an Android mobile phone. The PC uses a card reader to communicate with the phone, and the phone uses HCE to act as a card. The protocol used in the communication is APDU, as it is the main protocol for NFC that can be implemented on an Android device.

3.3.1 Terminal on PC

The Java PC application is able to communicate through NFC with a mobile device (a phone) using an external NFC card reader. The reader used during the creation of the implementation was the ACR1251U USB NFC reader. The APDU commands are sent from the `Terminal` class created in the main package of the application `groupsignature`. There is also a second class `Instruction` that is used to build the commands sent to the phone.

For communication Java's `smartcardio` library is used. It is a default Java library, so no external files are needed for this library. The first function called by the terminal each time when there is a need to establish a connection with the phone is called `InitializeConnection`. The command *Choose AID* sent by this function must have a specific format. The format of this command is shown in Figure 3.2. The AID is a 7-byte identifier of the mobile application, for this application the AID was chosen at random as F0203344886655. The Listing 3.5 shows how the initialization of the connection works in Java.

CLA	INS	P1	P2	Lc	DATA	Le
0x00	0xA4	0x04	0x00	0x07	AID	0x00

Fig. 3.2: Structure of the choose AID command.

The initialization function first checks if a terminal is connected, it then waits for a card to be present near the reader (line 8). It then connects with the card using a supported protocol and transmits the choose AID command. If the command was successfully received by the device the response from the device will be 0x90

0x00, usually the other possible response is 0x6A 0x82 which means the AID was not found, usually, this is because the application is either not installed or does not have the permission to use NFC. The return codes of the function give some feedback on what went wrong to the rest of the program. Unless this function returns 0 other parts of the program using NFC will not be executed.

Listing 3.5: The function to initialize connection with a mobile device

```
public int InitializeConnection(){
    try {
        TerminalFactory factory = TerminalFactory.getDefault();
        List<CardTerminal> terminals = null;
        terminals = factory.terminals().list();
        CardTerminal terminal = terminals.get(0);

        while (!terminal.isCardPresent());
        // Connect with the card, using the supported protocol
        card = terminal.connect("*");
        channel = card.getBasicChannel();
        //transmit the getAID command
        ResponseAPDU response1 = channel.transmit(new CommandAPDU(Instructions.getAID()));
        byte[] byteResponse1 = null;
        byteResponse1 = response1.getBytes();
        System.out.println("Card response for choose AID command: " + Instructions.bytesToHex(byteResponse1));
        if(Instructions.isEqual(byteResponse1, Instructions.getaOkay()))
            return 0; //successful
        else
            return -2; // return code -2, the choose AID was not successful, check application permission
    } catch (CardException e) {
        e.printStackTrace();
    }
    return -1; //something went wrong, check the terminal connection
}
```

The Terminal class then includes the functions for sending the data during the NIZKPK protocol. The communication during the protocol was split into two parts, as the protocol is long and the computation on the mobile device can take multiple seconds and it would be impractical to hold the device near a reader for that long. The first part includes sending the public Paillier parameters the e_1 and PK_m to the user's mobile device. After this step, the connection is closed until the mobile device verifies the PK_m, and computes e_2 and PK_i. After this computation, the connection is established again and these values are sent to the terminal. The terminal then checks the PK_i and computes σ_i that is then sent to the user and the connection is closed. The connection is not closed during the terminal's computation, since having more than two stages during the protocol would not be very convenient for the user.

The class also includes a function sendFileToSign used during the signing of the document. The function's parameters are the hash of the file to be signed and a boolean flag specifying if the signature should be checked before saving it or not.

The Instructions class contains functions for building the instructions as they change depending on the parameters generated by the application. So for most of the instructions, there is a specified header of the instruction and a function that takes the data as a parameter and builds the final instruction with the use of ByteArrayOutputStream. The class also includes functions for the comparison of byte arrays to help identify the APDU responses and functions for the conversion of byte arrays to Strings and vice versa, the bytesToHex function is taken from [50], the function for converting a hexadecimal string to a byte array is from [51].

3.3.2 Implementation of HCE on Android

HCE can be implemented on Android by creating a class that extends the HostApdu-Service class [52]. For the proper function of this class, there are also a few extra steps that must be done. The first one is modifying the AndroidManifest.xml file to acquire permission to use NFC and to register the class that extends HostApdu-Service as a service. Next the apduservice.xml file must be created that specifies the AID of the application. Contents of the files are depicted in Listings 3.6 and 3.7.

Listing 3.6: Modification to the AndroidManifest.xml

```
<manifest
...
 <uses-permission android:name="android.permission.NFC"/>
 <application
 ...
```

```
 <service android:name=".MyHostApduService" android:exported="true"
     android:permission="android.permission.BIND_NFC_SERVICE">
     <intent-filter>
         <action android:name="android.nfc.cardemulation.action.HOST_APDU_SERVICE"/>
         <category android:name="android.intent.category.DEFAULT"/>
     </intent-filter>
     <meta-data android:name="android.nfc.cardemulation.host_apdu_service"
     android:resource="@xml/apduservice"/>
</service>
...
</application>
</manifest>
```

Listing 3.7: Content of apduservice.xml file

```
<host-apdu-service xmlns:android="http://schemas.android.com/apk/res/android"
     android:description="@string/servicedesc"
     android:requireDeviceUnlock="false">
     <aid-group android:description="@string/aiddescription"
         android:category="other">
         <aid-filter android:name="F0203344886655" />
     </aid-group>
</host-apdu-service>
```

The class extending the service in this application is called `MyHostApduService`. It overrides the two main methods of the parent class this class extends. The first method is called `onStartCommand`, this method initializes the instance of the class. This application for example registers the `LocalBroadcastManager`. The second function is the `processCommandApdu` method. This method specifies the answers to the incoming instructions based on the instruction's CLA and INS bytes.

Passing the data inside the application

As complex computations are needed in order to construct the APDU answers during the NIZKPK or the signature algorithm the data from this class must be passed to the main class of the application. This is because this class might not have access to all the objects it needs in the computations and because we want to be able to refresh the GUI of the application during the communication. This can be done by using the `LocalBroadcastManager` [53] class.

This class can be used to safely send data through the application without leaking the data to other applications. This is done by creating local broadcasts that include Intents with data and registering receivers in the classes that the data are intended for. The function in this class for sending the data to the main class is called `SendBytesToMainAcc`. Its parameters are a String path that specifies what data are being transmitted and data represented by a byte array. The receiver is created as an inner class that extends the `BroadcastReceiver`. The receiver then has specified steps depending on the path variable received with the data. The receiver that listens for data from the `MyHostApduService` in the main class of the application then modifies the GUI and creates new threads that run the computations. The results of these computations are then passed back to the `MyHostApduService` and sent to the terminal. The receiver must then be registered in the class. The code that shows how to register such a receiver and how the `Receiver` class is used to react to messages is shown in Listing 3.8.

Listing 3.8: Creation of a receiver in the MyHostApduService class

```
IntentFilter messageFilter = new IntentFilter(Intent.
    ACTION_SEND);
MyHostApduService.Receiver messageReceiver = new
    MyHostApduService.Receiver();
HandlerThread handlerThread = new HandlerThread("ht");
handlerThread.start();
Looper looper = handlerThread.getLooper();
LocalBroadcastManager.getInstance(this).registerReceiver(
    messageReceiver, messageFilter,looper);
//definition of the Receiver class with reactions to
    different messages
public class Receiver extends BroadcastReceiver {
    @RequiresApi(api = Build.VERSION_CODES.KITKAT)
    @Override
    public void onReceive(Context context, Intent intent)
    {
```

```
12          if(intent.getStringExtra("path").equals("2")){
13              zkCommandBytes=intent.getByteArrayExtra("
    value");
14              ZkPartDone=true;
15          }
16          ... //other conditions here
17      }
18  }
19  //example of a broadcast
20  LocalBroadcastManager.getInstance(MyHostApduService.this)
    .sendBroadcastSync(messageIntent);
```

In Listing 3.8 the lines 1-6 are run in the `onStartCommand` function of the class. Similar registration must be done in other classes that want to be able to receive the broadcasts. On lines 8-18 is the specification of the `Receiver` class. The example reaction to the path with value "2" is used when the class acquires computed e_2 and ZK_i from within the main class. It saves the message in a local variable and changes the variable `ZkPartDone` to true so the class knows it can continue with the NFC communication and pass this data to the terminal. The final line 20 shows an example of sending an intent through the application.

3.4 File managing on the devices

The last important back-end building block of these two applications was managing the files needed for the system to work. On the PC side, these are files for storing the information about a created group (manager key, group public parameters, manager's user list, and revocation list). And in the Android application, this is mainly the storage of the user's private key and the computed group key δ_i. This part also includes a class for modifying and reading a PDF's meta-data.

3.4.1 Files in the PC application

The classes that deal with operations with files are in a separate package called `fileManaging`. The class that operates with the files is the `FileManagerClass`. Similarly as when sending the data through a communication channel, Java's `Serializable` interface is used while saving the files. This way we can create classes that hold multiple variables of various types and just serialize their instances into a file. This combination of `FileOutputStream` and `ObjectOutputStream` can be used to write an object to a file and a pair of `FileInputStream` and `ObjectInputStream`

can then be used to load this data back to an object. All the files are saved to a folder `files` in the working directory of the application.

The public parameters of the group are saved as a serialized object of class `FileOfGroup` this object holds information about the ID of the group and the manager's public key that is needed in the verify algorithm of the signature. The file name has a name format of `ID_group_public_key.ser`. The file can only be read back by the same class in the same package. The revocation list is saved in a similar way but no special class is created for saving it and it is just saved as a serialized `HashSet` of the revoked users' $\overline{pk_i}$ values. The file name's format is `ID_revoked_users.ser`.

Password protected manager file

For saving the private group parameters as the manager's private key and the `HashMap` with users' IDs and values of $\overline{pk_i}$ class called `FileOfManager` was created. The instance of this class is initialized with the creation of a group and then saved to a file each time a change to this class is made. The file-saving process of this file is not as simple as other files. As the information included in the `FileOfManager` object is sensitive the object is first encrypted with *Advanced Encryption Standard* (AES). This is done by introducing password protection for the managers.

Each manager must enter a password during the registration phase. This password is then salted and hashed with `PBKDF2WithHmacSHA512` algorithm provided by Java's `SecretKeyFactory`. The advantage to using this algorithm instead of simple SHA-512 is that it applies the hash function multiple times making it harder for an attacker to try to guess the password with a brute-force attack, and the salt gives it protection against a dictionary attack. Additionally, the password is hashed again using SHA-256 to generate an AES-256 key from the password. The hash function used in the key-derivation for AES must be different than the one used in the password check hash, as the hash used for password checking is saved to the file.

The saved file of the manager is then a two-dimensional byte array serialized to a file, the file name has a format of `managerID_keyEnc.ser`. The first array is the initialization vector used in AES, the second one is the encrypted serialized `FileOfManager` object, the third is the salted hash of the password and the final part is the salt of the password. When loading this file back to the program the manager must enter his password, it is first hashed with the salt from the file and checked if the new hash is the same as the hash in the file, if so the password is used to derive an AES key and the encrypted `FileOfManager` object is decrypted. The protection should be as strong as the password itself, of course for a weak password an attacker could guess it if he got hold of the file. But this way provides a good level of security

for the manager's file while still being practical. For a better understanding diagram in Figure 3.3 was created to show how the program works during the login phase and what data are saved in the file.

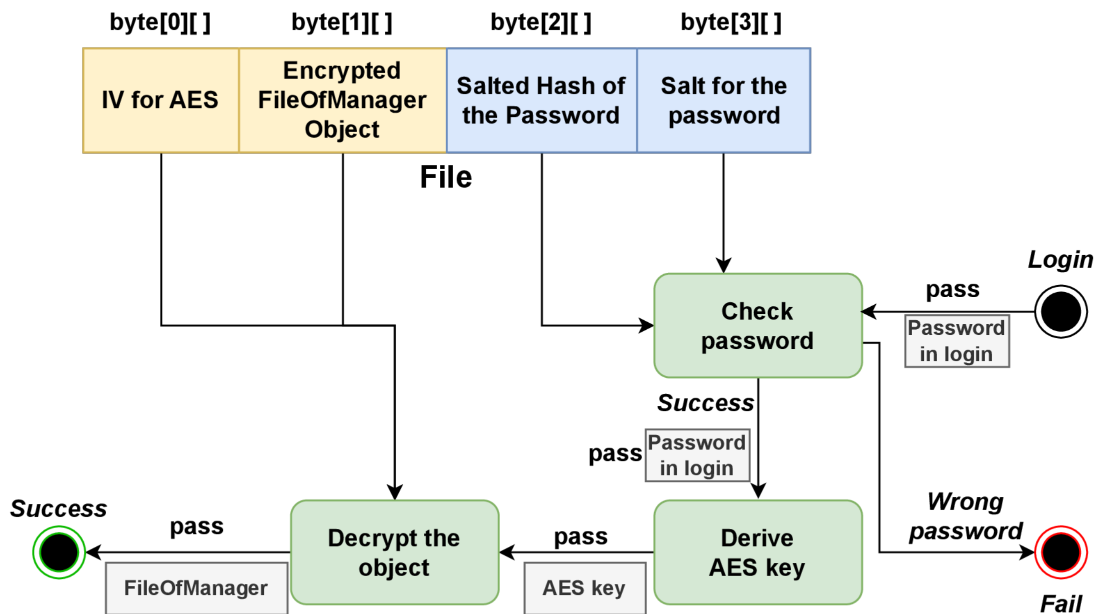


Fig. 3.3: Checking the password for the file.

Modifying and reading PDF's metadata

There is no easy way to read and modify the metadata of a PDF in Java, so an external library was used to make this part easier. The library ItextPdf [54] was used for this part. It is an external library and is free to use in open-source implementations. The class that does operations with PDFs in the application is called `PDFManager`. The code that is used to add the group signature to the metadata is shown in Listing 3.9.

Listing 3.9: Saving the group signature to the metadata

```
1 public static String saveSignatureToMetadata(String src,
    byte[] signature){
2     try {
3         Path path= Paths.get(src);
4         byte[] fileBytes= Files.readAllBytes(path);
5         PdfReader reader=new PdfReader(fileBytes);
6         PdfStamper stamper=new PdfStamper(reader,new
    FileOutputStream(src));
7         HashMap<String,String> info=reader.getInfo();
```

```
8            info.put("GroupSignature", Instructions.
    bytesToHex(signature));
9            stamper.setMoreInfo(info);
10           stamper.close();
11           reader.close();
12           return src;
13       } catch (Exception e) {
14           e.printStackTrace();
15           return null;
16       }
17   }
```

The function takes the path to the PDF file and the signature as arguments. The file is then read (lines 3-4) to an array because we cannot pass the file itself to the `PdfStamper` that modifies the metadata since it is not possible to read and write to the file at once. A `PdfReader` and `PdfStamper` objects are then initialized and a `HashMap` of the metadata is read (line 7). It is then possible to add entries to this `HashMap` and save it back to the file. First, a different approach was taken, where a temporary file was created as a copy with the signature and then the original file was deleted. But this approach had some problems as the library would not release the original file, and it could not be deleted, even though all data streams were closed.

A very similar approach as when saving was taken for the reading of the metadata. The `PdfReader` object can be used to extract the `HashMap` of the metadata from the file and get the signature from it.

As changing the metadata of the file changes the bytes of the file returned by the function `readAllBytes`, the bytes of the file given to the hash function during the signature had to be obtained differently. For this purpose the function `getContentBytesOfPDF` was created. It uses the `PdfReader` to read the content of all the pages of the PDF to a byte array. This array is then returned by this function. Since the contents of the file will not change with the modification to the metadata, the hash will be the same before and after the signature is added to the file.

3.4.2 Storing application data on an Android device

On the Android mobile device, it is needed to save the private key and ID of the user as well as the value δ_i used in signing and the ID of the group. One of the ways to store the data easily is by using Android's `SharedPreferences` API. With it, it is possible to save application data to an application-private file that is not accessible from a different application. The values are saved in the `String` format

as this API only supports a few types to be saved. Each value is then identified by a `String` key so it can be extracted from the file.

The class created for saving the values to the file is called `UserOperations`. It provides a function `generateUser` that is called when a new user is created. It generates his private key and ID and calls another function `saveUser` to save these values to the file. Separate functions were also created to save the group's ID and the value δ_i after the NIZKPK protocol is done. The saved values are then loaded with the function `loadUser` each time the application starts. In case this function fails (for example the application was just installed), a new user will be created.

3.5 Building the applications with GUI

In the previous sections, the main building blocks of the back end of the applications were described. The last step was to create a GUI and connect it with the back end.

3.5.1 GUI and final application on PC

The GUI of the PC application was created with the help of *Swing UI Designer*. This way each window has a Java class that handles the user's inputs and a `form` file that is used to help design the window. There are a total of six windows defined for the application and all the classes are located in a package called `gui`.

For communication between the GUI and the back end of the application, a new class was created. The class `ModelViewHandle` takes care of calling the back-end functions and handles their output, its functions then return values useful for the GUI, for example, if an action was successful. For some functions such as the NIZKPK protocol, the class's function creates new `SwingWorker` threads to execute the parts of the protocol. This is needed so the computations do not block the main thread that operates the GUI. It is also possible to update the GUI from within the function after the thread finishes its work. This class works in part with a class called `Server` that holds the data of the manager needed in the computations in the program. It also provides specific functions for generating the NIZKPK setup with the use of a predefined goth group, functions for adding the users to the group, their revocation, and for opening the messages.

The main window that opens when the application is run is defined in class `StartWindow`. It serves as a crossroad for different users of the application. The reason why all the applications for PC are started from this window is that it would be impractical to have a separate file to start each application. The opened window can be seen in Figure 3.4. The user of the application can generate a new manager account, where he will be asked to enter a password for the manager (in a new

window specified in `RegisterWindow` class). Next, there is an option to log in as the manager and open the manager part of the application. Here the user chooses a manager key file to load and must enter the password of this manager. The process of choosing the files from the file system is done with the help of Java's `JFileChooser` class. Next is the option to open the client application that allows a client to choose a PDF file from the computer's file system and sign it with the help of his mobile device with the saved key. Final is the verifier application that is used to verify the validity of signatures.

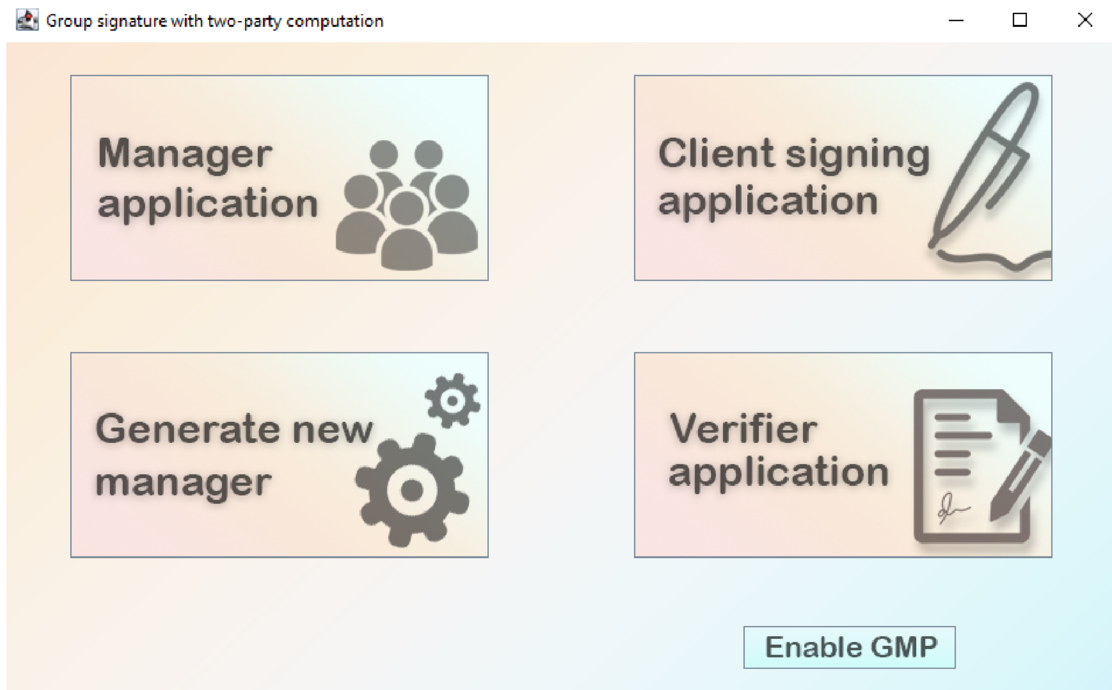


Fig. 3.4: The main window of the application.

The manager's application is shown in Figure 3.5. The GUI of this part of the application is defined in the `ManagerWindow` class. The manager has options to add users to the group, revoke users and open the signatures. He also can see the list of the users' IDs that are registered in his group. The add user option will start the NIZKPK protocol, first generating the parameters and then initializing the NFC communication with the mobile device. The user is also informed of the process in the text label under the button. For revoking users the manager must insert the ID of a user he wants to revoke, the user will then be added to the revocation list and will be shown as revoked in the field of registered users. To open the signature the manager must choose a signed PDF file from the file system, the open function is then run on it and if the user is part of the manager's group an ID of the user will be shown to the manager. The manager has also the option to choose if the setup

that generates before the NIZKPK protocol is computed each time or is generated only once and used for each user during that session.

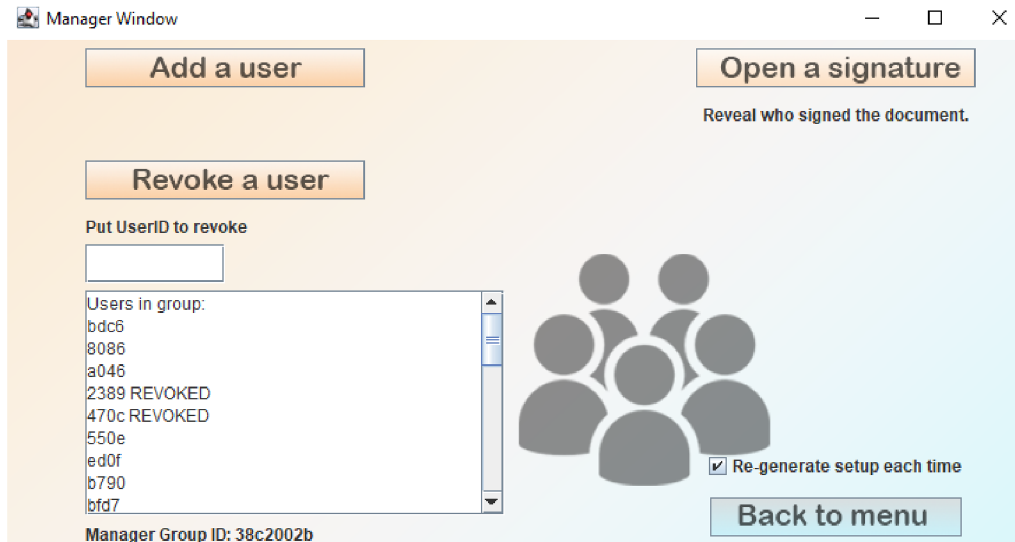


Fig. 3.5: The manager's application.

Fig. 3.6: The signer's application.

The signer's application is specified in the `UserWindow` class. The application only allows the user to choose a PDF file from the file system to sign. After choosing the file NFC communication will be initialized and the hash of the file is sent to the mobile device and then the signature received is saved to the file's metadata. The window is shown in Figure 3.6.

Lastly, the verifier's application shown in Figure 3.7 is very similar to the signer's one. It gives the verifier an option to choose a signed PDF file from the file system and check the validity of the signature. It is defined in the `VerifierWindow` class.

After the verification, the window informs the verifier of the validity of the signature and shows him the ID of the group that created the signature, if the signature was valid.

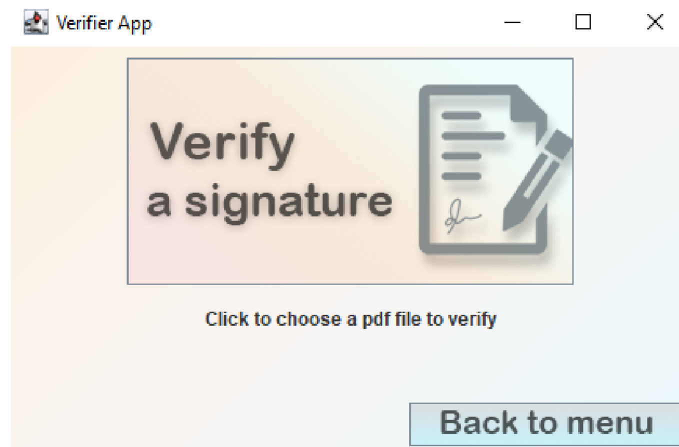


Fig. 3.7: The verifier's application.

3.5.2 GUI on Android

In the Android application for mobile phones, the GUI is defined in the `activity_main.xml` file. Android Studio provides this file for each project, so the file is created automatically. The functionality of the GUI is then provided by the `MainActivity` java class. How the application looks can be seen in Figure 3.8.

The GUI is much simpler than the PC application as this application does not need that much user input. The screen show to the user his ID and the ID of the group he is registered in, if there is one. The user can then delete his user data and generate new ones. He can also disable GMP support. This will move all the computations during the NIZKPK to BigInteger, this is not recommended on the mobile device as it will result in slower computations. The user is also informed about the progress of the protocol on the screen, Figure 3.8 shows the screen after the two-party protocol was just completed.

The computations during the running protocol are run on a different thread (using the `Runnable` interface) so that GUI can be updated during a running protocol using the `runOnUiThread` class. In the final application, during the run of the program, the `MyHostApduService` takes care of the NFC communication and passes it to the `MainActivity`, this class then takes care of creating different threads that then call functions from other back end classes, while handling the GUI. The computed values are then passed back to the NFC class if needed.

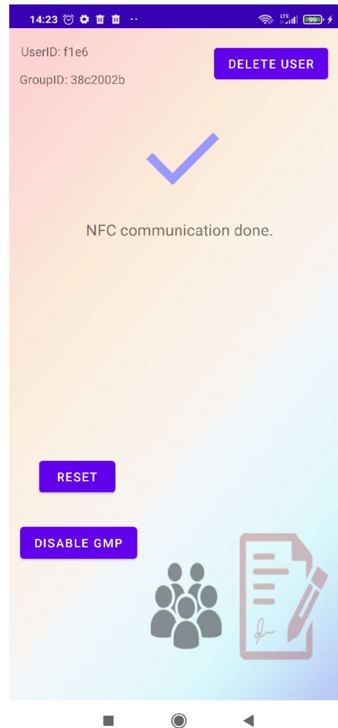


Fig. 3.8: GUI of the Android phone application.

3.6 Benchmarks of the implemented group signature protocol and the applications

This section is devoted to benchmarking the protocol on different devices. The benchmarks were performed with the final Java application for PC and the Android application for mobile phones. The devices used for the PC application were:

1. PC: HP Pavilion 15 with OS: Windows 10 Home 20H2, RAM: 16 GB, CPU: Intel i5-9300H - 4 cores 2.4-4.1 Ghz
2. VM running on the same PC with OS: Ubuntu 64-bit Linux (VMWare, RAM: 8 GB, CPU: 4 cores)
3. Raspberry Pi 4 Model B with OS: Raspbian, RAM: 2 GB, CPU: ARM Cortex-A72 - 4 cores 1.5 GHz

For the mobile application, it was:

1. Mobile phone Xiaomi Redmi Note 8 Pro with OS: Android 10, RAM: 6 GB, CPU: MediaTek Helio G90T 8 cores 2.05 GHz
2. Galaxy Watch 4 Classic with OS: Wear OS 3, RAM: 1,5 GB, CPU: Samsung Exynos W920 - 2 cores 1.18 GHz

The mobile phone is the main device for the implementation, as it was not aimed at smartwatches. However, it is possible to run it on an NFC-enabled smartwatch

with WearOS. The only part that has to be changed is the GUI. So the smartwatch is tested to see if the protocol could be used on it in practice. All the benchmarks were run 10 times and the result is the average of these times.

3.6.1 Benchamrks of the NIZKPK protocol

As the application can be run with GMP use or without, both cases were tested, although, from the benchmarks of the libraries, it is clear that on Windows BigInteger is faster, and on Android mobile phones and Linux GMP is. On the Raspberry Pi, only GMP was tested, as the implementation with BigInteger was too slow to even be used. Figure 3.9 shows the steps of the protocol with the communication for a better understanding of what was benchmarked. The first benchmarks are the setup and manager's precomputations of e_1 and PK_m, these are less important as these are pre-computed before the communication is established. Therefore this part is excluded from the benchmarks of the 2-party computation with the NFC overhead. The benchmarks of the Manager's pre-computation are shown in Figure 3.10.

The setup part, shown in the graph, consists of loading the goth group from a file and generating the other Paillier parameters. The most demanding operation (about 80% of the time) was the generation of the prime numbers p, q for modulus n. This operation takes a different amount of time each time depending on how fast the prime numbers can be found since this is very random. The operation that takes most of the rest of the time in the setup is the generation of generator g.

The manager's part of the computation can also be pre-computed before communication with a user is needed. The computation of e_1 is much faster than the computation of the manager's proof of knowledge PK_m. This is logical as PK_m includes more operations of modular exponentiations with big numbers. Note that computation of $\mathfrak{c}$ is also included in the time of PK_m.

From the graph shown in Figure 3.10, it is obvious that out of all the tested devices and configurations, the fastest one in this part is Ubuntu Linux with the use of GMP for modular exponentiation and prime generation. Even though Ubuntu is run on a virtual machine, in Java the computation times are very similar to the Windows installation running on the machine directly. The effect of using GMP in the protocol on Windows is a slower run of the algorithms, mainly in the setup. Also, the GMP on Windows was not very stable as sometimes after a few runs of the algorithm the library would get slower and a restart of the machine was needed, although the library used for the modular exponentiation and prime generation should not include any memory leaks. This might be due to some Windows-specific behavior as on Linux this would not happen using the same library.

The second plot shown in Figure 3.11 displays the computation time on the

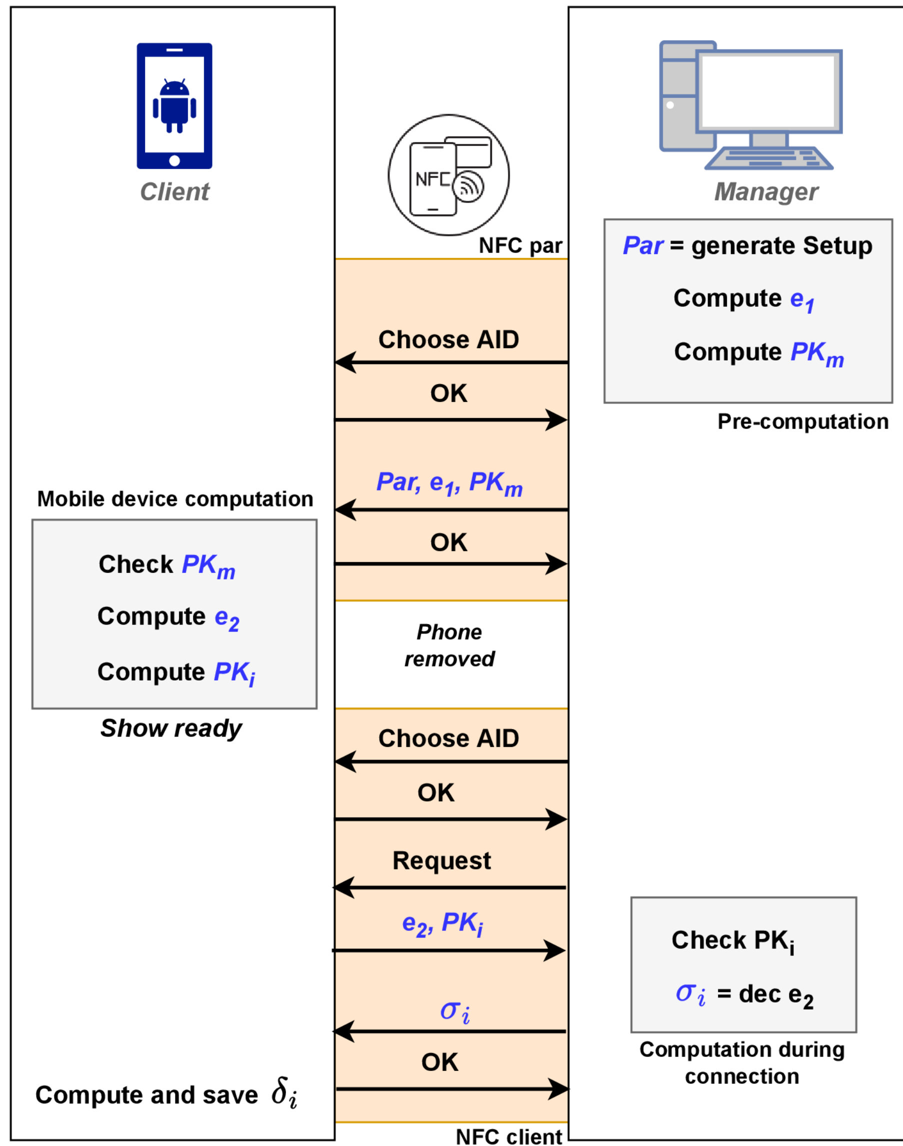


Fig. 3.9: NIZKPK protocol with NFC communication.

manager's side. It also includes times of how long the data transfer through NFC takes. The first algorithm *NFC par* stands for how long it takes to send the public parameters along with e_1 and PK_m to the client and to get confirmation from the client that the message was received. The second algorithm *NFC client* is the time it takes to transfer data, including e_2 and PK_i, from the client to the manager. Since practically the communication is split into two parts, where the mobile computation is done without the NFC connection, the time that the device must be held on the reader the second time is given by the sum of times of *NFC client*, *Check* PK_i and *dec* e_2. Also, one more command is sent where the deciphered e_2 (σ_i) is sent to the client, but as this is a small message it adds only about 50 ms. Because of the need

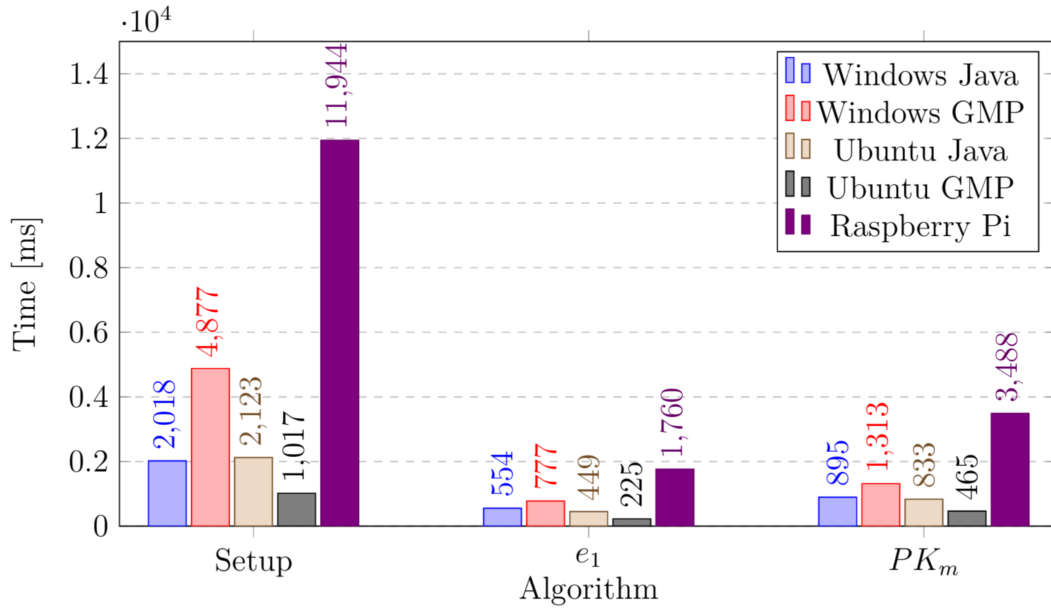


Fig. 3.10: Times of manager's pre-computation on different devices.

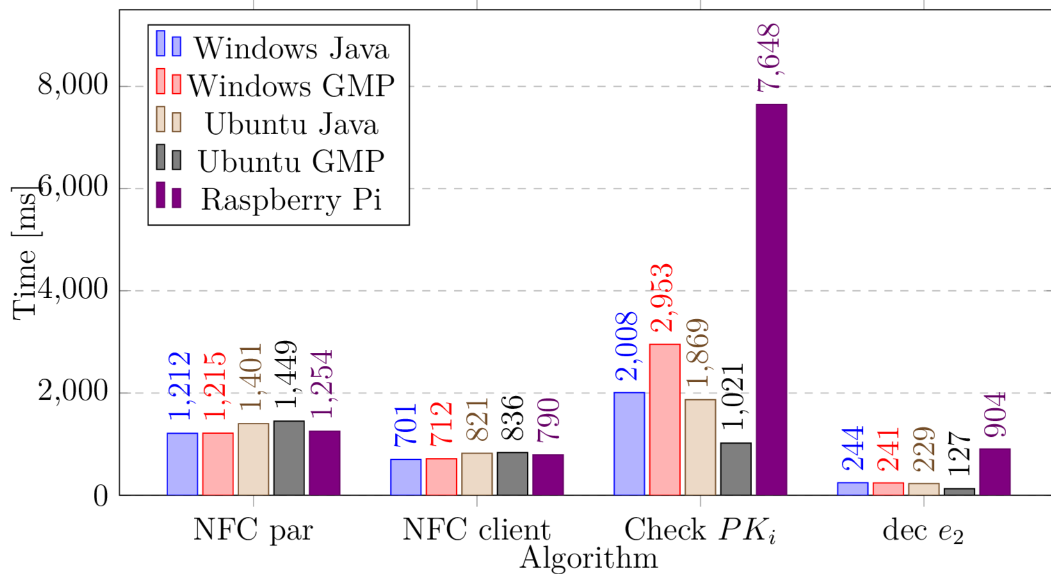


Fig. 3.11: Times of the NFC transfers and manager's computation during the active protocol on different devices.

to keep the connection up for this time the only practical devices for this are either the Windows machine with the Java implementation or a VM ideally with GMP, the Raspberry Pi is not very practical for the NIZKPK protocol.

The reason why the NFC communication takes that long, as seen in the graph,

is that the program sends a big amount of data. In the *NFC par*, about 10 000 bytes are sent, and in the second part *NFC client* about 5 600 bytes are sent from the client. So this information shows that the effective speed of the NFC was only about 8 000 bytes per second. So in the future, the applications could be modified to also support sending the data through an IP network. The speed of the NFC is very similar for all tested devices, as it should be mainly given by the reader. Only when ran from a VM it was a little slower. The speed was mostly stable but in rare cases, the transfer could be slowed to about double the normal time.

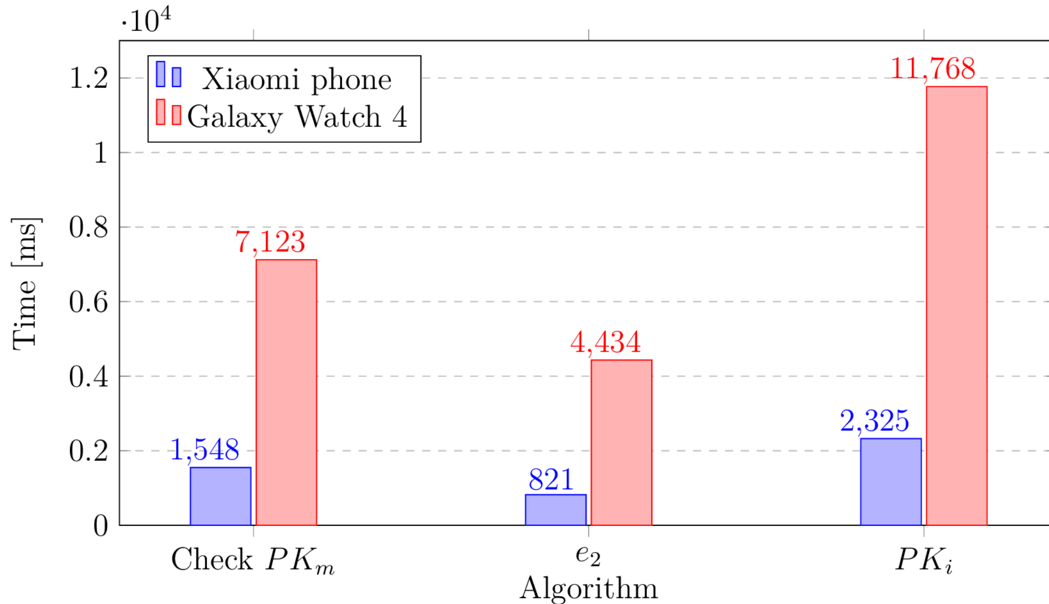


Fig. 3.12: Times of the mobile device's computations during the NIZKPK protocol for a phone and a smartwatch.

The plot shown in Figure 3.12 shows the computation times of the functions ran on a mobile device of the user. These are run when the NFC communication is not active and only include the computation itself, as the NFC speed can be calculated on the terminal side. The most demanding algorithm is the computation of PK_i, which also includes the computation of $\mathfrak{c}_1$. This is because it is the most complex algorithm. The smartwatch is much slower than the phone and as the total time of computation is about 23 seconds, the smartwatch is not very suitable for this protocol. The smartwatch can also sometimes have problems keeping the NFC connection so there had to be added another initializing of the communication after the PC computes the σ_i. The watch can also stop the NFC service of the application when the screen turns off. So it is not recommended to use for this application. For the phone, the computations take under 5 seconds, which is the time that the device will be practically removed from an NFC reader. The time of the final computation

when receiving σ_i is not included in the times, as it is a very fast computation of one multiplication on an elliptic curve and takes less than 1 ms on the phone, so it does not make sense to show in in the graph.

The final plot in Figure 3.13 shows how long on average the protocol takes for different combinations of devices. One more factor that influences the final time is the time it takes for the user to put the device back on the reader after he is informed that the mobile computation is done. This action adds about a second to the total time. As mentioned before this does not include the pre-computation done by the manager. Looking at the picture of the running protocol in Figure 3.9, this time is taken from the first *Choose AID* command to the last *OK* response from the phone. For this part only the faster implementations were tested, meaning pure Java on Windows and GMP support on Linux.

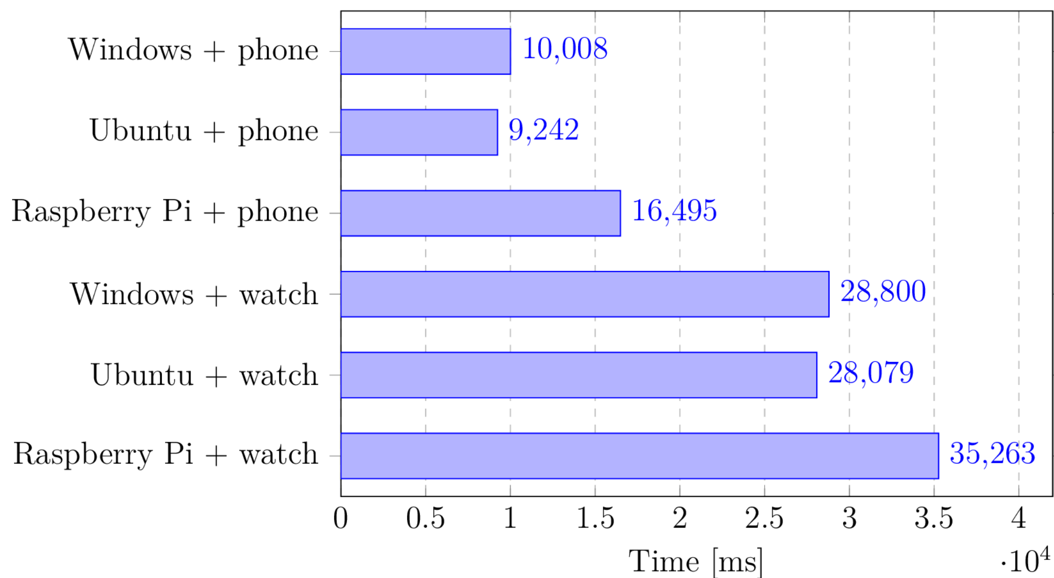


Fig. 3.13: Total time of the interactive part of the protocol for different combinations of devices.

The graph shows that the fastest combination of the devices is Linux which uses GMP with a mobile phone, closely followed by the combination of Windows with BigInteger and a phone. Even though the Linux combination is only faster by less than 10 %, this time is saved in the computation that is run during an active NFC connection, so the user has to hold the phone on the reader for a shorter period of time. Other combinations are not recommended as the Raspberry Pi is very slow in the computations and the watch also. The graph is the time needed to add one user to the group, so if the manager will use the same setup for each other user this time will be the same.

From the benchmarks done in this section, it is evident that the NIZKPK protocol is not very fast. This is because the protocol works in a big modulus and a lot of data has to be exchanged between the parties. But this protocol is only run once for each user so it is not such an obstacle for using this group signature with two-party computation.

3.6.2 Benchmarks of the group signature

In this section, the speed of the protocol used for the creation of the signature will be shown. This includes the signature with the NFC overhead and the verification on different devices and for different amounts of revoked users.

The first graph shown in Figure 3.14 displays how long different parts of the signing algorithm take. *Sign* here stands for computation of the signature in the mobile application, *NFC for Sign* is the time it takes to send the hash of the file to the mobile device and the signature back to the PC. The *Load file* function here stands for loading the PDF from the file system and hashing it. *Verify* stands for the verification of the signature with 0 revoked users.

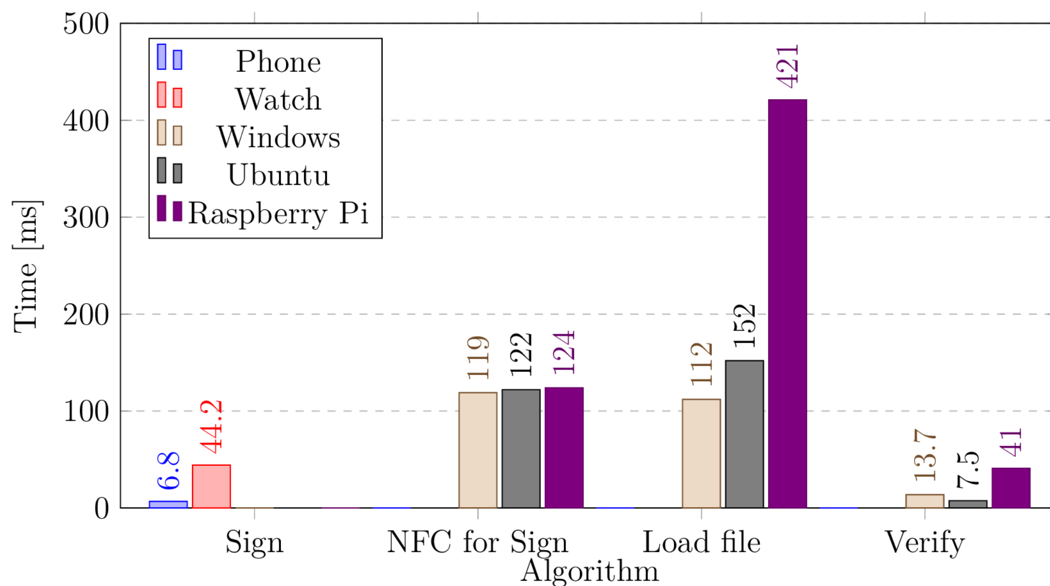


Fig. 3.14: Time consumption during the signing algorithm in the application.

The time the NFC communication has to be active, during the signing phase, is given by the sum of *Sign* and *NFC for sign*, as the hashing and loading of the file, as well as saving the signature after is done in the background of the PC application. Both the phone and the smartwatch give acceptable times in the signing algorithm, but the phone is still much faster than the watch.

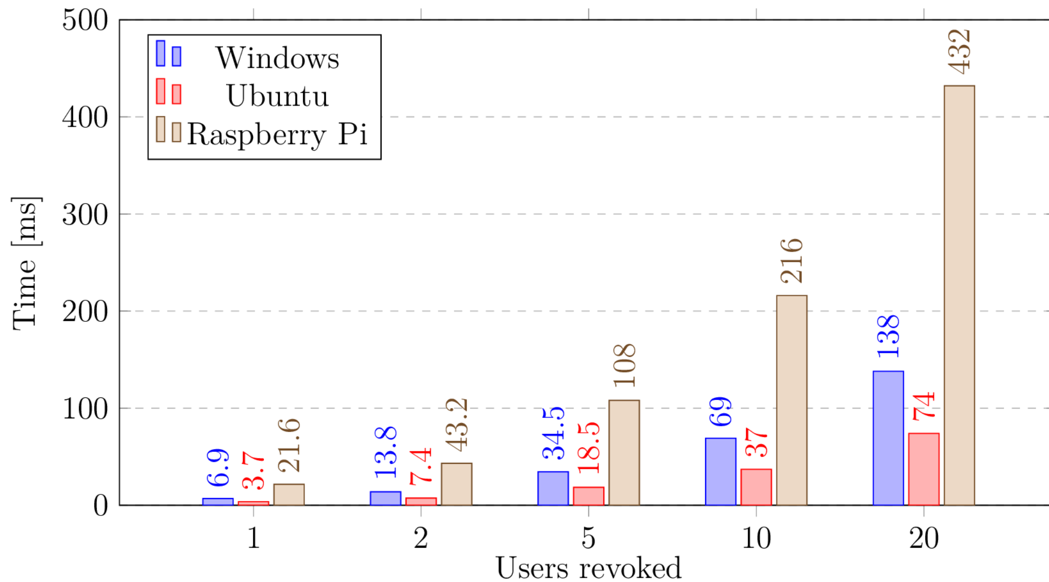


Fig. 3.15: Time needed to check the revocation depending on the number of revoked users in the group.

The time it takes to verify a signature in the PC application is given by the sum of times it takes to load and hash the file (algorithm *Load File*), verify the signature itself (algorithm *Verify*), and run the revocation check. The plot in Figure 3.15 shows how long the revocation check takes depending on the number of revoked users in the group. The verification algorithm itself consumes just a small part of the time it takes to verify a signature of a file. Most of the time is either taken by the loading and hashing of the file or by checking the revocation (depending on the number of revoked users). The time needed to check the revocation is given by how long on average it takes to run the pairing check $\mathbf{e}(\delta_i', \overline{pk}_i) = \mathbf{e}(\overline{\delta_i}, g_2)$ and the number of revoked users. The same applies to the time it takes to open the signature, but there it is given by the number of users in the group. All the devices running the PC application were capable of performing the verification in an acceptable amount of time, but for systems with a big amount of revoked users in a group the Raspberry Pi might be too slow.

From all the benchmarks performed, the best combination for speed is a Linux system using the GMP bindings with a mobile phone. But the Windows application without GMP is still a viable option mainly for its portability compared to Linux. It should also be pointed out that the Linux version that uses GMP should only be used if a side-channel attack is not a concern as the GMP function `mod_pow` is susceptible to it.

3.7 Implementing the NIZKPK into KVAC

The implementation of the blind issuance into KVAC with adding the NIZKPK was done using C++, as the existing code for KVAC [55] is written in C and there was also an existing NIZKPK implementation from [42] in C++, although without the proofs of knowledge. The main libraries this implementation uses is GMP for modular arithmetics and micro-ecc for elliptic curve cryptography used in KVAC. The implementation mainly uses the *secp256_r1* curve. Note that the protocol was not implemented with communication overhead but more as proof of work and to be compared with the original KVAC. Therefore for a real use with an overhead, some modifications might be needed. The full project can be downloaded from GitHub[1], as the Visual Studio project is too big to be put in the attached source files. But the thesis files include the source files of this implementation with a compilation command for Linux.

One of the main challenges of merging these two implementations was to pass the data between the KVAC part and the NIZKPK part, as the micro-ECC uses different specific types to store values (`uECC_word`). To solve this, conversion through `uint8_t` was performed to pass to GMP-specific type `mpz_t`. Then also the KVAC implementation was made for Arduino, so the library used for hashing had to be changed to a different implementation [56], as it used an Arduino-specific library. The NIZKPK C++ implementation used a Linux-specific random number generator so in order to be usable on different systems an alternative generator was made.

In order to implement PKs in this implementation, some changes were made to the structs defined for the computations. Then the PKs computations were written similarly as during the previous implementation in Java, just now purely using GMP. The original functions had to be also changed to accept as parameters the private key of the user and the value d computed from the private value of the issuer that acts as his private key in the computation.

The KVAC algorithm had to be changed too to work with the blind issuance of NIZKPK. First, the issue algorithm was split to work as shown in Section 2.3 in Algorithm 18. Now before the NIZKPK protocol a function called `SignGFirstHalf` is called. This function computes the sum d that is then needed for the two-party computation. After the NIZKPK protocol is run, the value of $k = (d + sk_i) \cdot r_1$ is returned along with the r_1 value for the client. Then the next two new functions are run, it is `SignGSecondHalf` that computes the value of σ^* and `signSigma` that computes the list of $\sigma^*_{x_1}, ..., \sigma^*_{x_n}$. After that, the user removes the r_1 from all the $\sigma*$ values of the credential.

Finally, the modified `Show` and `ShowVerify` algorithms were implemented. The

[1]KVAC project repository `https://github.com/xklaso00/KVACBlindIssue`

function `declareModified` is a modified function of the original `Show` function, it takes more parameters than the original one, as the user's private key is needed in this computation, and more values have to be returned compared to the original computation. The second function is `verifyModified`, which just takes one more parameter (S_r) compared to the original one.

The code is then tested and benchmarked from the function called `setup`. It is a function from the original KVAC implementation that initializes all the variables needed in the algorithms. So the initialization part was kept and the changes to use the NIZKPK during the `Issue` phase were made. The program then also runs the original KVAC for comparison.

3.7.1 Comparing the modified KVAC with the original version

This section deals with the speed comparison of the original KVAC algorithm and the modified version supporting blind issuance. The algorithms were tested only locally on multiple devices without the communication overhead. The devices used were the same as in Section 3.6.

The main disadvantage is the slow issue algorithm using the NIZKPK protocol. The computation time of this algorithm was the same as with GMP shown in Section 3.6 in Figures 3.10, 3.11, and 3.12 on mobile devices. So that means the issue algorithm with implemented NFC overhead would take about 10 s excluding the pre-computations on the PC using the mobile phone as the second device. Compared to the original issue which takes about 7 ms without the overhead on a PC. But the same as in the group signature this algorithm is only run once for each user.

For the KVAC algorithm what is more interesting is the speed of the `Show` and `ShowVerify` algorithms, as these would be run regularly in a real use-case. Although the whole algorithm was run on each device, the PC, VM, and Raspberry Pi were mainly benchmarked for the verifier part, while the Android phone and smartwatch were benchmarked for the user part. The tests were conducted for 10 issued attributes and 2 shown. With the rising number of shown attributes, the times of these algorithms will be shorter as shown in [55], but to show that was not the point of these benchmarks. The graph in Figure 3.16 shows the computation time of the `Show` algorithms on mobile devices and the graph in Figure 3.17 shows the speed of the verification algorithms.

In the graph in Figure 3.16 it can be seen that the modification to the Show algorithm introduced about a 10 % increase in computation time, which is still very fast. The modification to the Verify algorithm did not have a big effect on the computation time as seen in Figure 3.17, which is mainly because the change to this algorithm was minimal. Unlike the GMP library, the micro-ecc library is actually

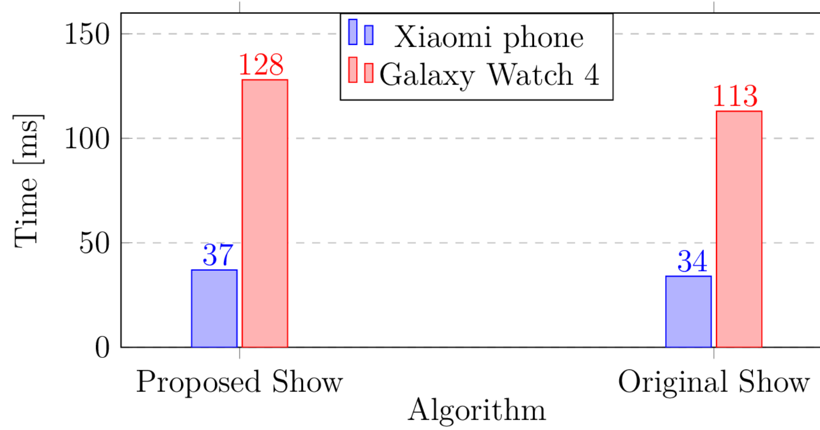


Fig. 3.16: Time it takes to compute the Show algorithm on mobile devices.

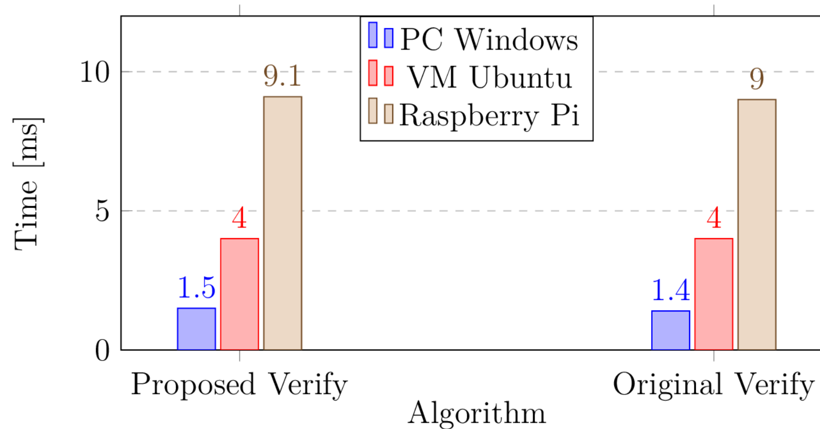


Fig. 3.17: Time it takes to compute the Verify algorithm.

faster on the device itself than on the Linux VM.

To sum up, regarding the KVAC modification implementing blind issuance, it must be said that the demanding NIZKPK protocol could not be run on a device like a smart card, so the modified protocol is aimed at smart devices such as smartphones. The modification did not have a big effect on the computational times of the `Show` and `ShowVerify` algorithms, meaning they still remain competitive.

Conclusion

The main aim of this thesis is to create an implementation of a group signing scheme that uses a two-party computation to hide the user's secret from the manager of his group. The implementation of the scheme was created and was used to generate signatures of documents using the implemented protocol.

The implementation was created to be run on multiple devices using mainly Java. To make the implementation as efficient as possible, computation speed tests of selected C and Java libraries on Android and a Windows PC were conducted. Because of this, the signature algorithm uses the C Mcl library which is much faster than a similar Java library providing bilinear pairings. Some of the demanding computations were done using C library GMP instead of BigInteger to save time on some devices.

An Android application is created for the members of a group. A user uses an Android mobile phone to take part in the two-party computation with the manager and to store the secret key deployed in the signature. A PC application is also created. This application is split into three smaller applications. The first one is the manager's application which allows the manager to manage his group by adding users, revoking their membership, or opening the signatures generated by members of his group. This application uses communication through NFC to perform the two-party computation with the user's mobile device. The second PC application is the client's signing application. It allows a member of a group to choose a PDF file from the system to sign and sends its hash through NFC to his mobile device, which then creates the signature and sends it back to the PC to save it to the file's metadata. The final PC application is for the verifier. It can be used to verify a signature of a signed file and gives the verifier only the knowledge of what group signed the message, while also checking if the signer was not revoked, without revealing the signer's identity.

In this thesis, it is also shown how the NIZKPK protocol can be used to implement blind issuance into the KVAC scheme. This was done by extending the existing implementations of KVAC and NIZKPK in C++ and merging them together. The article about the KVAC modification was also presented on STUDENT EEICT 2023 [57].

The NIZKPK protocol used in the two-party computation for the group signature was benchmarked on multiple combinations of devices, as well as the group signature, this also included the communication through NFC. The KVAC modification was then compared to the original implementation without blind issuance. These benchmarks showed that the two-party computation used in both protocols is rather slow because it computes in big modulus, but it has to be run only once for

each user so it is not such a big problem. With the combination of a standard laptop and a smartphone, this two-party protocol took about 10 s including the NFC overhead, which was split into two parts to make it more convenient for the user. This time did not include the manager's pre-computation that he can compute without interaction with the user. The group signature itself was very fast, taking about 6.8 ms on a phone and around 125 ms with the NFC overhead. The verification algorithm ran on the PC took about 14 ms itself and around 125 ms including time to load the file, this time would then raise by another 7 ms per each revoked user in the group.

In the KVAC scheme, the focus during the benchmarks was on the Show and Verify functions, as the NIZKPK protocol was the same as used in the group signature and its speed was very similar. The Show algorithm was slowed by about 10 % with the blind issuance modification while the Verify algorithm remained almost as fast as before. The main disadvantage of the blind issuance was that the demanding two-party computation would dramatically slow the Issue algorithm of the protocol, making it not usable for smartcards.

For future work, another communication overhead could be added to the group signature application to be used instead of NFC to decrease the time it takes to compute the two-party protocol. The ability to sign a file in the mobile device directly could also be implemented. A chain of trust could also be created above the groups to make the application usable in a bigger scenario.

Bibliography

[1] CHAUM D, HEYST EV. Group signatures. *In Workshop on the Theory and Application of Cryptographic Techniques* 1999-04-08 [cit. 2022-12-09] (pp. 257-265). Springer, Berlin, Heidelberg.

[2] SUDARSONO A, NAKANISHI T, NOGAMI Y, FUNABIKI N. Anonymous IEEE802. 1X authentication system using group signatures. *Information and Media Technologies.* 2010-05 [cit. 2022-12-09] (pp. 751-764).

[3] BELENKIY, M.; CAMENISCH, J.; CHASE, M.; KOHLWEISS, M.; LYSYANSKAYA, A.; SHACHAM, H. Randomizable Proofs and Delegatable Anonymous Credentials. *In Annual International Cryptology Conference* [online] 2009-08-16 [cit. 2022-12-09] (pp. 108-125). Springer, Berlin, Heidelberg. Available at: <https://eprint.iacr.org/2008/428.pdf>.

[4] PAILLIER, P. Public-key cryptosystems based on composite degree residuosity classes. *In International conference on the theory and applications of cryptographic techniques.* 1999-05-02 [cit. 2022-12-10]. (pp. 223-238). Springer, Berlin, Heidelberg.

[5] Boneh D, Boyen X, Shacham H. Short group signatures. *In Annual international cryptology conference* [online]. 2004-08-15 [cit. 2022-11-21]. Springer, Berlin, Heidelberg. Available at: <https://crypto.stanford.edu/~dabo/pubs/papers/groupsigs.pdf>.

[6] CAMENISCH, J.; DRIJVERS, M.; DZURENDA, P.; HAJNÝ, J. Fast Keyed-Verification Anonymous Credentials on Standard Smart Cards. *In ICT Systems Security and Privacy Protection* [online]. 2019 [cit. 2023-05-05] (pp 286-298) Springer Nature Switzerland. Available at: <https://link.springer.com/chapter/10.1007/978-3-030-22312-0_20>.

[7] SCHOENMAKERS, Berry. Lecture Notes Cryptographic Protocols. *Technical University of Eindhoven.*[online]. 2022-02-02 [cit. 2022-12-09].Available at:<https://www.win.tue.nl/~berry/CryptographicProtocols/LectureNotes.pdf

[8] BRUSH, Kate. Asymmetric cryptography (public key cryptography). *TechTarget.com* [online]. 2021-09-27 [cit. 2022-11-10]. Available at: <https://www.techtarget.com/searchsecurity/definition/asymmetric-cryptography>.

[9] SULLIVAN, Nick. A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography. *The Cloudflare Blog* [online]. 2013-10-24 [cit. 2022-11-01]. Available at: <https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/>.

[10] CERTICOM RESEARCH. Standards for Efficient Cryptography: SEC 2: Recommended Elliptic Curve Domain Parameters. *Certicom Corp* [online]. 2000-09-20 [cit. 2022-11-20]. Available at: <https://www.secg.org/SEC2-Ver-1.0.pdf>.

[11] DZURENDA P, RICCI S, HAJNY J, MALINA L. Performance analysis and comparison of different elliptic curves on smart cards. *In 2017 15th Annual Conference on Privacy, Security and Trust (PST)* 2017-08-28 [cit. 2022-12-09]. Available at: <https://ieeexplore.ieee.org/document/8476956

[12] KOHLI, Kerman. Learning Cryptography, Part 3: Elliptic Curves. *Medium* [online]. 2019-08-15 [cit. 2022-11-20]. Available at: <https://medium.loopring.io/learning-cryptography-elliptic-curves-4cfd0bdcb05a>.

[13] BUTERIN, Vitalik. Exploring Elliptic Curve Pairings. *Medium* [online]. 2017-01-16 [cit. 2022-11-20]. Available at: <https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627>.

[14] WASHINGTON, Lawrence C. Elliptic curves: number theory and cryptography. *Chapman and Hall/CRC* 2008 [cit. 2022-12-09]. (pp.90-92) ISBN 9781420071467

[15] BUCHANAN, William J. Crypto Pairing. *Asecuritysite.com* [online]. 2022 [cit. 2022-11-20]. Available at: <https://asecuritysite.com/pairing/>.

[16] BONEH, Dan and Xavier BOYEN. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *J Cryptol 21, 149–177* [online]. 2008 [cit. 2022-11-20]. Available at: <https://doi.org/10.1007/s00145-007-9005-7>.

[17] KOGAN, Dima. Lecture 5: Proofs of Knowledge, Schnorr's protocol, NIZK. *Stanford Crypto* [online]. 2019 [cit. 2022-11-22]. Available at: <https://crypto.stanford.edu/cs355/19sp/lec5.pdf>.

[18] YI, Xun, Russell PAULET a Elisa BERTINO. Homomorphic Encryption and Applications. *Springer Cham* 2014 [cit. 2022-12-10] ISBN 978-3-319-12229-8.

[19] GILLIS, Alexander S. Homomorphic encryption. *TechTarget.com* [online]. 2022-08-24 [cit. 2022-11-28]. Available at: <https://www.techtarget.com/searchsecurity/definition/homomorphic-encryption>.

[20] GENTRY, Craig. A FULLY HOMOMORPHIC ENCRYPTION SCHEME. *Stanford, California* [online] 2009 [cit. 2022-11-28]. Available at: <https://crypto.stanford.edu/craig/craig-thesis.pdf>. A Dissertaion. Standford University.

[21] RICCI, Sara. MOK - Modern Cryptography: Secure Computation Cycles - Homomorphic Encryption. *Brno: VUT* [lecture]. 2022-10-21 [cit. 2022-11-29].

[22] BLEUMER, Gerrit. Group Signatures. *SpringerLink* [online]. 2011 [cit. 2022-12-05]. Available at: <https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_208>.

[23] CAMENISCH, J.; MÖDERSHEIM, S.; NEVEN, G.; PREISS, F.S.; RIAL, A. A Prolog Program for Matching Attribute-Based Credentials to Access Control Policies. *Researchgate.net* [online] 2015 [cit. 2023-05-05]. Available at: <https://www.researchgate.net/publication/283672143_A_Prolog_Program_for_Matching_Attribute-Based_Credentials_to_Access_Control_Policies>.

[24] TUSHIE, David. An Introduction to NFC Standards. *ICMA* [online]. 2012-10-16 [cit. 2023-04-30]. Available at: <http://www.icma.com/ArticleArchives/StandardsOct12.pdf>.

[25] NFC-FORUM. NFC - Specification Releases. *NFC-Forum* [online]. 2021-03-01 [cit. 2023-04-30]. Available at: <https://nfc-forum.org/our-work/specification-releases/>.

[26] DARDE, Laurent. Start a Conversation with NFC: Three Operating Modes. *NXP* [online]. 2014-11-24 [cit. 2023-04-30]. Available at: <https://www.nxp.com/company/blog/start-a-conversation-with-nfc-three-operating-modes:BL-START-CONVERSATION-WITH-NFC>.

[27] Android Developers. Host-based card emulation overview. *Android Developers* [online]. 2022-03-30 [cit. 2023-05-01]. Available at: <https://developer.android.com/guide/topics/connectivity/nfc/hce>.

[28] ELENKOVm Nikolay. Accessing the embedded secure element in Android 4.x. *Blogspot* [online]. 2012-08-22 [cit. 2023-05-01]. Available at: <https://nelenk

ov.blogspot.com/2012/08/accessing-embedded-secure-element-in.html>.

[29] CardLogix. Application Protocol Data Unit (APDU). *Cardlogix.com* [online]. 2023-03-01 [cit. 2023-05-01]. Available at: <https://www.cardlogix.com/glossary/apdu-application-protocol-data-unit-smart-card/>.

[30] Oracle. What is Java technology and why do I need it? *Java.com* [online]. 2022 [cit. 2022-12-09] Available at: <https://www.java.com/en/download/help/whatis_java.html

[31] HARTMAN, James. What is Java? Definition, Meaning & Features of Java Platforms. *Guru99.com* [online]. 2022-11-29 [cit. 2022-12-09] Available at: <https://www.guru99.com/java-platform.html

[32] Oracle. BigInteger. *Oracle* [online]. 2011-08-06 [cit. 2022-11-12]. Available at: <https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html>.

[33] Apache Milagro. Apache Milagro Crypto Library (AMCL) [online]. 2019-06-11 [cit. 2022-12-09] Available at: <https://milagro.apache.org/docs/amcl-overview.html

[34] MIRACL. AMCL code. *GitHub.com* [online]. 2020-10-13 [cit. 2022-12-09] Available at: <https://github.com/miracl/amcl/tree/master/version3/java

[35] RAVIKIRAN, A.S. Use of C Language. *SimpliLearn.com* [online]. 2022-11-25 [cit. 2022-12-10] Available at: <https://www.simplilearn.com/tutorials/c-tutorial/use-of-c-language#why_learn_c_language

[36] GMP. The GNU Multiple Precision Arithmetic Library. *Gmplib* [online]. 2021-11-19 [cit. 2022-11-12]. Available at: <https://gmplib.org/>.

[37] MITSUNARI, Shigeo. MCL library. *GitHub.com* [online]. 2019-03-03 [cit. 2022-12-10]. Available at: <https://github.com/herumi/mcl>.

[38] MIXON, Erica. What is Android OS. *TechTarget.com* [online] 2020-04-08 [cit. 2022-12-10]. Available at: <https://www.techtarget.com/searchmobilecomputing/definition/Android-OS>.

[39] PEDAMKAR, Priya. Java vs Kotlin. *EDUCBA.com* [online] 2018-07-19 [cit. 2022-12-10]. Available at: <https://www.educba.com/java-vs-kotlin/>.

[40] ANDROID DEVELOPERS. Get started with the NDK. *Android Developers* [online]. 2020-09-30 [cit. 2022-12-10]. Available at: <`https://developer.android.com/ndk/guides`>.

[41] CAMENISCH, J, STADLER, M. Efficient group signature schemes for large groups. *In Kaliski, B., editor, Advances in Cryptology - CRYPTO '97, volume 1294 of Lecture Notes in Computer Science, pages 410–424. Springer Berlin / Heidelberg (1997).* [cit. 2023-04-27]

[42] SEČKÁR, M. and RICCI, S. Secure Two-Party Computation for weak Boneh-Boyen Signature. *Proceedings I of the 28th Conference STUDENT EEICT 2022 General Papers. 1. Brno: Brno University of Technology* [online]. 2022 [cit. 2022-11-14]. ISBN: 978-80-214-6029-4. Available at: <`https://www.eeict.cz/eeict_download/archiv/sborniky/EEICT_2022_sbornik_1_v2.pdf`>.

[43] HAJNÝ, J.; DZURENDA, P.; MALINA, L.; RICCI, S. Anonymous Data Collection Scheme from Short Group Signatures. *In SECRYPT 2018 Proceedings*[online]. 2018 [cit. 2022-12-09] (pp. 1-10). Available at: <`https://www.semanticscholar.org/paper/Anonymous-Data-Collection-Scheme-from-Short-Group-Hajny-Dzurenda/cd8e53342fe8cfeb56abf4ee69f681ffc99b3e4e`>.

[44] RICCI, S.; DZURENDA, P.; HAJNÝ, J.; MALINA, L. Privacy-Enhancing Group Signcryption Scheme. *In IEEE Access, vol. 9* [online]. 2021 [cit. 2023-03-22] (pp. 136529-136551) Available at: <`https://ieeexplore.ieee.org/document/9557324`>.

[45] SEČKÁR, Martin. Paillier NIZKPK. *GitHub* [online]. 2022-03-25 [cit. 2022-11-12]. Available at: <`https://github.com/xsecka04/Paillier_NIZKPK`>.

[46] GMP. Exponentiation Functions. *GmpLib* [online]. 2007-02-28 [cit. 2022-11-12]. Available at: <`https://gmplib.org/manual/Integer-Exponentiation`>.

[47] Cryptimeleon Mclwrap. *GitHub.com* [online]. 2022-03-09 [cit. 2023-05-03]. Available at: <`https://github.com/cryptimeleon/mclwrap`>.

[48] MITSUNARI, Shigeo. MCL-android library. *GitHub.com* [online]. 2020-11-22 [cit. 2023-03-21]. Available at: <`https://github.com/herumi/mcl-android`>.

[49] MOHR, Michael. GMP for android. *GitHub.com* [online]. 2020-03-03 [cit. 2023-04-05]. Available at: <`https://github.com/Rupan/gmp`>.

[50] StackOverflow. How to convert a byte array to a hex string in Java? *StackOverflow.com, User: maybeWeCouldStealAVan* [online]. 2012-03-24 [cit. 2023-05-01]. Available at: <https://stackoverflow.com/questions/9655181/how-to-convert-a-byte-array-to-a-hex-string-in-java?page=1&tab=scoredesc#tab-top>.

[51] StackOverflow. Convert a string representation of a hex dump to a byte array using Java. *StackOverflow.com, User: Dave L.* [online]. 2008-09-26 [cit. 2023-05-01]. Available at: <https://stackoverflow.com/questions/140131/convert-a-string-representation-of-a-hex-dump-to-a-byte-array-using-java>.

[52] Android Developers. HostApduService. *developer.android.com* [online]. 2023-04-12 [cit. 2023-05-01]. Available at: <https://developer.android.com/reference/android/nfc/cardemulation/HostApduService>.

[53] ROTH, Soren. LocalBroadcastManager modification. *GitHub* [online]. 2018-03-08 [cit. 2023-05-01]. Available at: <https://github.com/sorenoid/LocalBroadcastManager>.

[54] Apryse Group NV. The leading PDF library for developers. *Itext* [online]. 2023 [cit. 2023-05-02]. Available at: <https://itextpdf.com/>.

[55] CVRČEK, Tadeáš. Cryptography on Arduino platform [online]. Brno, 2020 [cit. 2023-05-07]. Available at: <http://hdl.handle.net/11012/190258>. Bachelor's thesis. Brno: Brno University of Technology. FEEC. Department of Telecommunications. Supervisor Petr Dzurenda.

[56] MOSNIER, Alain. Sha-2. *Github.com* [online] 2021-12-21 [cit. 2023-05-06]. Available at: <https://github.com/amosnier/sha-2>.

[57] KLASOVITÝ, K.; RICCI, S. Blind Issuance for Fast Keyed-Verification Anonymous Credentials. *Proceedings I of the 29th Conference STUDENT EEICT 2023 General Papers. 1.* Brno: Brno University of Technology, 2023. Accepted. [cit. 2023-05-08].

Symbols and abbreviations

AES	Advanced Encryption Standard
AID	Application Identifier
AMCL	Apache Milagro Crypto Library
APDU	Application Protocol Data Unit
BLS	Barreto-Lynn-Scott
BN	Barreto-Naehrig
DCRA	Decisional Composite Residuosity Assumption
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
ECDLP	Elliptic Curve Discrete Logarithm Problem
GMP	GNU Multiple Precision
GUI	Graphical User Interface
HCE	Host Card Emulation
IDE	Integrated Development Environment
JNI	Java Native Interface
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KVAC	Keyed-Verification Anonymous Credentials
NFC	Near-Field Communication
NDK	Native Development Kit
NIZKPK	Non-Interactive Zero-Knowledge Proof of Knowledge
OS	Operating System
PK	Proof of Knowledge

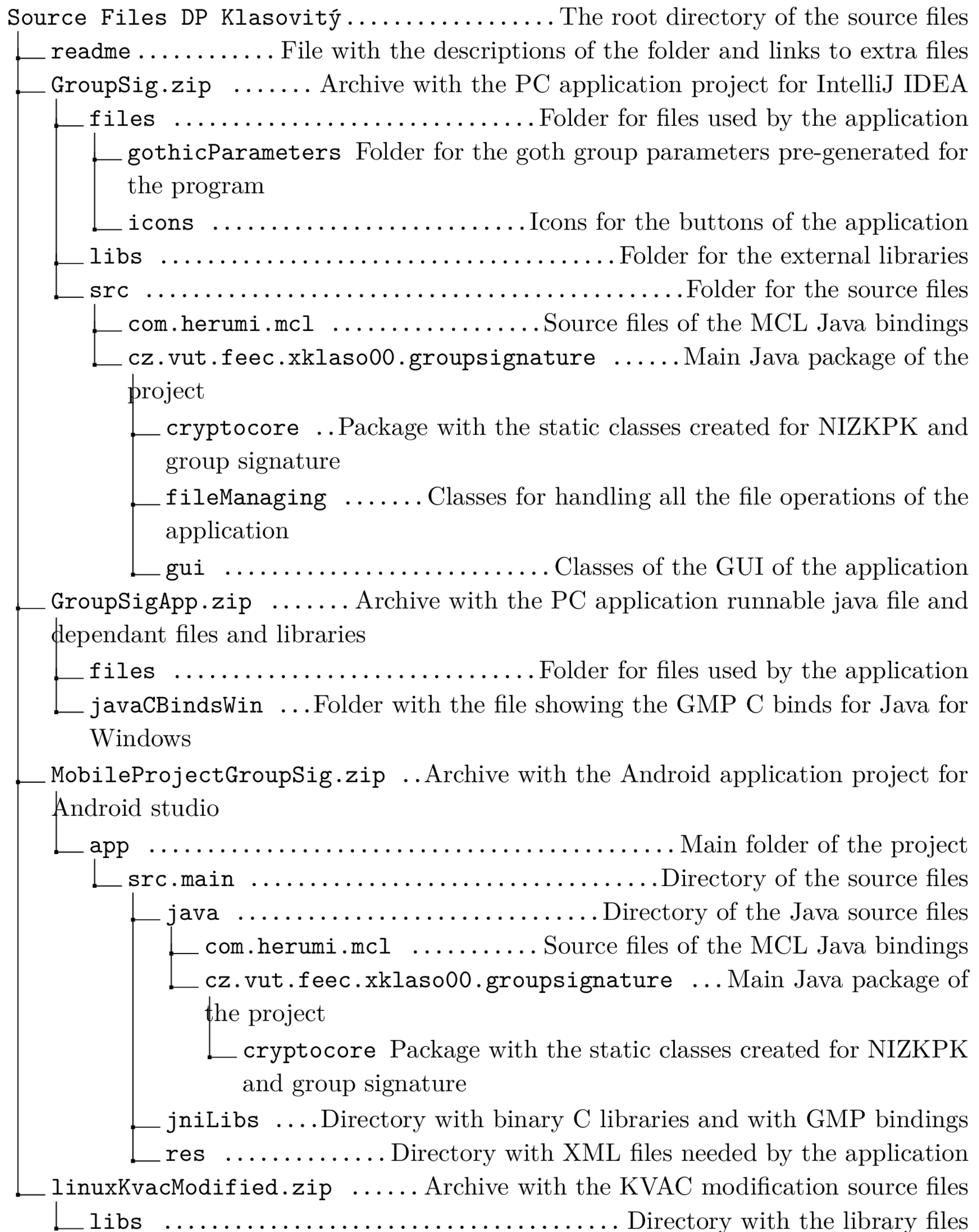
SHA Secure Hash Algorithm

RSA Rivest–Shamir–Adleman

UI User Interface

VM Virtual Machine

A Structure of the archive with the source files



- `Source Files DP Klasovitý` ... The root directory of the source files
  - `readme` ... File with the descriptions of the folder and links to extra files
  - `GroupSig.zip` ... Archive with the PC application project for IntelliJ IDEA
    - `files` ... Folder for files used by the application
      - `gothicParameters` Folder for the goth group parameters pre-generated for the program
      - `icons` ... Icons for the buttons of the application
    - `libs` ... Folder for the external libraries
    - `src` ... Folder for the source files
      - `com.herumi.mcl` ... Source files of the MCL Java bindings
      - `cz.vut.feec.xklaso00.groupsignature` ... Main Java package of the project
        - `cryptocore` .. Package with the static classes created for NIZKPK and group signature
        - `fileManaging` ... Classes for handling all the file operations of the application
        - `gui` ... Classes of the GUI of the application
  - `GroupSigApp.zip` ... Archive with the PC application runnable java file and dependant files and libraries
    - `files` ... Folder for files used by the application
    - `javaCBindsWin` ... Folder with the file showing the GMP C binds for Java for Windows
  - `MobileProjectGroupSig.zip` .. Archive with the Android application project for Android studio
    - `app` ... Main folder of the project
      - `src.main` ... Directory of the source files
        - `java` ... Directory of the Java source files
          - `com.herumi.mcl` ... Source files of the MCL Java bindings
          - `cz.vut.feec.xklaso00.groupsignature` ... Main Java package of the project
            - `cryptocore` Package with the static classes created for NIZKPK and group signature
        - `jniLibs` ... Directory with binary C libraries and with GMP bindings
        - `res` ... Directory with XML files needed by the application
  - `linuxKvacModified.zip` ... Archive with the KVAC modification source files
    - `libs` ... Directory with the library files

B Manual for the applications

This chapter serves as a user's manual on how to install, run and use the group signing applications.

B.1 PC application

The PC application runnable file is located in the `GroupSig_app.zip` archive that is part of the files attached to this thesis.

B.1.1 Installing the PC application on Windows

To run the application, the Java environment must be installed on the machine. The program is compiled for lower versions of Java (version 11) to have better compatibility, so versions 11 and newer should be present on the system. The version can be checked with `java -version` command in the command prompt. If Java is present the file can be run either by double clicking the `GroupSig.jar` file or by running `java -jar Groupsig.jar` in the command prompt, if you wish to see a bit more about the running application. The file should not be moved from the folder it is located in, as it uses the other files in the folder.

Using GMP bindings in Windows

It is possible to use the GMP bindings in Windows, however, it is not recommended, as it will result in slower runtime and is more demanding for the user to run the program this way. If you wish to use GMP on Windows, the compiled `gmp_forJava.dll` library should be usable on other Windows 64-bit platforms. The Java binding library should be able to link with the dynamic library files `libgmpxx-9.dll` and `libgmp-13.dll`. If there is a problem with using GMP the application will not enable it.

B.1.2 Installing the PC application on Linux

To run the application on Linux, it is needed to compile and install the GMP, MCL library, and Java version 11 or newer. A shell script `setup.sh` was created and is located in the `GroupSig_app` folder to install all necessary libraries to run the application. The script uses `apt-get` so it will be usable on Ubuntu, but on some systems, it might be needed to change it. To run it, make it runnable with `chmod +x setup.sh` and then call `./setup.sh`, on some systems calling it with `sudo ./setup.sh` might be needed. The script is shown in Listing B.1, in case you

want to know what is downloaded and installed. It can be also used to follow the commands if you, for example, have a JDK already installed since you will have to specify the location of JDK, as shown on line 26 of the listing.

Listing B.1: The setup.sh script to install the program with all the dependencies

```
#!/bin/bash
#install dependencies essential for the installation and
   avalible with apt-get
sudo apt-get update
sudo apt-get install -y build-essential g++ cmake git
   libgmp-dev libssl-dev pcscd
# Install JDK with the system variables
sudo apt-get install -y openjdk-11-jdk-headless default-
   jdk
# Clone mcl and build the library
git clone https://github.com/herumi/mcl
cd mcl
mkdir build
cd build
cmake ..
make
cd lib
sudo cp libmcl.a /usr/lib
cd ../..
# Build and install mcljava
cd ffi/java
mkdir build
cd build
cmake ..
cmake --build . --config Release
sudo cp libmcljava.so /usr/lib
# Build and install the GMP bindings used in the program
cd ../../../../
g++ -fPIC -I /usr/lib/jvm/default-java/include/ -I /usr/
   lib/jvm/default-java/include/linux/ -shared -o
   libgmp_forJava.so gmp_forJava_linux.cpp -lgmp -lgmpxx
sudo cp libgmp_forJava.so /usr/lib
# Restart pcscd service for NFC, as sometimes it might
   not start after installation
```

```
sudo systemctl restart pcscd
echo "Setup completed successfully"
#Finally run the app
java -jar GroupSig.jar
```

B.2 Installation of the mobile application

The mobile application can be installed from an apk file or directly from the Android studio project in the source files attached to this thesis. Since the apk file was too big it is not part of this directory, but it was uploaded to google drive [1] and can be downloaded from there. Installation with this file is recommended as it is easier and does not need any other software.

To install the application download the file to your Android mobile phone and open it. You might need to find it with the file explorer of your phone as Google Drive will not allow you to install it straight away, as it does not trust unknown applications. When you run the apk file you might have to confirm that you trust the application before the installation. Then you can just choose to install it and the application will install on your phone and can then be opened.

For the application to properly work, you will have to allow applications to use NFC on your mobile phone, this can be different in each device. Generally, navigate to the settings of your phone and search for NFC. Then you will need to enable the use of HCE and also in the NFC tap and pay settings, it should be checked that the opened application will be used instead of a default one. How these settings look on a Xiaomi device is shown in Figure B.1.

After you open the application you will have a generated ID shown on the screen. With the button `Delete User` you will delete all your user data such as keys and ID, so only do that if you want to make a new user. The `Reset` button resets the application in case something goes wrong during the communications. The `Disable GMP` button is for a case when you want to use pure Java instead in the computations, but it is not recommended as it will result in slower computation times.

[1] Apk file location: `https://drive.google.com/drive/folders/1icW_Hhojm8vLAM4dg99Pu0laEoQxtWDa?usp=sharing`

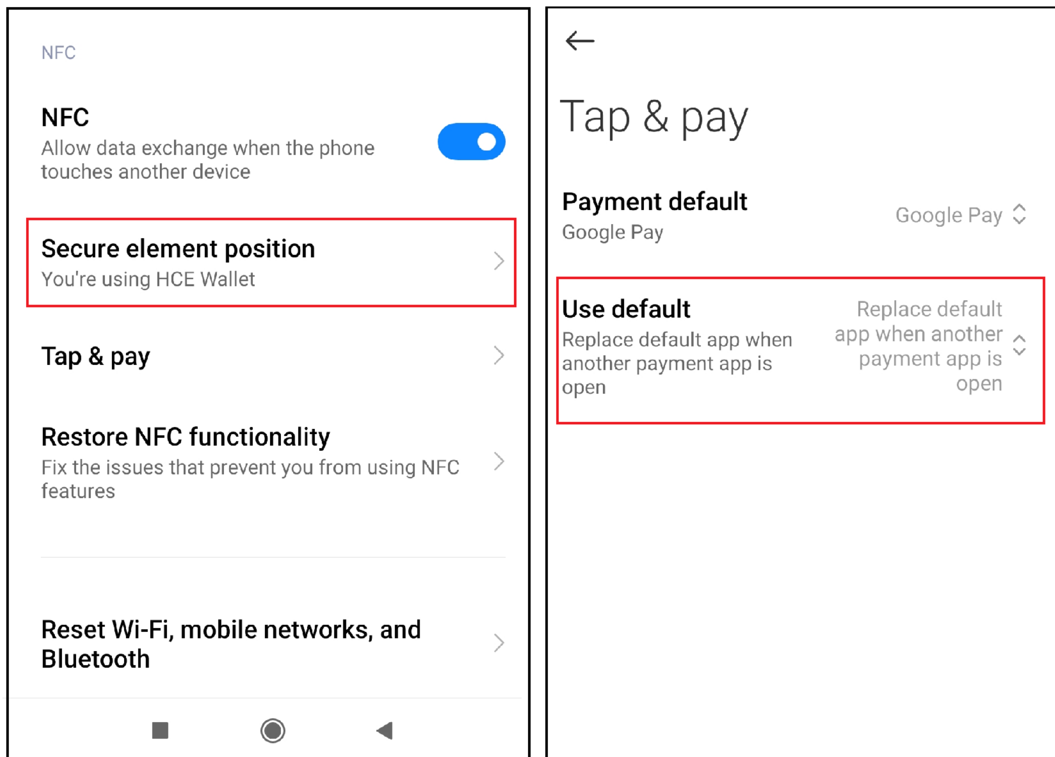


Fig. B.1: The NFC settings on a Xiaomi phone.

B.3 Using the PC application with the Android application

After the initial installation, the application can be run by either clicking the `GroupSig.jar` file (on Windows) or by running the `java -jar GroupSig.jar` command. In the screen that opens as shown in Figure B.2, you can choose which application to run. In order to use the application correctly you have to connect a card reader to the computer. The one used during the testing was *ACR1251 ACS reader*. On Windows the drivers should be installed automatically, on Linux they are installed with the installation script. If you are using a VM make sure the VM has control of the USB reader.

Manager application

If you wish to use GMP in the computations you can enable it with the `Enable GMP` button. For the manager, you can generate a new account with the `Generate new manager` button or use the existing one where the password is `hello`. If you create a manager you will be prompted to choose a password as seen on the left of Figure B.3. The password will be used to encrypt the key file and to log in as the manager. To log in as the manager click the `Manager application` button and you will be

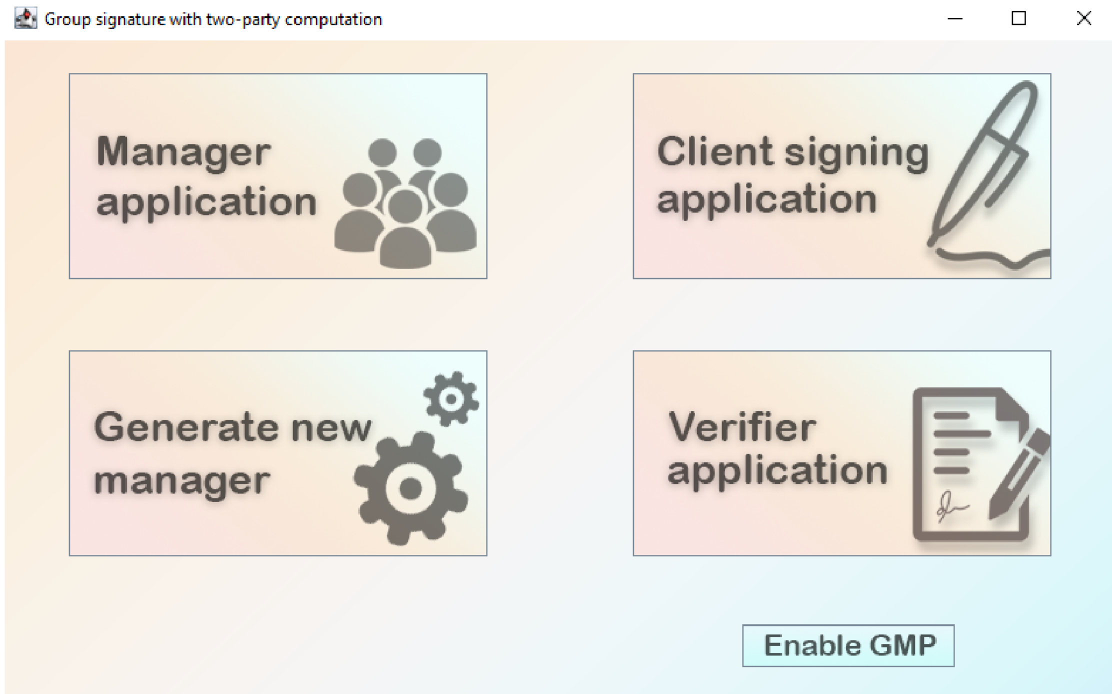


Fig. B.2: The main window of the application.

asked to choose a manager file from the system. The files are generated into the `/files` folder. Choose the file with format `yourID_keyEnc.ser`. You will then be asked to input your password as seen on the right of Figure B.3.

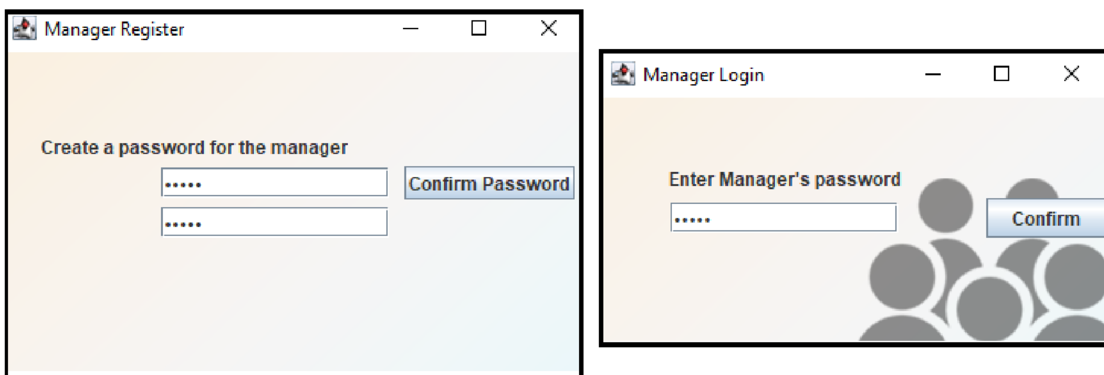


Fig. B.3: The register and login windows of the PC application.

After this, the manager application will be opened as seen in Figure B.4. You can add users to the group using the two-party computation. For this, you need an Android mobile phone with the mobile application installed and running. You can choose if you want to run a new setup for each user in the lower right corner. If you uncheck this, the setup will be generated only once and used for all users during the session. Click the `Add a user` button and the application will pre-compute the

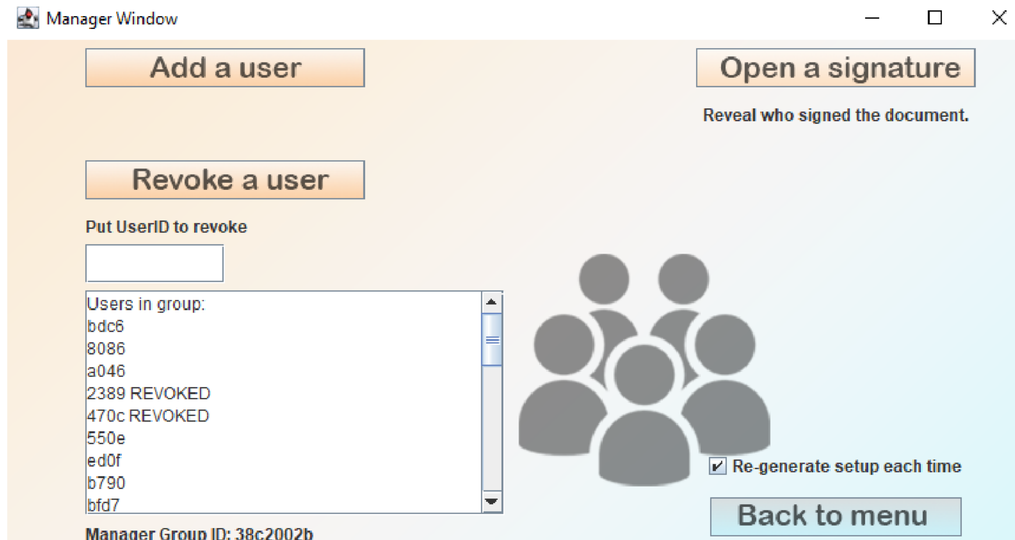


Fig. B.4: The manager application.

setup. The application informs you of the next steps with text under the button. When the setup is generated you will be prompted to put the phone on the NFC reader. When the application says *Manager ZK sent, waiting for mobile* and the loading symbol on the mobile screen changes to blue, as seen on the left of Figure B.5, you can remove the phone from the reader. Then the phone will inform you when the mobile computation is done by removing the blue loading symbol and printing text that the computation is done as shown in the middle of Figure B.5. Then you can put the phone back on the reader. Wait until the loading symbol that appears disappears again and a blue checkmark symbol appears, as seen on the left of Figure B.5. Then you will be informed that the user was added successfully, and can remove the phone. It is possible there sometimes will be an error in the communication, you will be informed of this in the PC application, in that case, run the protocol again by clicking the `Add a user` button and resetting the mobile with the `RESET` button.

In the lower-left corner, you can see the users in the group. You can remove the users from the group by typing their ID in the text field above and pressing the `Revoke a user` button. This user will be added to the revocation list and his signature will no longer be valid. You can also open signatures to find out who from your group signed a chosen file. To do this click the `Open a signature` button and choose a signed PDF file. The program will then show you the ID of the signer under this button.

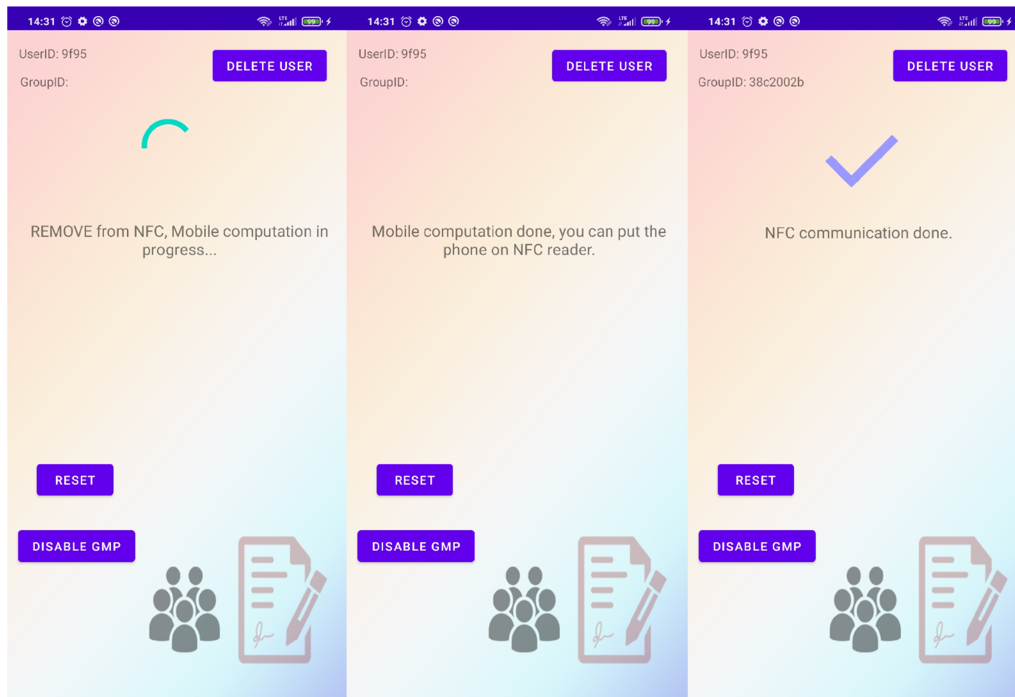


Fig. B.5: The mobile application during the protocol.

Client's signing application

From the main window shown in Figure B.2, you can also choose to run the client's signing application. This application will allow you to create a signature of a file and save it to its metadata. You must use a mobile phone that was added to a group already with the manager application. You also need an NFC reader for the communication between the PC and the phone.

When you choose this application, a new window will open as shown in Figure B.6 in the top left. Here click the middle button and choose a PDF file from the file explorer that opens. Make sure you don't have the PDF opened in a different window as that would result in the signature not being saved, as the program could not override an opened file. After you choose the file, you will be asked to put your phone on the NFC reader. Make sure you have the app running on the phone and that you are a member of a group. Then the signature will be created on the phone, sent to the PC, and saved to the PDF. You will be informed of a successful process on the screen as seen in the bottom part of Figure B.6.

The verifier application

The last part of the application is the verifier application. After clicking its button in the main menu new window will open. Here you have the option to verify the signatures of signed PDF files (signed with this application of course). When you

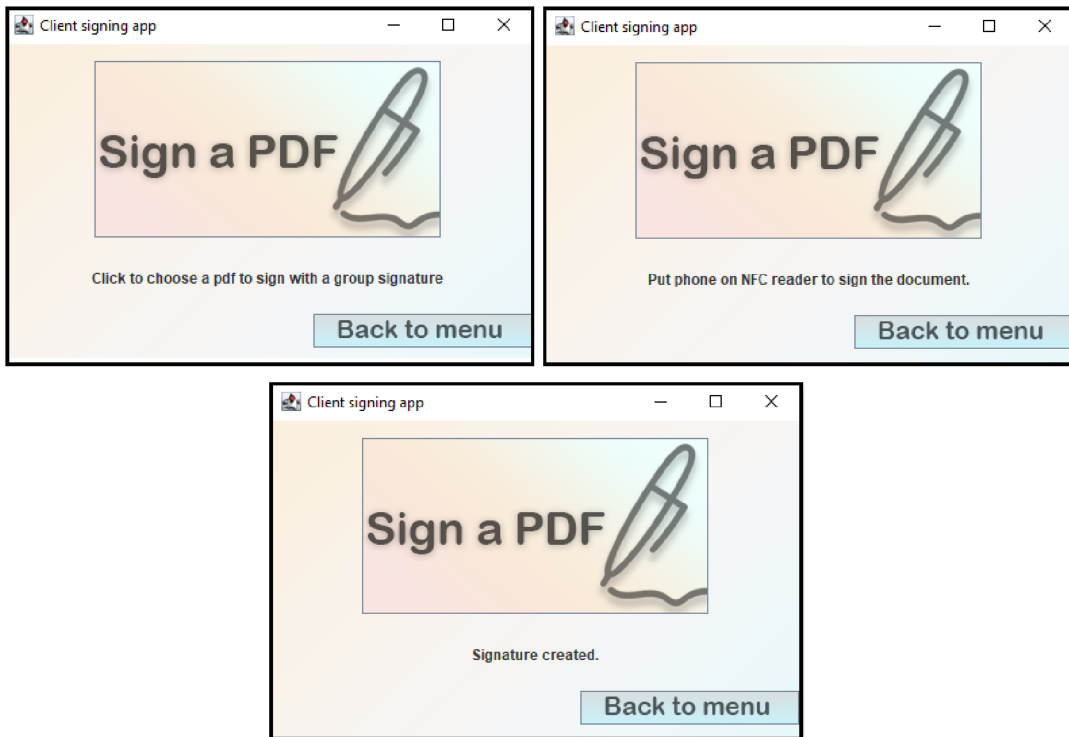


Fig. B.6: The client's signing application.

click the button you can choose a signed file to check the signature of. Then the application will inform you wherever the signature is valid or not as shown in Figure B.7, it will also tell you what group signed the file.

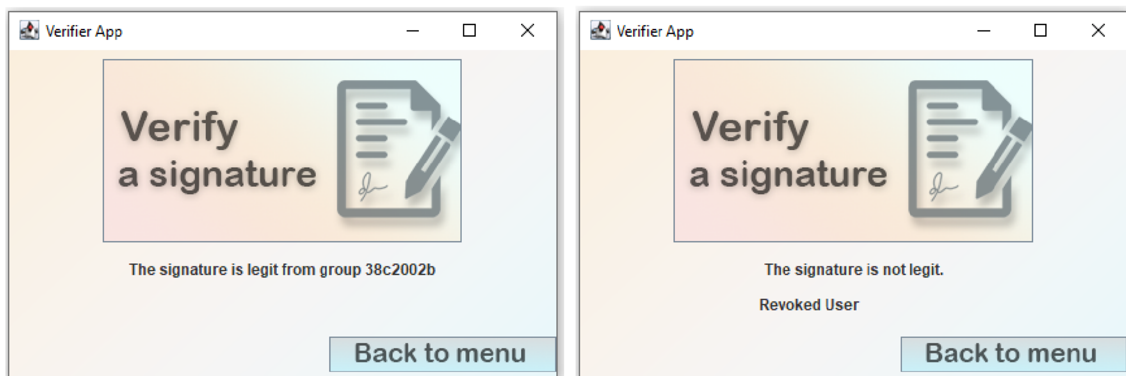


Fig. B.7: Result of the verification process.