NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

TIAGO ALEXANDRE PORTUGAL MORAIS

Bachelor in Computer Science and Engineering

RSA GROUP SIGNATURES FOR NATIONAL IDENTITY CARDS

LEVERAGING ZERO-KNOWLEDGE PROOFS TO PROVIDE ANONYMOUS AND AUTHENTIC SIGNATURES WITHIN KNOWN GROUPS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

RSA GROUP SIGNATURES FOR NATIONAL IDENTITY CARDS

LEVERAGING ZERO-KNOWLEDGE PROOFS TO PROVIDE ANONYMOUS AND AUTHENTIC SIGNATURES WITHIN KNOWN GROUPS

TIAGO ALEXANDRE PORTUGAL MORAIS

Bachelor in Computer Science and Engineering

Adviser: Doutor Alexander Andrew Davidson
Assistant Professor, NOVA SST

Examination Committee

Chair: Doutor David Fernandes Semedo
Prof. Auxiliar, NOVA SST

Rapporteur: Doutor Bernardo Luís Fernandes Portela
Prof. Auxiliar, FCUP

Member: Doutor Alexander Andrew Davidson
Prof. Auxiliar, NOVA SST

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2024

RSA Group Signatures for National Identity Cards

Leveraging Zero-Knowledge Proofs to Provide Anonymous and Authentic Signatures Within Known Groups

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.0) [1].

Acknowledgements

I would like to express my gratitude to my supervisor Professor Alexander Davidson for his contributions throughout the dissertation. His insights, advices, words of encouragement and unconditional support throughout every set-back over these past months are what kept me on track and with a clear vision in mind, which allowed me to accomplish the work presented here.

To all the friends and colleagues that kept me sane throughout the most stressful times I express my gratitude. Without you these past years would probably have gone by slower and with more headaches.

I thank my dog, Neco, for the unconditional love every time I arrived home for the past 12 years after school, and for giving his support in his own way.

Finally I want to thank my parents and brother, who have sacrificed so much for me and have always given me support and guidance, making me the man I am today.

Abstract

Throughout history, human collaboration has enabled us, as a species, to survive hardships, learn, and make better decisions. For example, nations have been shaped by the electorate vote, the process in which leaders are chosen to represent their citizens. These decision-making collaborations can be thought of as computations, which evolve with technology, allowing greater collaboration opportunities. In these computations, the inputs remain private while the overall result is public. This kind of computation is called Secure Multi-Party Computation.

Two issues that exist with this computation, however, result from trust assumptions. There are those individuals who may not want their identity shared with the rest, but would desire to collaborate with confidence that the rest are certified for the computation in question. And there are malicious individuals, which although certified, provide disruptive inputs that skew the final result, affecting decision-making and everyone involved. While some solutions have been adopted to address these problems, overall they remain inefficient.

This dissertation provides a solution to the certified anonymous individual problem, and examines the feasibility of guaranteeing legitimate inputs. To this effect it uses Zero Knowledge Proofs to guarantee that individuals are certified through an RSA group signature based on a known public key list which originates from a trusted source of information, such as the government. Additionally it explores how leveraging user-provided individual information available in Signed Identity Documents can provide the legitimacy required for proofs regarding a person's information.

This solution is of interest in today's widely available Internet services that require user authentication limited to citizens of a certain country, access to content that is age-restricted, or that require absolute identity authentication. Further research might allow for general-purpose proofs that can be used for any kind of input, facilitating integration with existing protocols.

Keywords: Secure Multi-Party Computation, Zero-Knowledge Proof, RSA, Privacy, Authenticity, Trusted Inputs, Signatures

Resumo

Ao longo da história, a colaboração humana permitiu-nos, como espécie, sobreviver a dificuldades, aprender, e fazer melhores decisões. Por exemplo, nações foram moldadas pelo voto eleitoral, o processo em que líderes são escolhidos para representar os seus cidadãos. Estas colaborações de decisão podem ser interpretadas como computações, que evoluem com a tecnologia, permitindo um maior leque de oportunidades colaborativas. Nestas computações, os *inputs* permanecem privados enquanto que o resultado final é público. Este tipo de computação denomina-se Computação Multipartidária Segura.

Dois grandes desafios nestes sistemas resultam de suposições de confiança. Existem indivíduos que poderão não querer a sua identidade partilhada com os restantes, mas que desejam colaborar com confiança de que os restantes estão certificados para a computação em questão. Além destes existem indivíduos maliciosos, que embora certificados, podem disponibilizar dados disruptivos que enviesam o resultado final, afetando decisões e todos os envolvidos. Embora algumas soluções tenham sido adotadas para endereçar estes problemas, no geral permanecem ineficientes.

Esta dissertação fornece uma solução para o problema de indivíduos anónimos certificados, e examina a viabilidade de garantir dados legítimos. Para este efeito usa Provas de Zero Conhecimento de forma a garantir que indivíduos são certificados através de assinaturas RSA de grupo baseadas em listas de chaves públicas conhecidas, originárias de fontes confiáveis como o governo. Adicionalmente, explora como alcançar validação dos dados fornecidos através da sua disponibilidade em Documentos de Identidade Assinados, de forma a construir provas sobre informação pessoal que garanta a sua legitimidade.

Esta solução é relevante dado o contexto atual dos serviços de Internet, que requerem autenticação limitada a cidadãos de um certo país, acesso a conteúdo restringido por idade, ou que até mesmo identificação absoluta de um indivíduo. Investigação futura poderá permitir provas de uso geral que possam ser utilizadas para outros tipos de dados, facilitando integração com protocolos existentes.

Palavras-chave: Computação Multipartidária Segura, Prova de Zero-Conhecimento, Privacidade, RSA, Autenticidade, *Inputs* Confiáveis, Assinaturas

Contents

List of Figures

List of Tables

Acronyms

AMA Agência para a Modernização Administrativa *(pp. 24, 25, 32, 33)*

CA Certifying Authorities *(pp. 7, 8, 18, 19, 23–26, 32, 35)*
CRS Common Reference String *(pp. 11, 12)*

DID Decentralized Identifiers *(p. 18)*

NP Nondeterministic Polynomial *(pp. 8, 9)*

PKCS Public Key Cryptography Standards *(pp. 14, 74)*
PKI Public-Key Infrastructure *(pp. 7, 26, 32, 35, 44)*
PSI Private Set Intersection *(p. 19)*
PSM Private Set Membership *(p. 19)*

QSCD Qualified Signature Seal Creation Device *(pp. 8, 24)*

R1CS Rank-1 Constraint Systems *(pp. 10, 13, 54, 56)*

SCEE System of Electronic Certification of the Portuguese State *(p. 8)*
SFE Secure Function Evaluation *(p. 5)*
SMPC Secure Multi-Party Computation *(pp. 1–3, 5, 6, 17–19, 23, 28, 30)*
SNARK Succinct Non-Interactive Argument of Knowledge *(p. 10)*
STAR Secret Sharing for Private Threshold Aggregation Reporting *(pp. 7, 22–26)*

W3C World Wide Web Consortium *(p. 18)*
WSL2 Windows Subsystem for Linux 2 *(p. 54)*

zk-SNARK Zero-Knowledge Succinct Non-Interactive Argument of Knowledge *(pp. 3, 9–13)*

zk-STARK Zero-Knowledge Scalable Transparent Argument of Knowledge *(p. 9)*

ZKP Zero Knowledge Proof(s) *(pp. 2–4, 8–10, 17–20, 22, 24, 44, 74)*

1 Introduction

1.1 Motivation

Collaboration is an essential and defining characteristic of humans as a whole, which has enabled our species to survive throughout history. Its purpose isn't limited to facing hardships, but also to learn and collect information that can allow for more thoughtful decisions regarding those involved in it. That learning, collection, and eventual decision, can be thought of as a computation.

Take the democratic vote as an example: Individuals of a nation make a single vote, which amassed allows them to elect a leader together. The process of collecting, counting, and deciding on that leader, can be referred to as the computation.

As technology evolved so did collaboration. With the development of computers and networks, more opportunities arose for collaboration that benefits from these technologies. Individuals can collaborate with one another over networks to compute through their machines some data that interests them alike.

Nowadays many identity cards such as the ones used in Portugal support electronic identification which enables computations such as the democratic vote to a higher degree than before, further emphasizing the importance these technologies have when it comes to collaboration.

In some cases of collaboration, individuals might have a need for computing some data collaboratively while keeping their inputs hidden. This type of collaboration uses a form of computation called Secure Multi-Party Computation (SMPC). Inputs provided by individuals remain hidden throughout the process of the computation and thus allow them to learn the result of the computation with guarantees of their inputs being hidden [2].

1.2 Problem Definition

Two problems present in this collaboration result from trust assumptions. Participants of the computation knowingly engage and share their identity to ensure that the computation

has a solid source of inputs, but in many cases would rather not share their identity while keeping this confidence that all participants are qualified. Additionally, malicious individuals, although certified, might take advantage of their input anonymity and provide invalid or disruptive inputs capable of manipulating or skewing the collaborative final result of the computation.

Real-world scenarios where this behavior might be exploited not only cause harm to the honest individuals of the computation, who might depend on that result for their personal purposes but also third-party users that benefit from the honest result of the computation and are subsequently affected by wrong results.

Such examples where individuals can be affected directly and indirectly are sustainability metrics [3], in which gathering of fossil fuel emissions from a group of organizations can be manipulated to give an estimate of emissions that fails to disclose its true magnitude to the wider population accurately. The nature of the inputs provided by the organizations, which are associated with intellectual property, requires that they remain private, thus directly validating them through trusted parties becomes very invasive.

As outlined by [4], some proposed solutions implement protective measures such as digital signatures, encryption, and authorization. A known reputation table of participants might also be used for a participant to decide if they want to engage with another. Another example is data use agreements that identify the purpose of the data, and have penalties enforced if the agreement is violated.

1.3 Solution

This dissertation presents a solution to the first of these problems while analyzing the feasibility of a solution to the second. Through the use of Zero Knowledge Proof(s) (ZKP), we determine if the solution is efficient and secure enough so that it can be broadly used. These can be used as additional inputs in SMPC protocols to ensure participant identity validity and input validity as well, keeping their value, individual information, and related information hidden, other than the fact that they are legitimate. They can also be used independently in other protocols, as long as some mechanism is used to mask the source of the proofs to ensure anonymity.

To this effect, we have implemented a ZKP circuit capable of validating a user's identity from a known list of public keys, and ensuring that the data they provide is accompanied by a signature to guarantee that the information originated from the legitimate user and has not been tampered with, providing authenticity and integrity.

This circuit is capable of being used in computations such as democratic votes or petitions, where participants might not want to disclose their identity but feel the need to provide their input. Through this circuit, any entity could validate every vote and be confident that their source is legitimate.

Another use case of this circuit is for specific data, such as user-provided individual information available in their personal Signed Identity Document, guaranteeing that their

input is valid according to their personal Signed Identity Document, without exposing it or any other private information. It's important that this information is valid according to such document, as it originates from a trusted source of information, in this case, the government, and can provide authenticity mechanisms.

However, these mechanisms are not used in the circuit and would require that another program be in charge of building the proofs upon reading the data itself.

These are interesting cases as nowadays many services on the web require that users sign certain documents to gather petitions, make votes, or authenticate themselves to prove they are above a certain age, from a certain country, or that they are who they claim to be.

1.4 Overview of Techniques

Our solution is a circuit which makes use of Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK), a non-interactive ZKP. It leverages signatures created with the Cartão de Cidadão's signing key, which act as important data points. It can also be adapted to work in restriction with another program, so that other data such as individual information can be included in the proof. However, certificate validation would need to be done outside of the proof due to limitations in the technology.

These proofs are generated by the client, which must have access to the *Wasm* file and the circuit's proving key, both of which are provided by the verifier. Upon computing a *witness* of all inputs, using the Wasm file, the proof is generated using the proving key, alongside the public inputs. The proof and the public inputs are then sent to the verifier for validation. While we haven't included it, some masking of the client should be required to guarantee anonymity, which includes other mechanisms such as SMPC protocols.

The verifier, upon receiving the proof and the public inputs, can begin the verification process using the verifying key in order to determine if the proof is legitimate, even though it does not have knowledge of the secret inputs. If that is the case it is accepted, otherwise the proof is rejected and the client's inputs are not accounted for.

1.5 Contributions

The following contributions were derived from both the development of the solution and the analysis of various other aspects.

1. **Feasibility Analysis of Using Zero-Knowledge Proof for Identity Verification** - We analyze the requirements for developing a ZKP that can validate an individual's personal information embedded in smart cards while keeping the data hidden. Additionally, we present an alternative approach and assess its feasibility.

2. **Development of Zero-Knowledge Proof for Verifying Group Signatures** - A Zero-Knowledge Proof was developed to verify group signatures, ensuring that a certified

user has provided a valid signature without revealing the identity of the user or any details other than the content of the signed document.

3. **Program for Restricting Inputs in Group Signature Zero-Knowledge Proof** - A program was developed to work alongside the Group Signature Zero-Knowledge Proof, ensuring that the inputs provided in the signed content correspond to the information on a citizen's card. This program was created in response to the requirements outlined in item 1.

1.6 Document Structure

The document is structured in the following manner, divided into chapters:

- **Chapter 1** - Describes the context and motivation for the thesis along with a brief description of the proposed solution and what contributions are expected from it.
- **Chapter 2** - Establishes and provides a base understanding of the technologies and areas of work related to this dissertation, such as Signed Identity Documents and ZKP.
- **Chapter 3** - Showcases work that has been done and is related to the dissertation, in order to support its solution proposal and establish the notoriety of the context.
- **Chapter 4** - A brief discussion of the plans we had in mind in the initial stages of the dissertation, before deciding to change direction.
- **Chapter 5** - A brief overview of the motives behind the change in direction.
- **Chapter 6** - Establishes the proposed solution's design, a description of its requirements, decisions made, system architecture and technologies used, along with other details.
- **Chapter 7** - Describes in detail the implementation of the Cartão de Cidadão Data Reader & Signer, and considerations taken during its development.
- **Chapter 8** - Describes in detail the implementation of the RSA Signature ZKP
- **Chapter 9** - Describes in detail the implementation of the RSA Group Signature Verification ZKP, including its two variants and their differences.
- **Chapter 10** - Presents the methods used to conduct the evaluations, along with the required setup, and their results. The results are accompanied by observations and discussions, and ultimately with a final analysis.
- **Chapter 11** - Presents the conclusions obtained from the development of the dissertation, and reflections regarding future work.

2

Background

This chapter provides knowledge about the concepts addressed in this dissertation that are essential to its development. It is divided into 4 sub-sections:

1. The first describes what Secure Multi-Party Computation is (in addition to what issues are still present);
2. The second describes what Signed Identity Documents are and how they achieve authenticity;
3. The third describes what Zero Knowledge Proofs are, its components and technologies, and how they can contribute to the solution.
4. The fourth describes what RSA is, including its signature scheme PKCS#1 v1.5.
5. Finally the fifth describes what Group Signatures and Ring Signatures are, and why they are used.

2.1 Secure Multi-Party Computation

First introduced as Two-Party Computation by Yao [5] through the *Millionaires Problem*, SMPC (or Secure Function Evaluation (SFE)) is defined by Goldreich [6] as the arbitrary function of m arguments in which m interested parties willingly provide their corresponding input in order to obtain the result of the function, whilst preventing other parties from gathering any information besides the result of said function, as figure 2.1 illustrates.

Instead of depending on a trusted third party that could receive the parties' inputs and compute the function for them, these parties attempt to achieve the same goal by communicating with one another instead. In what is called the *Real-World/Ideal-World* paradigm, protocols that implement SMPC try to establish some security conditions that are met in the *Real-World* model the same way they are met in the *Ideal-World* model, which usually uses the trusted third-party approach. This way, parties communicating with one another through the protocol are as secure as if they were using a trusted third-party [7].

SMPC protocols must satisfy two conditions:

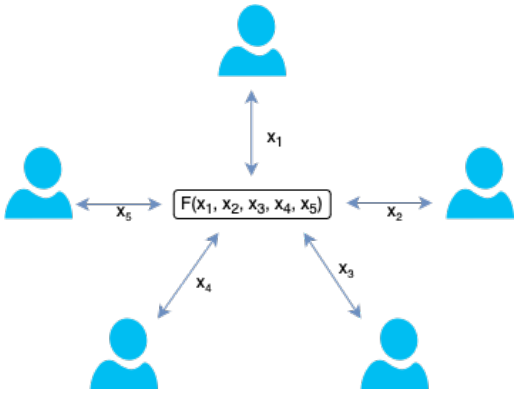

Figure 2.1: Example of SMPC

- **Correctness**: The correct result of the function is computed and captured by the participants;
- **Privacy**: The only new information revealed is the result.

These conditions are ensured through **Formal Security Models** and proofs of security that reveal a protocol's behavior is indistinguishable from one with Ideal Functionalities.

It must be acknowledged that participating parties might not follow the guidelines established by the protocols, and such scenarios are more than plausible to happen in the real world. The parties might manipulate their inputs such that they don't fit in accordance with the protocol, or they could deviate their behavior from what is expected from it. These parties are considered adversaries, parties that may try to corrupt or learn more than intended from a protocol. As Goldreich et al. put it [8], they are faulty machines that can be *passive* or *malicious*.

- **Passive Adversary**: A party that may not want to corrupt the computation but would use third-party methods to learn information about the parties' inputs.
- **Malicious Adversary**: A party that may break the privacy constraint but whose main goal is to change the outcome of the computation.

Many current SMPC protocols have some degree of protection against these adversaries by using multiple techniques such as majority quorums, secret-sharing, oblivious transfer, encryption, signatures, and zero-knowledge proofs [9].

As pointed out by [10] it is assumed that participating parties have an incentive to offer inputs that will result in an outcome they genuinely desire to acquire. The very nature of SMPC is to hide inputs, hence verifying inputs would counter privacy guarantees. Thus, it is difficult to determine if the inputs are reasonable without checking them. You may recognize that this problem is what we will be trying to solve.

It is difficult to resolve this issue no matter how secure a protocol is, but it's important to note that even though the end result might not hold some value we trust in, wrong inputs don't affect the conditions of security of SMPC or its behavior.

Multiple protocols have been developed to satisfy these conditions in each of their planned scenarios. Secret Sharing for Private Threshold Aggregation Reporting (STAR) is an example of such a protocol. Other protocols include BaRK-OPRF for private set intersection and PRIO for computation of aggregate statistics [11].

2.2 Signed Identity Documents

Signed Identity Documents used to identify individuals are comprised of information regarding the individual they are assigned to. Name, date of birth, and a face picture are usually present in such documents, among other important information the document might entail. These documents aren't restricted to a physical format however, technological advancements in the last century have enabled and created a need for digital formats. Thus, it has become the norm for physical documents to accommodate chips capable of holding that individual's information in digital format, commonly known as smart cards.

Physical documents, which are susceptible to forgery and consequently identity falsification, employ measures of ensuring they are legitimate. This can be done through a plethora of ways, but ultimately they allow for verification procedures that capture illegitimacies and authenticate the documents. Digital documents are no exception to this risk, and so, measures are also employed to allow the capture of illegitimacies and authentication.

These measures are based on Public-Key Infrastructure (PKI). PKI employs cryptographic and mathematical algorithms which are established as norms to provide private and public key pairs to users. The former is held in secret by the user and used to sign documents, and the latter is publicly available and is used to verify said signature. In combination with trusted third-party entities called Certifying Authorities (CA), the authenticity of such signatures is guaranteed through signed certificates. The individuals are then able to emit digital signatures that behave the same way as traditional signatures or even to a greater extent [12, 13].

2.2.1 Cartão de Cidadão

Cartão de Cidadão is Portugal's identity document assigned to its citizens. In actuality, it's a combination of many documents, such as the ones used for social security and taxes, to name a few.

It displays general information about the individual it belongs to, such as name, date of birth, and face picture. More importantly, it's accompanied by a smart card chip holding this information and more, such as an address, as can be seen in figure 2.2. The smart card holds numerous certificates, of which, and to our interest, including authenticity and qualified digital signature certificates, both provided by a CA named Multicert [14].

These certificates are built using X.509 v3, which links a non-repudiation public key (to protect against individuals who falsely deny having performed a certain action) to its

Figure 2.2: Specimen of Cartão de Cidadão [15]

holder (in this case, the individual). They present other fields that identify the certificate by a unique serial number, the subject the certificate was assigned to, the issuing CA and its signature, and many more. Additionally, these certificates are built following policy guidelines identified in the certificate itself, which include the System of Electronic Certification of the Portuguese State (SCEE)[16–18].

The smart card is a Qualified Signature Seal Creation Device (QSCD) [19] which follows strict requirements that ensure with high guarantee the protection of private keys from replication or forgery methods, thus ensuring high legal certainty regarding digital signatures. The certificate of qualified digital signature has a public key associated with a private key held by the QSCD. By using a PIN code known by the individual, QSCD enables this private key to be used to sign digital documents which in turn can be verified by anyone using the public key of that certificate [17].

2.3 Zero Knowledge Proofs

ZKP were initially introduced by Goldwasser, Micali, and Rackoff [20] as probabilistic theorem-proving procedures. They make it possible for a prover to convince a verifier of the validity of a claim without revealing any information besides the statement itself, by providing proofs.

Provers can assert knowledge of a certain property while ensuring that the verifier cannot gain any additional information beyond what it already knows and the proof itself. The probabilistic nature of these proofs provides the verifier a way to conclude with high certainty that the prover does in fact possess knowledge about said property [21]. As noted by Goldreich et al., one can compare a ZKP to being *"computationally equivalent to an answer of a trusted Oracle"* [22].

These proofs are done over Nondeterministic Polynomial (NP) problems, which as [23] put it, are hard computational problems with no efficient solution algorithm in polynomial time, but whose proposed solutions can be verified efficiently. A solution, commonly

known as a witness, to the problem, is known by a prover who doesn't want to disclose it.

As presented by [21], ZKP can be constructed for every NP problem. This is important as it gives credit to how general and widely applicable ZKP are in numerous fields, and not just mathematics or cryptography. For example, by conducting a series of rounds wherein a color-blind individual alternates the positions of two distinct-colored balls hidden behind their back, and consistently providing correct answers to the question "Have I switched the balls?", the prover can convince the color-blind person of their ability to differentiate colors without explicitly revealing the specific colors involved.

Three properties define ZKP within the bounds of a negligible soundness error, which is the probability of a false proof convincing a verifier [24]:

- **Completeness**: A verifier will be convinced when given proof of a true statement;
- **Soundness**: A verifier can't be convinced given proof of a false statement;
- **Zero-Knowledge**: The only information the verifier learns, given proof of a true statement, is that the statement itself is true.

That being said, Zero-Knowledge protocols can either be interactive or non-interactive [25, 26]:

- **Interactive**: The proof has multiple rounds of interaction between the prover and verifier. It's useful for a single verifier;
- **Non-Interactive**: The prover sends a single proof to the verifier. It's useful when there are multiple verifiers;

2.3.1 ZK-Snarks and ZK-Starks

zk-SNARK, first introduced by [27] and Zero-Knowledge Scalable Transparent Argument of Knowledge (zk-STARK), introduced by [28], are a variation of non-interactive ZKP that have been widely adopted, as they are short and easily verifiable. Furthermore, they are capable of being used without communication between prover and verifier, which proves to be great when no communication can be established between them[1] or for multiple verifiers.

zk-SNARK and zk-STARK differ in some ways. While zk-SNARK requires a trusted setup [29], zk-STARK does not. Also, the latter has better scalability even though it has larger proof sizes. Verification is faster with zk-SNARK but building proofs proves to be quicker using zk-STARK [30, 31].

The trusted setup is the process in which the prover and the verifier agree on two publicly available keys, a proving key pk, and a verification key vk. These keys are public parameters that only need to be generated once, by the verifier, using a secret λ for a given

[1] After a trusted setup has been finalized

program C, used for the proof generation and validation. This λ must remain secret, or else a prover can provide proof of a false statement which is validated by the verifier [32, 33].

It is not our focus to highlight the differences between these two approaches, but instead to understand the capability they provide to create non-interactive zero-knowledge proofs, that can be leveraged for scaled applications.

Multiple schemes and software libraries exist that support creation and use of these proofs in various languages and environments [34] such as the following, to name a few:

- Circom, a language used to construct circuits that can be used to generate zero-knowledge proofs
- snarkjs, a Javascript implementation of the zk-SNARK scheme using the Groth16 protocol
- arkworks is Rust library for zero-knowledge proof construction.
- DIZK, a Java library for zero-knowledge proof construction using parallel and distributed systems.
- ZoKrates, a ZoKrates is a toolbox for creating, deploying, and verifying zero-knowledge proofs on Ethereum.

2.3.2 Groth16

Groth16 is a cryptographic proof system introduced in 2016 by Jens Groth [35], often used for ZKP. It is a type of Succinct Non-Interactive Argument of Knowledge (SNARK) that enables verification of computations without revealing the inputs. Groth16's key advantages are its minimal proof size (three elliptic curve elements) and fast verification times. However, it requires a trusted setup, meaning parameters must be regenerated for each new program.

It transforms a program into an arithmetic circuit, expressed via Rank-1 Constraint Systems (R1CS), relying on pairing-friendly elliptic curves such as *ALT_BN128* and *BN254*, which belong to the Barreto-Naehrig family of curves [36], and *BLS12-381* [37] which belongs to the Barreto-Lynn-Scott family of curves, involving complex polynomial commitments. These curves are defined by several factors such as security, efficiency, pairing-friendliness and compatibility. The curves used in ZKP balance security and efficiency, such that in order to break their cryptographic scheme using brute force would require as much operations as required to break RSA encryption using a 3072-bit key, while keeping computations on the curve fast.

These elliptic curves are used due to their efficiency in performing complex operations such as pairing-based cryptography. Pairing-based cryptography is a cryptographic method that uses bilinear pairing to create secure cryptographic protocols by allowing two elliptic curve points to be "paired" to produce a third element in a finite field, enabling

compact and verifiable proofs that prove the validity of a computation without revealing any sensitive data.

2.3.3 Powers of tau

As mentioned before, non-interactive zero-knowledge proofs, such as zk-SNARK require a trusted setup, also called trusted ceremony, in order to define the keys that are used to generate and validate a proof. The keys are generated using parameters, also known as Common Reference String (CRS), defined during the trusted setup and are the outcome of multiple computations contributed by different participants in the ceremony.

Powers of Tau is one such ceremony, emerged from the need for trusted setups described in the Groth16 paper [35]. It is a multi-party cryptographic process that involves numerous participants who contribute to the computation. It has been improved over time from contributions such as [38, 39], reinforcing security, efficiency and decentralization. Its security is guaranteed as long as one participant is honest and securely destroys their intermediate data generated during the ceremony. The setup ensures that the generated parameters are trustworthy, and that proofs generated from it cannot be tampered with.

This intermediate data is commonly refered as *Toxic Waste* because if not properly destroyed can be used to falsify proofs. To prevent such falsification, only the CRS is retained.

Because this setup is security sensitive, it is crucial that the parameters are extremely difficult to determine, but also that the setup is trustworthy as possible. To achieve this end, Powers of Tau depends on multiple participants, only requiring that one is honest. Each participant contributes with a randomness factor (Toxic Waste), solving a designated challenge created by a coordinator, and successively posting its solution along with their randomness factor in a round-based manner, subsequently for each participant respecting the chain of previous challenges.

These randomness factors tend to be complex, and often times unknown in the present. Such examples include radioactivity levels as entropy sources and future blockchain hashes [2]. However, as long as the number of participants is high, the randomness does not require such complexity.

Powers of Tau is composed of two phases, and what follows is their succinct description.

2.3.3.1 Phase 1 - Universal Setup

The first phase of the Powers of Tau ceremony focuses on generating universal cryptographic parameters, known as the CRS, that can be used for any zk-SNARK circuit. These parameters are not tied to a specific computation but are required to ensure that zero-knowledge proofs can be verified without revealing the underlying data.

[2] using commitment schemes

Multiple participants take part by contributing randomness to the setup. Each participant downloads a challenge file, performs a computation to add their randomness, and uploads the response file. The CRS produced at the end of this round serves as the foundation for zk-SNARK proofs.

The process is designed to ensure that even if most participants are compromised, the setup remains secure as long as at least one participant acts honestly by securely discarding their toxic waste.

2.3.3.2 Phase 2 - Circuit-Specific Setup

The second phase is specific to a particular zk-SNARK circuit. This round takes the universal CRS from phase 1 and generates additional parameters that are tailored to the specific zk-SNARK circuit for which the proof will be generated.

A new group of participants or even the same group can contribute randomness to this circuit-specific setup. This ensures that any zk-SNARK project can derive unique parameters for their specific use case.

The security is similar to Round 1 in that only one honest participant is required to make the setup trustworthy.

2.3.4 Perpetual Powers of Tau

Perpetual Powers of Tau is an extension of the original Powers of Tau concept. It is designed to generate reusable Phase 1 parameters that can be shared and reused across multiple circuits. Perpetual Powers of Tau allows multiple ceremonies over time, hence its name. Unlike its counter part, it is not a one-time use, providing a sustainable and versatile foundation for other circuit adaptations. It has the same level of security as the original, and any participant can contribute at any time with fresh randomness without requiring the process to be restarted, ensuring a higher level of resilience and scalability.

By using the Perpetual Powers of Tau, developer burden is reduced thus benefiting zk-SNARK projects as a whole, only requiring that they conduct their circuit-specific setup phase. One example of Perpetual Powers of Tau is *The Privacy & Scaling Explorations* trusted setup ceremony based on the *Zcash* Powers of Tau ceremony [40].

2.3.5 Circom

The *Circom* ecosystem of tools [41] includes a programming language created to define arithmetic circuits to generate zero-knowledge proofs. Its most favorable aspect is the modularity that the language provides, through the means of templates, which are parametric circuits that can be scaled and instantiated to form larger ones. This provides better audits since testing and verifying smaller reusable circuits is more efficient. Besides, Circom provides a library of already audited circuits that users can explore and use, facilitating development overall. Along with its compiler, Circom provides the possibility

to easily construct the constraint representation in the form of a R1CS file which is used in witness generation and the Groth16 zk-SNARK trusted setup.

2.3.6 snarkjs

Snarkjs [42] is a library written in JavaScript and Web Assembly used for processing zk-SNARK. It provides the code needed to generate and validate zero-knowledge proofs from the artifacts produced by Circom. It uses the Groth16 Protocol, *PLONK* and *FFLONK* protocols and includes tools to perform multi-party trusted setup ceremonies such as the Powers of Tau ceremony and circuit-specific ceremonies.

2.4 RSA

RSA (Rivest-Shamir-Adleman) [43] is a widely used public key cryptographic algorithm for secure data transmission. The algorithm relies on a pair of keys: one for encryption, E, and the other for decryption, D.

These keys are generated through a **KeyGen** function, using two large prime numbers, P and Q, which are computationally infeasible to factor, particularly when their bit length is 2048 or greater [3].

$$KeyGen(P, Q) \rightarrow E, D$$

Multiplying these primes yields N, the modulus used in both the public and private keys.

$$N = P * Q$$

The public key E is derived from N, most specifically $\phi(N)$ (Euler's Totient Function), which represents the number of positive integers less than N that are co-prime with it.

$$\phi(N) = (P - 1) * (Q - 1)$$

This number is easily calculated if you know the values of P and Q, otherwise it is computationally infeasible as well.

E is a co-prime of N and $\phi(N)$, and in practice is typically chosen to be **65537**, as it is a big enough prime number with an efficient binary representation (only two positive bits), which speeds up the encryption process.

The private key D is computed as the modular inverse of E with respect to $\phi(N)$, ensuring that:

$$D \times E \mod \phi(N) = 1$$

[3] This key size is expected to remain secure until at least 2030

or equivalently,

$$D \equiv (E^{-1}) \mod \phi(N)$$

using the extended Euclidean Algorithm.

To encrypt a message M in order to obtain the cipher-text C, the **Encryption** function is used:

$$Encryption(M, E, N) \rightarrow C$$

Which applies

$$C = M^E \mod N$$

To decrypt a cipher-text C in order to obtain a message M, the **Decryption** function is used:

$$Decryption(M, D, N) \rightarrow M$$

Which applies:

$$M = C^D \mod N$$

With these functions, as long as the private key is kept secret, only the holder of the private key can decrypt and reveal the contents of the cipher-text.

Additionally, RSA allows the creation of digital signatures by encrypting data with the private key instead, and verifying it using the public key. However, to ensure the security of this process, additional steps such as encoding and hashing are employed.

2.4.1 PKCS

Public Key Cryptography Standards (PKCS) are a set of standards devised and published by *RSA Security LLC* that define the use of several cryptographic techniques, some of which have been patented by the company. These standards are critical for ensuring the secure implementation of cryptographic algorithms.

More specifically, *PKCS#1* [44] is what establishes definitions and recommendations for the secure implementation of the RSA algorithm. These definitions include key properties, primitive operations, secure cryptographic schemes and encoding syntax representations.

One of the cryptographic schemes included is the *PKCS#1 v1.5* signature scheme, which outlines algorithms for generating and verifying digital signatures using RSA. PKCS#1 v1.5 remains widely used, despite newer versions, due to its compatibility and historical significance in many cryptographic protocols.

2.4.1.1 PKCS#1 v1.5

Commonly known as *PKCS#1 v1.5*, *RSASSA-PKCS1-v1_5* is a signature scheme with appendix [44], which succinctly means that it allows a message signed with a signer's RSA private key to be verified using the corresponding RSA public key. This signature scheme is widely used with X.509 certificates, which are common in web communications and identity cards.

PKCS#1 v1.5 applies an encoding method known as *EMSA-PKCS1-v1_5* on the messages. This method is commonly known as a padding scheme. Only after encoding can the message be signed. Similarly, the encoding method is applied to a message prior to verification, to determine there is a match. This encoding algorithm takes a message $\boldsymbol{M}$, and produces an octet string $\boldsymbol{EM}$ whose length is determined by the RSA key size, which we'll call $\boldsymbol{EMlen}$.

The signature scheme works by applying the following functions

- $\boldsymbol{ApplyHashFunction(M, hashAlgorithm) \rightarrow hashedMessage}$
  Applies an hash function on the message **M** (e.g. *SHA-256*);
- $\boldsymbol{EncodeDigestInfo(hashedMessage, hashAlgorithm) \rightarrow T}$
  Encodes the hashed message into the standard format *ASN.1 DER DigestInfo* that includes information about the hashing algorithm and the hash value. This output has an octet string length of $\boldsymbol{Tlen}$;
- $\boldsymbol{GeneratePaddingString(EMlen, Tlen) \rightarrow PS}$
  Generates an octet string sequence $\boldsymbol{PS}$ of length $\boldsymbol{EMlen - Tlen - 3}$, consisting only of $\boldsymbol{0xFF}$ octet strings;
- $\boldsymbol{ProduceEncodedMessage(PS, T) \rightarrow EM}$
  Produce and output $\boldsymbol{EM = 0x00 \,||\, 0x01 \,||\, PS \,||\, 0x00 \,||\, T}$.

Once the $\boldsymbol{EM}$ is produced, it is signed using the aforementioned signer's private key, with the **Decryption** function, producing the signature $\boldsymbol{S}$. To validate the signature $\boldsymbol{S}$, the verification primitive is applied with the signer's public key, using the **Encryption** function. If the result matches the encoding of the message $\boldsymbol{M}$, then it's a valid signature.

2.5 Group Signatures & Ring Signatures

Chaun and Heyst first introduced the concept of a group signature scheme [45] in 1991 as a mechanism to validate signatures coming from a group of cooperating individuals. These individuals are aware of their inclusion in the group but don't know which particular individual provided a signature. Thus this mechanism ensures authenticity and preserves anonymity.

They propose numerous schemes where the privacy of the signer is protected computationally. Although some schemes require a trusted authority, most of them guarantee

that even individuals from the group can't determine who made a certain signature, even the trusted authority.

Ring signatures, introduced by Rivest, Shamir, and Tauman [46] in 2001, differ from the concept of group signatures since they not require any prior setups to be employed. Any individual can at any time select a set of other individuals, without them being aware of their inclusion, and build a signature from their private key and the set members' public keys.

This mechanism has been known for its applicability in data leakage, in which a certified individual wants do disclose some sensitive information while not revealing their identity, but providing confidence that the information is legitimate.

3

Related Work

In this chapter, the attention is directed toward existing works that possess functionalities akin to the ones intended and developed in this dissertation, or that address the context in which it is part of. A comprehensive overview of several works is provided, and some of them are examined in greater depth. The chapter is divided into 4 sections:

1. The first section shows that there has been considerable study regarding ZKP and identity mechanisms, such as proofs of identity through knowledge and development of schemes for authenticity using various types of documents;

2. The second shows that the need for valid inputs in SMPC has been mentioned and addressed by several studies, arguing for possible use cases that can benefit from this functionality, and hope for improvement in this regard.

3. The third addresses proofs of provenance over TLS, which can be used to assert that some data is provided by a trusted entity without requiring mechanisms such as signatures, allowing for greater accessibility.

4. The forth addresses ECDSA ring and group signature verification proofs using ZKP, which can be used to assert that some message has been signed by a member of the ring our group. thus providing authenticity while remaining anonymous.

3.1 Zero Knowledge Proofs for Anonymous Identity Credentials

Identity credentials are important documents whose main purpose is to identify and authenticate individuals. These can be physical or digital documents and provide entities assurance of who an individual claims to be.

The use of ZKP opens the door to the possibilities of anonymous credentials and the privacy-ensuring properties they bring along with them.

The use of ZKP with identity documents isn't a new proposition, in fact, it has been proposed by Feige et. al [47] in 1987 as a means for parties to prove their identity through knowledge, and like Thomas Beth [48], for authentication protocols in smart cards.

Burmester et al. [49] have also leveraged ZKP to build a cryptographic scheme for passport authentication and secure remote logins, which in their belief, at the time of writing in 1991, could further be enhanced by using smart cards.

More recently, the idea of privacy-preserving identification was also suggested by Rosenberg et al. [50] in response to the increasingly expanding internet services' need for identification, which makes users susceptible to trackers, data exposure risks, and, of course, privacy-invading procedures.

If these mechanisms were to be established as the norm, there would be general-purpose ZKP capable of dealing with the world's complexity regarding identity credentials and specific eligibility protocols.

In addition, through the use of Blockchain technology, decentralization from CA could be achieved, enabling greater control over personal information while ensuring greater security and privacy. In 2022 the World Wide Web Consortium (W3C) officially recommended the use of Decentralized Identifiers (DID), a decentralized solution that provides genuine trust through respectful two-way relationships that safeguard against forgery, prioritize privacy, and elevate usability, supported by many institutions, such as the U.S. Department of Homeland Security [51].

3.2 Valid Inputs in Secure Multi-Party Computation

As mentioned in previous chapters, while SMPC ensures the privacy of inputs, it, unfortunately, suffers from not having validity guarantees over them, which deters mutually suspicious parties from adhering to such protocols.

3.2.1 Validity of sustainability metrics

The need for solutions to this problem has been addressed by McDaniel and Shih [3], with an emphasis on sustainability. McDaniel makes a call for action to reach sustainability goals, by addressing problems with emission metrics shared by big corporations, which may not hold legitimacy. Even though regulators exist, there's the understanding that corporations would want to keep their privacy regarding how that data is collected, thus, some suggestions are made in order to reach the desired goal:

- Verifiable data collection gathered from sensor readings using proofs of construction;
- Sensor disclosure of sustainability metrics that preserve privacy;
- Construction of proofs of computation that reveal compliance to manufacturing processes, whilst not exposing sensitive private information;

Essentially, McDaniel is exposing the need for validation of inputs in SMPC and suggesting solutions that can reach that goal without breaking privacy properties.

3.2.2 Private Set Intersection

Camenisch and Zaverucha [52] also address the issue of input validation, albeit in the context of private set intersection through a solution using certified sets.

Private Set Intersection (PSI) is a SMPC technique in which two parties, each holding a private set, compute their intersection while hiding elements not part of the intersection from one another. An example of this technique is the intersection of medical databases, which protects patient information while allowing a better understanding of their illnesses through common symptoms and medical practices.

Through their solution, they protect parties from malicious participants using arbitrary sets. They do this by ensuring inputs are validated by mutually trusted CA which binds them to the participant prior to the computation, ensuring high security and, subsequently, trusted validity.

3.3 Private Set Membership

Private Set Membership (PSM) is a cryptographic protocol that enables a client to verify whether a specific element is part of a set held by a server while preserving the privacy of both parties. In a standard PSM protocol, the client learns only the membership status without gaining any additional information about the dataset. Similarly, the server does not obtain any knowledge about the client's query, ensuring mutual privacy.

Garg et al. [53] introduced Credible Private Set Membership (C-PSM), which enhances traditional PSM protocols by incorporating soundness guarantees. Their approach ensures that a dishonest client cannot manipulate queries to gain additional information about the dataset, addressing a critical limitation of earlier PSM constructions. This advancement strengthens the security model by preventing adversarial misuse while maintaining efficiency.

Existing work in the field of PSI has also contributed to the development of efficient PSM schemes.

3.4 Proof of Provenance

Establishing proofs regarding the origin of data can give credibility to statements in which such data is used, therefore it is crucial in some cases to assert that such data is provided by trusted entities.

Of course, as we have already talked about, digital signatures and certificates exist for this purpose, but Zhang et al. propose a different solution that leverages ZKP over TLS connections to guarantee to some interested party that the provenance of such data is genuine [54].

In their paper, they make two important points:

- Current solutions not only require server-side modifications but are based on undesirable trust assumptions.

- Users should be free to export their data, without help and permission from data holders, with preserved integrity.

TLS, or Transport Layer Security, is a secure communication protocol widely used to establish secure connections between clients and servers, ensuring confidentiality, integrity, and authenticity of data transmitted over the network. Its big limitation, however, is that it doesn't allow a user to prove to third parties that a piece of authentically accessed data came from a particular source, as it can't be exported securely.

To this effect, they propose DECO - Decentralized Oracle - a secure improvement over existing Oracle schemes capable of providing zero-knowledge proofs of provenance over TLS connections, that enable more accessible data without the need for server-side modifications. It can also provide proofs over substrings of session-data commitments which serve substantial efficiency over general ZKP.

3.5 Ring and Group Signature Proof

Ring signature schemes are commonly used to allow a signer to sign a message within a group (or "ring") of multiple members, each possessing their own public key. This ensures the identity of the actual signer remains anonymous while still maintaining the validity of the signature and the authenticity of the signer within a known certified group.

Whistle blowers have been known for using such schemes to demonstrate their belonging within an organization, while remaining anonymous, when revealing sensitive information of said organization. However, because not every member has ready access to a suitable key, and collecting a large enough number of keys may gather unwanted attention — together with the fact that organizations refrain from issuing keys for this purpose — Faz-Hernández et al. [55] propose using existing keys present in hardware security modules.

These modules are supported by standard hardware authenticators such as *WebAuthn*, allowing signatures through standardized algorithms like *ECDSA*. However, these signatures and certificate chains that provide authenticity to the public key used reveal the the identity of the signer. Thus, Faz-Hernández et al., faced with the infeasibility of updating those modules to provide privacy, propose using ZKP of valid ECDSA signatures under a commited public key.

Their proposal ensures that a verifier can check the validity of a signature without knowing the public key associated with it. Also, because this public key is kept secret through the hiding commitment, the proof can show additional properties of the public key, such as its belonging to a of a public list of keys, which constitutes a ring signature (or a group signature).

This work is very similar to ours, in the sense that through the means of a group signature, they can provide a proof of authenticity without revealing the identity of the subject. It differs in the type of signature scheme used, and its context, but nonetheless describes the type of functionality we aimed for. Our initial plan also considered providing proofs regarding personal information through the means of a Signed Identity Document, which is not considered in their proposal, but it proved difficult due to communications necessities and certificate commitments.

4

Intended Plan

The initial plan was to integrate a ZKP protocol capable of verifying the legitimacy of inputs within the STAR protocol. Given our prior experience with STAR, and because this dissertation was done in collaboration with one of its authors, Assistant Professor Alexander Davidson, we considered it a suitable option.

However, due to time constraints and technical limitations, we weren't confident that we could provide input legitimacy with the resources available, and the lack of time to investigate further put us on track to try other implementations while maintaining the core objectives.

This chapter serves as a way to describe the limitations encountered, while providing some possible alternatives that could have been taken to overcome them.

4.1 STAR

Secret Sharing for Private Threshold Aggregation Reporting, or STAR for short, is a Private Threshold Aggregation System of user measurements, and a solution to the Private Heavy-Hitters Problem developed by Brave focused on high efficiency and low cost, all the while leveraging simple and well-established cryptography for easier adoption from developers scattered across a wide range of expertise [56].

Threshold Aggregation Systems are collaborative computations generally used for data collection, in which N parties contribute their inputs or computations to a central entity, who after reaching a threshold of K inputs, computes the aggregation result.

The Private Heavy-Hitters Problem is characterized by the need of gathering the set of all popular values for a certain parameter above a certain threshold, from a pool of clients, without learning anything else about any specific client's parameters. Another solution that addresses this problem is Poplar [57]. Use cases include, for example, the need to know the most commonly searched word within a search engine without learning which searches a client is making.

4.2 Challenges

4.2.1 Use Case

Suppose Jon Doe is an honest individual that holds a card with authentic credentials. He could choose to provide a statement claiming his age to be 30, using his card data as a witness to the proof. To do so, he would need to satisfy all the required relations:

Relation 1. *Statement == TRUE*

Relation 2. *Verify(StatementSignature, Statement, Certificate) == TRUE*

Relation 3. *Verify(Certificate, CA) == TRUE*

All relations should hold whenever the verification algorithm ran, thus providing confidence in Jon's statement. If Jon were to be dishonest, however, or hold invalid credentials, at least one of the relations wouldn't hold, thus allowing the protocol to reject his message.

Yet, looking back, we find this plan very eager. We made assumptions regarding the witnesses that don't hold. Witnesses are supposed to satisfy circuit computations, and since a user can input anything, statements like the one described above only work if the inputs are restricted by a program. In practical terms, if this restriction is broken then any input can be provided to satisfy the statement, regardless of providing a signature - as they can also be provided indiscriminately). This oversight allows an individual to create a valid proof for a false statement.

An alternative would be using a certificate which has the data as well, properly certified by a CA. In this scenario, the individual could present a proof where part of their data is signed by themselves, and, using the provided certificate as a witness, a verifier can determine whether the signed data matches the data presented in the certificate, ensuring individual authenticity, anonymity and input legitimacy within the framework of a trusted CA. Unfortunately we did not find such certificate.

4.2.2 Requirements

Our initial proposal was meant to adhere to certain requirements which were met with challenges that made them difficult to fulfill.

Requirement 1. *It is crucial to ensure, throughout our proposed solution, that the properties of privacy and correctness remain true, regardless of ensuring valid inputs.*

SMPC has its foundations set on two properties: **privacy** and **correctness**, as previously mentioned in 2.1. Thus, requirement 1 was not an issue, as they were mainly already satisfied by STAR and further development would hardly compromise them.

Requirement 2. *It is required to collect the card data through a card reader so that proofs can be built with them as witnesses.*

As noted in 2.2.1, the Cartão de Cidadão smart card holds individual data, certificates from a CA, and a private key in a QSCD requiring PIN authentication, enabling digital signatures. Fullfilling requirement 2 was not an issue, however, using the data as a witness can only go so far as to serve as a proof.

The only way to ensure data legitimacy, without a separate trusted program uninterruptedly feeding the inputs to the ZKP — which could compromise soundness — is by validating a certificate chain attesting to the data, or contacting the CA, which zero-knowledge proofs can not do. Furthermore, in order to pursue these alternatives, access to Agência para a Modernização Administrativa (AMA)'s API is required. However, as of today, our request for access has not been granted.

Requirement 3. *STAR is exempt from any communication done with a third party CA which if needed must be strictly done by the client.*

Because we wished our solution to remain as simple and as efficient as possible, we established requirement 3, to have as minimal impact on the protocol itself. This exemption from communication further limited our ability to validate data authenticity, Nonetheless, if some communication were to be absolutely necessary, it would require a separate protocol working in conjunction with the ZKP.

Requirement 4. *The ZKP must ensure the properties of completeness, soundness, and zero knowledge.*

As outline by requirement 4, the ZKP would need to ensure the properties specified in 2.3. The alternative solutions mentioned before would inevitably compromise the soundness property, and thus false statements regarding data could potentially be evaluated as truthful.

4.2.3 Evaluation and Testing

Because the evaluation and testing of our proposed solution was of utmost importance to ensure all properties were met, we originally decided to do it periodically, as a way to detect flaws in our solution and lead to improvements throughout development. However, our original plans were to use test cards, but the Cartão de Cidadão API does not support their use unless an access key is provided. Till this day, after making our request, we have not been given a response.

5

Change of Direction

This chapter addresses the changes in the dissertation, including explanations along with the rationale for the new approach. We reflect on our choices and the evolution of the study over time.

5.1 Setbacks

Our initial proposal for this dissertation was the implementation of Zero Knowledge Proofs in STAR. The goal was to implement a zero-knowledge proving mechanism to complement the protocol, acting as a middle-man ensuring measurements collected were legitimate. These measurements were to be provided by citizens of Portugal through means of their citizen's card, "Cartão de Cidadão", thus enabling the zero-knowledge proof construction based on their information. However, during research and development, we came across some setbacks:

- The Cartão de Cidadão API provided by AMA is accessible for free, however, the use of their signing algorithms requires credentials that are provided at a cost that is not publicly disclosed. Those credentials were solicited and although it was specified that it was for academic purposes, no response has been given to this date.
- The development of Zero-Knowledge Proofs proved to be a challenging task, despite multiple attempts at different libraries. Most of the materials available for these libraries lack any proper documentation, such that without a proper introduction to them, their use at this level is not ideal or productive.
- The libraries available for zero-knowledge proof production don't contemplate communication and rely solely on inputs provided by the user. This makes it infeasible to verify validity with the CA without disclosing information that might identify the user.

In addition to these setbacks, our original intention was to use Zero-Knowledge Proofs to make statements such as "I'm at least 18 years old" or "I'm a male" using the citizen's

card. The plan was to sign the card's data with the individual's private RSA key and verify that the signature was valid and its original content corresponded to the claimed data. However, this introduced further challenges:

- To verify a signature, the corresponding public key is needed. If the public key is only available through a PKI, the required communication creates a blocking point for proof generation.

- To ensure the signed data is truly the data included on the card, we would either need to enforce that the algorithm used to generate the proof only accepts input from a secure Card Reader and Signer algorithm, or require communication with the card issuer for this guarantee, which again would block proof generation.

In light of the mentioned setbacks, the time we had dedicated to come up with a solution involving STAR became extremely limited, and we believed that further time investment could compromise the core objective of the dissertation.

5.2 Our decision

Because of these challenges and the ones mentioned in chapter 4, in addition to the little time we had in our hands to make implementations, we opted to simplify things by making some assumptions and some changes to the original idea. Faced with the difficult task of guaranteeing that data is in fact the card's data, we decided to scrap this idea (although with a closed system and some verifiable key to indicate that the data came from a secure and trusted program, this would be possible). Instead we considered other important scenarios in which verification is crucial to guarantee the legitimacy of data, other than the card's.

We began thinking then of other useful proofs. One case we found particularly interesting, involving signatures, was providing zero-knowledge proofs of signatures without revealing the users, through group signatures. Doing so would provide authenticity and anonymity.

To ensure the legitimacy of data, signatures are commonly used as a seal of guarantee. Digitally, these signatures are done resorting to RSA signature schemes, which are supported by a Signature Certificate, used to validate the signature. This Signature Certificate is further supported by a chain of Certificates, beginning with the Root Certificate issued by a trusted CA, increasing the certificate's credibility and in turn, the signature's.

Furthermore, to overcome the need for certificate chain verification, we considered a simpler scenario where all public keys are assumed to be trustworthy. This would allow us to bypass the communication issues mentioned earlier and rely solely on public key signature validation.

Assuming we have access to all public keys and we have assurance that they are trustable, then we skip the certificate chain verification in the proof itself and lean solely on the public key's signature validation capability.

This way, if the public keys of all citizen's cards were theoretically made public, we could have them as a public input of the proof, and should verify the signature's validity, meaning that one private key associated with one of the many national public keys was used. This is not just a means to an end, as this is a common and previously discussed problem!

Real life examples of this use case are voting systems where there is the desire to guarantee that each vote is legitimate, but without identifying who the voter is. Or petitions, which require signatures, but may not receive many due to users being hesitant to sign, preferring to preserve their anonymity. These are problems that can be solved using a Group Signature Zero-Knowledge Proof. This problem does not fit the entirety of our initial proposal, but we feel it is still relevant in its context.

These require a guarantee that whoever signed the petition, or voted in an election, is a legitimate individual with that capacity. This can be done through a Group Signature verification, in which we can verify that whoever signed the petition, or voted, is a real individual part of a group of individuals, in this case, all Portuguese citizens.

6

Solution Design

6.1 Context

The main objective of our solution is to provide a mechanism capable of evaluating the authenticity of group signatures through Zero-Knowledge Proofs in conjunction with Signed Identity Documents in an effort to preserve anonymity in sensitive collaborations such as votes and collection of petition signatures, where individuals might desire that their identity remains undisclosed while at the same time have confidence that all other contributions come from verified and credible individuals.

As outlined in chapter 3, Zero-Knowledge Proofs for anonymous identity credentials using identity documents isn't a new idea and some implementations have even been proposed. Moreover, there is valid concern for legitimate inputs in SMPC while preserving privacy, and likewise some proposals have been made to achieve this goal. More specifically, there have been proposals to validate ring signatures using Zero-Knowledge Proofs, allowing a signer to prove the validity of a signature within a group without revealing their identity. One approach uses Zero-Knowledge Proofs for ECDSA signatures under a committed public key, ensuring privacy while verifying authenticity. This aligns with our goal of proving authenticity without revealing identity, though our focus on Signed Identity Documents adds further complexities.

6.2 Requirements

Our solution is based on Cartão de Cidadão signing keys and data, therefore provers participating in the protocol must have their signing keys and signature comply with the card's signing keys, RSA, and signature scheme, PKCS#1 v1.5.

Our Zero-Knowledge Proof must correctly determine if the signature is valid according to the aforementioned scheme, and if the signer's public key belongs to the list of public keys from the group of individuals, in order for it to be valid.

The group of individuals is public and provided by the prover, but the verifier can previously decide on a group to be provided. Apart from the verification of the signer's

inclusion in the group, the circuit does not validate the groups legitimacy or equivalence to a previously established one. This responsibility falls on the verifier, by basing their group on a known verifiable list of public keys from a trusted source, and evaluating if the group is the same as the one they provided, if that is the case.

6.3 Design

Our solution follows 4 disctinct phases, present in figure 6.1 which together provide the desired outcome.

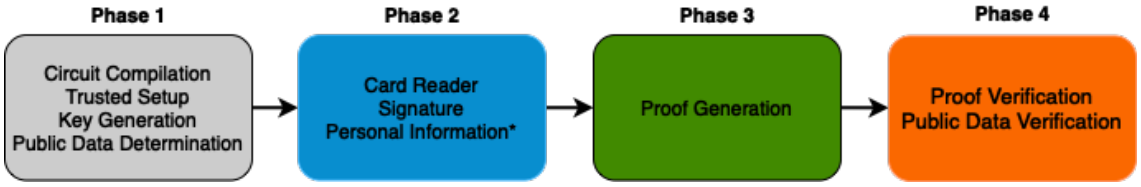

Figure 6.1: Solution Design [1]

The **first phase** occurs after the verifier has conducted the necessary steps to compile their circuit, and generate the corresponding *R1CS* and *Wasm* files, and the proving and verification keys required for the zero-knowledge proof operations. A trusted setup between the verifier and the prover can be done in this phase but is not required due to the aforementioned Perpetual Powers of Tau 2.3.4. The verifier determines the hash(es) to be signed and can also pre-determine the set of individuals that will be part of the group signature. The prover receives these files and information, so that they can generate their proof afterwards.

The **second phase** consists of reading the prover's Cartão de Cidadão through the means of an electronic card reader. By gaining access to the card's information and functions, the prover can perform signatures using their unique signing key on whichever data they desire. They also gain access to the information embedded in the card which can be used as witnesses in other scenarios. This phase provides the necessary inputs for the proof generation: the signature (and potentially the data that is signed).

The **third phase** can only occur after the first and second, as it incorporates the proof generation using the signature and the provided files from the verifier. The prover constructs their proof using a public list of public keys (which can be provided prior by the verifier) from which they belong to. Their signature is provided to the proof generation algorithm as a private input so that it remains secret. After generating the proof using the R1CS, Wasm, proving key, and their input, the proof is sent to the verifier for validation.

The **fourth phase** finalizes the protocol by having the verifier validate the proof provided by the prover. The proof is verified using the verification key, and afterwards the

[1] Although the personal information is not used in our solution specifically, it may be used in other scenarios.

public inputs, and outputs, are verified as well to determine if they have been tampered with. If all is correct, then the verifier can confidently account for the prover's input.

Although this design does not contemplate a masking procedure to hide the identity of the prover during communication, we believe it could be a necessary step unless it is implemented in a SMPC protocol, where their input is untraceable.

6.4 Technologies

For the development of the Card Reader & Signer, we chose to use *Java*. This decision was based on the fact that Java is fully compatible with the Cartão de Cidadão API, making it an ideal choice for interacting with the citizen card system. Java's widespread use, robust documentation, and cross-platform capabilities further contributed to its selection, ensuring that the implementation would be both reliable and easily maintainable.

For the development of the Zero-Knowledge Proof circuit, we opted to use *Circom*. This choice was driven by several factors, including the availability of extensive learning materials and documentation that greatly facilitated the development process. In addition to its ease of use, Circom's growing community and its established reputation in the field of cryptographic circuit design made it a suitable framework for our project.

For the proof generation and verification, we decided to utilize the *Snarkjs* library. This library is well-regarded for its efficiency in generating and verifying zero-knowledge proofs. To streamline the trusted setup process, we incorporated a *Perpetual Powers of Tau* file, which allowed us to simplify the trusted ceremony. This approach not only saved time but also ensured a higher level of trust and security in the cryptographic setup phase, aligning with the project's core goals.

6.5 Threat Model

Our solution leverages Groth-16 and Perpetual Powers of Tau, ensuring zero-knowledge and soundness, while completeness is enforced through additional measures that enforce public input constraints outside the proof system. This means that public inputs and outputs are evaluated by the verifier to detect false proofs.

The Public Keys can be pre-determined and bounded to the proofs as public inputs, ensuring they are only valid if verifiable and their public inputs match the pre-determined ones, ensuring completeness.

Defining the groups as Public Keys, and having them as public inputs, will prevent Unauthorized Participation and only allow valid identities to participate by detecting malicious individuals trying to include themselves in the group.

Even though the Signatures are private inputs, every group signature proof is accompanied by an output corresponding to its MIMCSponge Hash, preventing Replay Attacks (duplicate proof submissions by the same signer) and therefore ensuring correct results.

Our choice for RSA key bit sizes can make our solution affected by future quantum computing advancements, and should instead be increased to 4096 bits. Quantum cryptography advancements must also be monitored as they can affect the zero-knowledge property of our solution.

7

Cartão de Cidadão Data Reader & Signer Implementation

7.1 Context

Our initial idea requires that the Cartão de Cidadão's data is read and signed by the card's holder, in order to provide legitimacy to whichever data is associated with them. The verification of this signature would be done through a CA's PKI which would ensure to anyone that the data was in fact signed by the holder of the card.

Due to some setbacks in development and technology limitation, as already discussed in chapters 4 and 5, the verification could not be done this way, and instead would need to rely on a publicly available list of public keys. Nonetheless we found ways of verifying signatures, albeit without as many safeguards as a PKI would give. We opted to verify using the card's signature certificate as an exercise and as a building block for what is to come. This verification is only limited in two aspects:

- It depends on the card's signature certificate. Verification using the same certificate, without the card, requires it to be publicly available or accessible;
- It does not do a chain verification to determine the validity of the certificate itself, it only focuses on the public key.

Our implementation can be found here [58].

7.2 Implementation

For the development of the card data reader we decided to use Java since it's supported by the available SDK provided by AMA, "**Middleware Oficial de Identificação Eletrónica em Portugal - Cartão de Cidadão, da Chave Móvel Digital e Sistema de Certificação de atributos profissionais**"[59].

To use this SDK it's required that we install the installation package which includes the end user's GUI application. Once installed, the SDK files will be available in your

machine's library. In the case of MacOS, which is the environment in which our program was developed, it is installed inside */usr/local/lib/ptedid_jni*. When using this library with Java, we must include the *pteidlibj.jar* file in our Java build path. This was made easy through Eclipse IDE, where you can easily setup the build path of the program project.

To use this program correctly, a card reader is necessary. In our experiments we used a *TooQ TQR-210B* card reader to successfully read data from a citizen's card. This program does not allow test mode in which test cards can be used, since it requires a paid credential which was never provided to us, so a real card must be used. Additionally, it does not allow "*Chave Móvel Digital*" which is the SMS token mechanism to sign documents. All signatures used in this program require the card's authentication PIN. If a card is not setup to accept this pin, or you haven forgotten it, then the program will fail when trying to sign.

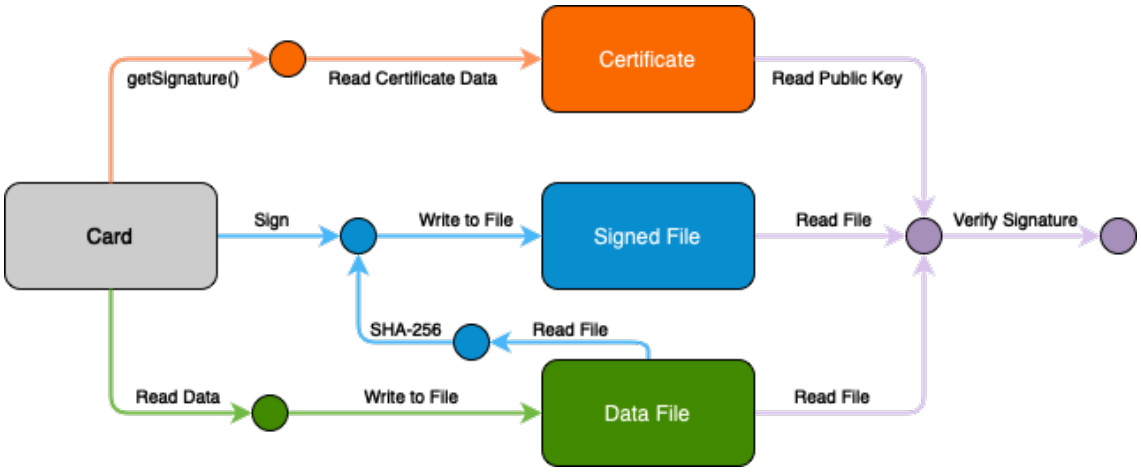

Figure 7.1: Cartão de Cidadão Data Reader & Signer Diagram

When inserting our card in the card reader, and running our program, we call the AMA's **PTEID_ReaderSet** class **instance** method to collect an instance of the card, which is available through the **PTEID_EIDCard** class.

In our case, our card has a 3072-bit RSA public key as opposed to test cards which should have a 2048-bit RSA public key [17] (although we can't say for certain due to lack of testing). Our program only creates signatures using the PKCS#1 v1.5 signature scheme, based on SHA-256 hashes, with the purpose of signing selected data used for attestations.

Ideally a user would select which data from their card they wish to sign, however for simplicity we decided to gather their full name, date of birth and gender. Their data is collected, concatenated in the same order, and saved on a file with UTF-8 encoding. In practice you can sign any data.

After this file is saved, its content is read and hashed with a SHA-256 algorithm. In our case we used **Java Security's Message Digest** class. With this hash now created, we proceed with the signature using the aforementioned AMA's SDK by using its **PTEID_EIDCard**

class. This class, when the card is successfully read, allows you sign SHA-256 hashes, with the **Sign()** method, provided that you know the authentication PIN associated with it. This method takes as parameters the hash byte array to be signed and a boolean flag indicating if it is signing with the card's signature key. When the signing method is called, an instance of the end user's GUI application is launched, requesting the PIN. After inputting the card's PIN, the signature is created using the card's RSA private key, and saved as a UTF-8 encoded file.

Verification of the signed file is done by accessing the card's signature certificate through the **PTEID_EIDCard** class' **getSignature()** method. This method outputs a **PTEID_Certificate** object. By collecting the certificate's data as a byte array, we can create an instance of **Java Security's Certificate** class, in order to access the Certificate's information, which includes the public key. Then, by using an instance of **Java Security's Signature** class configured for **Sha256withRSA**, we initiate verification by providing the certificate's public key, original pre-hash data, and the signature. If the verification is successful an output message is given indicating success, otherwise a failed message is given.

8

RSA Signature Verification Zero Knowledge Proof Implementation

8.1 Context

As mentioned in the previous section, we only verify Cartão de Cidadão's signatures using its Signature Certificate, available trough the card. Thus we do not validate the chain of certificates. With this in mind, we approach signature verification proofs in the same manner, albeit with even less safeguards than the previous method. What we mean by this, is that we rely solely on the signature's Public Key, as we will not have access to the certificate itself. As already mentioned previously, we cannot make use of the CA's PKI to validate signatures because Zero Knowledge Proofs don't support communication.

We propose a solution using the Public Key itself to verify the signature, or a set of Public Keys to verify a group signature. This solution assumes that these Public Keys are accessible, even though in most cases they are not. Nonetheless, this presents an opportunity to experiment and understand if the solution is viable. If the solution ends up being viable, then optimizations could be made to implement it in accordance to existing PKI.

Even though our card has a 3072 bit length RSA keys, we implemented a solution for 2048 bit length RSA keys. We originally planned to use test cards to experiment with our solution, and these do in fact have 2048 bit length RSA keys, but due to some setbacks we were unable to use these cards. Our solution only supports 2048 bit RSA keys but can be altered to support other bit length keys. This however will have implications in execution and verification time.

8.2 Implementation

To implement Zero Knowledge Proofs you must first define a circuit which is the basis for the restraints that the proof will depend on. There are many languages that support circuit definition, but we decided to use Circom since it not only has a lot of documentation available, but workshops as well, which makes it easier to get familiar with.

While researching the topic of RSA Signature Verification proofs we came across an existing circuit [60] written in Circom that verifies 2048-bit RSA PKCS#1 v1.5 signatures. This signature scheme is exactly what the card uses. The only difference is that the card uses 3072-bit keys as opposed to 2048-bit. However, test cards employ 2048-bit keys, and with that in mind, we assumed to be suitable for future testing.

Manipulation of Big Integers in Circom is possible by resorting to Circom's ECDSA Big Int library's operations. Additionally, in our case, PowerMod's template allows us to make modular exponentiation operations such as the ones used by RSA encryption/decryption algorithms.

The following sections describe the details of the implementation:

8.2.1 Input Signals & Template Configuration

To generate the proof, a user must provide the following input signals:

- **Signature** - An array of 32 elements each with 64-bit integers.
- **Public key modulus *N*** - An array of 32 elements each with 64-bit integers.
- **Public key exponent *E*** - An array of 32 elements each with 64-bit integers.
- **SHA-256 hash of original message** - An array of 4 elements each with 64-bit integers.

The reason behind using 64-bit integer arrays is due to Circom's limitation regarding integer representation. Circom inputs must be provided as strings, which when cast as integers, are constrained by the bit length of the curve's prime number, which is 254 bits for alt_nb128 [61]. Thus, since we are working with 2048-bit integers, it is required, beforehand, that they are represented in this manner. These arrays are comprised of 32 64-bit elements, called registers, and are divided using the algorithm in figure 8.1.

Note that bit representation is reversed when converting a 2048-bit integer into an array of 32 64-bit integers, such that the least significant bits are placed in the first position of the array, and the most significant ones in the final position.

```
let result: bigint[32] = [];
let mod = 2^64;
var temp: bigint = input;
for (var i = 0; i < 32; i++):
    result[i] = temp % mod;
    temp = temp / mod;
```

Figure 8.1: Input Signal's Division - 2048-bit Integer represented as an array of 32 64-bit Integers

The signature verification circuit instance must be instantiated with 4 configurable parameters:

- **w** - Determines how many bits the integer registers have - in our case 64.
- **nb** - Determines how many registers are used in the signature and public key's modulus N arrays - in our case 32.
- **e_bits** - Determines how many bits are used by the public key's exponent E - in our case, 17, because we test mainly with 65537.
- **hashLen** - Determines how many registers are used in the hash array - in our case, 4, because we test with SHA256.

These parameters are directly related to the signal inputs acting as bounds within the execution and determining the amount of constraints.

8.2.2 Circuit Execution

Before verifying that the signature is valid according to the provided public key, modular exponentiation of the Signature must be done. This is accomplished through the PowerMod circuit template.

After modular exponentiation, the result must be validated following PKCS#1_v1.5 specifications to ensure that the signature was properly produced and that the hash provided matches the hash present in the output of the modular exponentiation operation. This execution flow, present in figure 8.2, is described in detail in this subsection.

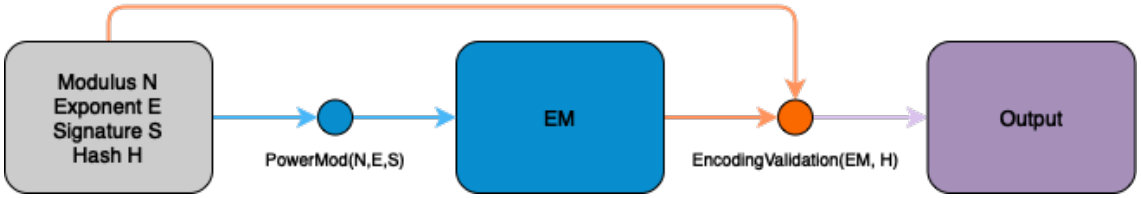

Figure 8.2: RSA Signature Verification Circuit - Execution Flow

8.2.2.1 PowerMod

The RSA Signature Verification circuit's first step is to apply modular exponentiation with our submitted inputs in order to obtain the original encoded message EM:

$$EM = Signature^{E} \mod N$$

To do so, it uses a **PowerMod** template which takes w, nb and e_bits as configurable parameters. Likewise, it takes as input signals the **Signature**, which we will be calling

Base, and the modulus N. The **PowerMod** template considers **E = 65537**, as it is the most used and supported public key exponent. Additionally, the template is a revised implementation from the original that aims to reduce total number of constraints without affecting the result.

Its output is the encoded message **EM** as an array of 32 64-bit integers, as per our chosen parameters.

PowerMod makes use of other circuits from the Circom ECDSA Big Int library. These circuits have been developed and evaluated by the Circom community [1].

Although Big Int circuits are essential to the circuits discussed in this chapter, their implementation is not our primary focus. These circuits handle mathematical operations — such as addition, multiplication, and division of large integers — within already discussed restraints — allowing us to abstract away those details and concentrate on developing and understanding our implementations. While the Big Int circuits do impact the overall solution, we can trust in their optimization, as they have been thoroughly evaluated and refined by the Circom community.

Our **PowerMod** implementation begins by creating an array with a length of **e_bits + 1**, where each element is an instance of the **BigMultModP** template from the Big Int library. **BigMultModP** is the circuit that enables modular multiplication. Instances of circuit templates are called components, and, in this case, take **w** and **nb** as configurable parameters aswell.

BigMultModP has 3 input signals:

1. **p** - The modulus in the modular multiplication operation. It's an array of integers and is assigned the value of the modulus **N** as shown in figure 8.3;
2. **a** - The first value for the multiplication operation and is defined as an array of Big Integers
3. **b** - The second value for the multiplication operation and is defined the same way **a** is.

Modular exponentiation possesses several properties that can be leveraged to optimize calculations:

- **Exponentiation by Squaring** - By decomposing the exponent into a sum of distinct powers of two, we can significantly reduce the time complexity from $O(n)$ to $O(\log n)$, where **n** is the exponent.
- **Modular Reduction During Computation** - Rather than computing $a^b \mod p$ directly, intermediate results of the form a^c and a^d, where $c + d = b$, are reduced modulo **p** during computation. This approach prevents the intermediate values from becoming excessively large. Therefore, if $a^c \mod p = X$, then $X * a^d \mod p$ is congruent to $a^b \mod p$.

[1] Their implementation can be found here

```
component muls[e_bits + 1];

for (var i = 0; i < e_bits + 2; i++):
    muls[i] = BigMultModP(w, nb);

    for (var j = 0; j < nb; j++):
        muls[i].p[j] <== N[j];
```

Figure 8.3: PowerMod Step #1 - BigMultModP instantiation

Recall that the variable **e_bits** is predefined and represents the number of bits used by the value 65537, which is 17. Given that 65537 has only two non-zero bits (the most significant and the least significant), it can be expressed as $\mathbf{65537 = 2^{16} + 1}$. Thus, to compute $\boldsymbol{Base}^{65537}$, we can simplify it as follows:

$$\boldsymbol{Base}^{2^{16}+1} \equiv \boldsymbol{Base}^{2^{16}} \cdot \boldsymbol{Base}$$

To efficiently calculate $\boldsymbol{Base}^{2^{16}}$, we can construct an array $\boldsymbol{a}$ of 17 elements using a geometric progression, where each element a_n is given by $\boldsymbol{a_n = Base}^{2^{n-1}}$. By iterating just 16 times, we can compute $\boldsymbol{Base}^{2^{16}}$ by continually squaring the previous element: $\boldsymbol{a_n = a_{n-1}^2}$.

This array $\boldsymbol{a}$ effectively serves as our collection of **BigMultModP** components, where each result is already reduced modulo **N**. The process can be visualized using the algorithm in Figure 8.4. The first element in this sequence computes **Base** (by multiplying **Base** with 1), and the second element computes $\boldsymbol{Base}^2$. We continue this process, with each component's input being the output of the previous one, to obtain the subsequent squared values, culminating in the value at position **e_bits - 1**, which yields $\boldsymbol{Base}^{2^{16}}$ **mod** $\boldsymbol{N}$.

Finally, by multiplying this last output by **Base** and storing the result in position **e_bits**, we obtain the desired result of $\boldsymbol{Base}^{65537}$ **mod** $\boldsymbol{N}$, which marks the completion of this circuit's execution with the **EM** as output.

8.2.2.2 Encoded Message Validation

The second and final step of the circuit is effectively the validation of the encoded message obtained from the modular exponentiation operation using the Signature and the public key. If the public signature is correct, then EM will be an encoded message following PKCS#1 v1.5 specifications and have a matching hash of the original message.

```
var result_index=0;
var base_index=0;
var muls_index=0;

for(var j = 0; j < nb; j ++):
    if (j == 0):
        muls[0].a[j] <== 1;
    else:
        muls[0].a[j] <== 0;
    muls[0].b[j] <== base[j];

muls_index++;

for (var j = 0; j < nb; j++):
    muls[muls_index].a[j] <== base[j];
    muls[muls_index].b[j] <== base[j];

base_index = muls_index;
muls_index++;

for (var i = muls_index; i<e_bits; i++):
    for (var j = 0; j < nb; j++):
        muls[muls_index].a[j] <== muls[base_index].out[j];
        muls[muls_index].b[j] <== muls[base_index].out[j];
    base_index = muls_index;
    muls_index++;

for(var j = 0; j < nb; j++)
    muls[muls_index].a[j] <== muls[0].out[j];
    muls[muls_index].b[j] <== muls[base_index].out[j];

result_index = muls_index;

for (var i = 0; i < nb; i++):
    out[i] <== muls[result_index].out[i];
```

Figure 8.4: PowerMod Step #2 - BigMultModP Exponentiation by Squaring

Recall from 2.4.1.1 that an encoded message EM following EMSA-PKCS1-v1_5 specifications must have an octet-string representation matching

$$\textbf{0x00} \,||\, \textbf{0x01} \,||\, \textbf{PS} \,||\, \textbf{0x00} \,||\, \textbf{T}.$$

EM has a total of 2048 bits because of the RSA key length being used. This corresponds to 256 octet strings, which are divided in the following manner:

1. **0x00** || **0x01** is a fixed prefix made up of 2 octet strings used to identify PKCS#1 v1.5 padding. Likewise, **0x00** is the single separator octet string to mark the end of the padding.

2. **T** is the ASN.1 DER encoding for SHA-256, which is made up of 51 octet strings, of which 32 correspond to the hash, **H**, itself.

3. **PS** octet-length is calculated using the expression $\boldsymbol{Length(EM) - Length(T) - 3}$ which results in 202 octet strings.

As already noted in 8.2.1, the Big Integer array representation being reversed means that its first element corresponds to the 2048-bit integer's least significant bits.

Given this understanding, we can verify the signature's validity by iterating through each of the 32 elements of PowerMod's 64-bit integer array output, which correspond to 8-octet strings, and comparing their values to the expected ones.

1. The octet string representation for SHA-256 ASN.1 DER encoding is:

   0x30 0x31 0x30 || **0x0d 0x06 0x09 0x60 0x86 0x48 0x01 0x65** ||
   0x03 0x04 0x02 0x01 0x05 0x00 0x04 0x20 || **H**

   We can group 8 octet strings to represent an 8-octet string (64-bit integers), from right to left. By doing so we get a new representation:

   0x30 0x31 0x30 || **938447882527703397** || **217300885422736416** || **H**

   To get the remaining 5 octet strings to group with **0x30 0x31 0x30**, if we account for **PS** || **0x00**

   0xFF 0xFF 0xFF 0xFF 0x00 0x30 0x31 0x30 ||
   938447882527703397 || **217300885422736416** || **H**

   we can effectively create an 8-octet string, and complete the representation:

   18446744069417742640 || **938447882527703397** || **217300885422736416** || **H**

   Since we used 4 **0xFF** octet strings, we only have 198 to spare.

2. Before getting the remaining **PS**'s 8-octet string decimal representation, let's first get the decimal representation of the first 8-octet string:

   0x00 0x01 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

   We borrowed from **PS** another 6 octet strings, reducing the remaining number of **PS** to 192. With these octet strings we get the first 8-octet string's decimal representation:

   562949953421311

   So far **EM**'s 256-octet string decimal representation is

   562949953421311 || **0xFF * 192** || **18446744069417742640** ||
   938447882527703397 || **217300885422736416** || **H**

3. Since **192/8 = 24**, we can account for 24 8-octet strings, made up of **0xFF** octet strings. This evaluates to a decimal representation, for each, of

   18446744073709551615

4. We know now what our final decimal representation is:

   562949953421311 || **18446744073709551615 * 24** || **18446744069417742640** ||
   938447882527703397 || **217300885422736416** || **H**

The SHA-256 hash input signal is an array made up of 4 64-bit integers, where the first element corresponds to the hash's least significant bits, and the last to the most significant. We know that the encoding follows this same order, therefore comparing the last 4 8-octet strings of EM to the 4 8-octet strings of the hash array should match if the signature is valid. An iteration is made through the **PowerMod** output array as presented in figure 8.5 to validate the encoding, including the hash.

The validation of the encoding is made using the circuit template **IsEqual** from CircomLib [2]. It takes two inputs and provides their subtraction to another circuit template **IsZero** from the same library. This template outputs 1 if the subtraction is indeed zero, and outputs 0 otherwise. The use of these libraries are necessary as they provide a means of comparing two values in Circom, optimally, and obtain a statement regarding their differences to use in restraint construction.

In our implementation we have 32 instances of the **IsEqual** template to match each element of **PowerMod**'s output to its apropriate 8-octet string in the encoding message. If the output is equal to the encoding all instances should evaluate to 1, thus the sum of all these evaluations must be 32. Any deviation from this number means that the signature either isn't valid for the given public key, the given SHA-256 hash or the encoding following PKCS#1_v1.5 specifications was done improperly. This final validation is done as presented in figure 8.6. If the signature is valid, our output returns 1, otherwise it returns 0.

[2] Its implementation can be found here

```
signal powermodOutput: bigint[32] = PowerMod.output;
signal input hashed: bigint[4];
var hashPrefix[3]: bigint[3] =
    [217300885422736416, 93847882527703397, 18446744069417742640];
var pm: bigint = 18446744073709551615;
var paddingStart: bigint = 562949953421311;
component isEqual[32]; signal equal[32];
for (var i = 0; i < 4; i++): //Check hash
    isEqual[i] = IsEqual();
    isEqual[i].in[0] <== powermodOutput[i];
    isEqual[i].in[1] <== hashed[i];
    equal[i] <== isEqual[i].out;
for (var i = 4; i<7; i++): //Check hash prefix and separator byte
        isEqual[i] = IsEqual();
        isEqual[i].in[0] <== powermodOutput[i];
        isEqual[i].in[1] <== hashPrefix[i-4];
        equal[i] <== isEqual[i].out;
for (var i = 7; i < 31; i++): //Check PM for 24 8-octet strings
        isEqual[i] = IsEqual();
        isEqual[i].in[0] <== powermodOutput[i];
        isEqual[i].in[1] <== pm;
        equal[i] <== isEqual[i].out;
isEqual[31] = IsEqual(); //Check padding start
isEqual[31].in[0] <== powermodOutput[31];
isEqual[31].in[1] <== paddingStart;
equal[31] <== isEqual[31].out;
```

Figure 8.5: Encoding validation

```
signal interm: int[32];
interm[0] <== equal[0];
for (var i=1; i<31; i++):
    interm[i] <== interm[i-1]+equal[i];
interm[31] <== interm[30]+equal[31];
component signatureVerifies = IsEqual();
signatureVerifies.in[0] <== interm[31];
signatureVerifies.in[1] <== 32;
out <== signatureVerifies.out;
```

Figure 8.6: Final encoding validation

9

RSA Group Signature Verification Zero Knowledge Proof Implementation

9.1 Context

As mentioned in the previous chapter, by relying solely on a signature's Public Key, we can verify signature validity to a certain extent, since ZKP don't support communication required for certificate validation. We propose a solution in which we verify a signature using a Public Key which we assume to be accessible, and we also propose a way of verifying signatures from a set of Public Keys. This is what is called a Group Signature, in which a signature is verified using a Public Key from a known set, without disclosing which. As mentioned in the prior chapter, this solution operates under the assumption that these Public Keys are accessible, even though they typically are not (in this manner). However, this provides an opportunity to experiment and assess the feasibility of the solution. If the approach proves to be viable, further optimizations can be pursued to align it with existing PKI standards.

9.2 Implementation

We developed two implementations of RSA Group Signature Verification ZKP in Circom, using the RSA Signature Verification circuit discussed in the previous chapter.

Both our implementations achieve the same outcome, albeit with differences in constraint numbers which affect security and feasibility. On one hand, one implementation can be considered as more secure but less feasible due to high number of constraints, causing long execution time and storage used. On the other hand, the other implementation can be considered less secure but more feasible due to the reduced constraints which allow for faster execution times and lower storage needs.

We call the first "Strong" and the other "Efficient", and we present both implementations in the following sections, discussing their properties here and in more depth in chapter 10. Our implementations can be found here [62].

9.3 RSA Group Signature Verification - Strong

This implementation is called "Strong" because it builds a proof by iterating through each Public Key of the Group, and verifying the signature accordingly. It checks if at least one Public Key has verified the signature successfully, and if so it constitutes a valid proof.

It does not need to know the exact Public Key associated with the signature, and because it verifies with each key, the number of constraints grows linearly with a significantly greater slope.

9.3.1 Input Signals & Template Configuration

To generate the proof, a user must provide the following input signals:

- **Public keys** - A 3 dimensional structure representing the Public Keys. Its first dimension has a length of *npk* which is the number of keys present. The second dimension has a length of 2 to differentiate the exponent *E* and modulus *N*. The third dimension has a length of 32 which is the number of 64-bit integers used by the E and N arrays.
- **Signature** - An array of 32 elements each with 64-bit integers.
- **SHA-256 hash of original message** - An array of 4 elements each with 64-bit integers.

The circuit instance must be instantiated with 4 configurable parameters:

- **w** - Determines how many bits the integer registers have - in our case 64.
- **nb** - Determines how many registers are used in the signature and public key's modulus N arrays - in our case 32.
- **e_bits** - Determines how many bits are used by the public keys' exponent E - in our case, 17, because we test mainly with 65537.
- **hashLen** - Determines how many registers are used in the hash array - in our case, 4, because we test with SHA256.
- **npk** - Determines the number of public keys present in the group.

9.3.2 Circuit Execution

For each Public Key provided, an instance of the RSA Signature Verification circuit, already described in chapter 8, is created using the aforementioned configurable parameters. Each of these instances is stored in an array called *Verifications*, and receives one public key, the signature and the hash. Each of these instances' output can be obtained, using their index, by accessing *Verifications[index].out*.

The output of each signature verification is either 0 or 1, meaning that the signature is invalid or valid, respectively. Thus we can determine that one Public Key has verified the signature if the sum of all outputs equals 1.

Since the Public Keys will be public inputs, and the signature will be a private input (to preserve the identity of the public key holder) we decided to output a unique hash of the signature using *MiMCSponge*, a circuit from *Circomlib* which will enable us to determine if a valid signature has already been submitted, so that the same holder doesn't provide multiple proofs for the same hash.

This execution flow, present in figure 9.1 is described in detail in this subsection.

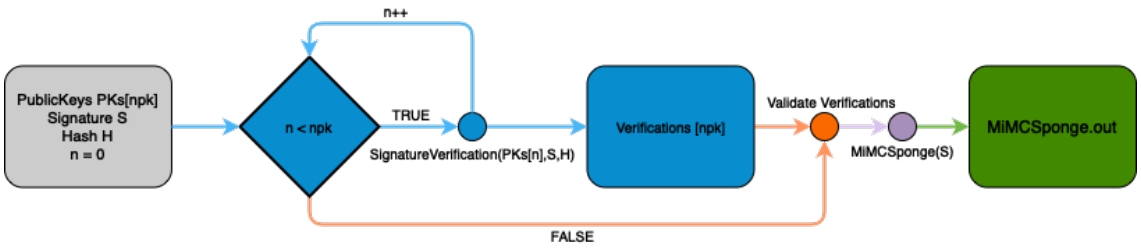

Figure 9.1: Group RSA Signature Verification Circuit "Strong" - Execution Flow

9.3.2.1 RSA Signature Verifications

The Strong Group Signature Verification circuit's first step is to determine for each public key if it can validate the signature. For that effect, it uses RSA Signature Verification instances, reducing the amount of code required in a single circuit, and stores them inside a component array called *Verifications* for later validation, as shown in figure 9.2.

```
signal input publicKeys[npk][2][nb];
signal input sign[nb];
signal input hashed[hashLen];
signal output out;
component verifications[npk];
for (var i = 0; i<npk; i++):
    verifications[i] = RSAVerification(w, nb, e_bits, hashLen);
    verifications[i].exp <== publicKeys[i][0];
    verifications[i].modulus <== publicKeys[i][1];
    verifications[i].sign <== sign;
    verifications[i].hashed <== hashed;
```

Figure 9.2: Strong - RSA Signature Verification Instantiation

After all instances have been initiated, their output is read and assigned to an intermediate signal array which will hold the sum of all outputs in its final position, as shown in figure 9.3. This final position's value is evaluated and if equal to 1 then the constraint is satisfied, meaning that the group signature is valid.

```
signal interm[npk];
interm[0] <== verifications[0].out;

for (var i = 1; i<npk; i++):
    interm[i] <== interm[i-1]+verifications[i].out;

interm[npk-1] === 1;
```

Figure 9.3: Strong - RSA Group Signature Verification

9.3.2.2 MiMCSponge

After verifying the signatures, a *MiMCSponge* instance is created, configured to accept 64-bit integers and perform 220 rounds. MiMCSponge is based on *MiMC* [63] which is a block cipher and hash function family designed specifically for SNARK applications. It's available as a circuit from *Circomlib*.

The MiMCSponge instance takes as input the signature and outputs its hash, as presented in figure 9.4. We output this hash as well, to determine if signatures aren't being provided more than once, as they are kept secret.

```
component mimc = MiMCSponge(nb, 220, 1);

for (var i = 0; i<nb; i++) :
    mimc.ins[i] <== sign[i];

mimc.k <== 0;
out <== mimc.outs[0];
```

Figure 9.4: Strong - MiMCSponge

9.4 RSA Group Signature Verification - Efficient

This implementation is called "Efficient" because unlike the implementation described in the previous section 9.3, it's not built by iterating through each Public Key and verifying the signature accordingly.

This implementation receives two additional secret input signals which are the exponent and the modulus of the Public Key used to verify the Signature.

Upon valid verification, we simply check if the provided Public Key is part of the provided Public Keys set, which is public. Because of this, the number of constraints grows linearly with a significantly smaller slope, as opposed to the Strong implementation.

9.4.1 Input Signals & Template Configuration

To generate the proof, a user must provide the following input signals:

- **Public keys** - A 3-dimensional structure representing the Public Keys. Its first dimension has a length of *npk* which is the number of keys present. The second dimension has a length of 2 to differentiate the exponent *E* and modulus *N*. The third dimension has a length of 32 which is the number of 64-bit integers used by the E and N arrays.
- **Signature** - An array of 32 elements each with 64-bit integers.
- **SHA-256 hash of original message** - An array of 4 elements each with 64-bit integers.
- **Public Key** - A 2-dimensional structure representing the Public Key used to verify the Signature. Its first dimension has a length of 2, used to differentiate the exponent E and modulus N. The second dimension is a 64-bit integer array of 32 elements used to represent E and N.

The circuit instance must be instantiated with 4 configurable parameters:

- **w** - Determines how many bits the integer registers have - in our case 64.
- **nb** - Determines how many registers are used in the signature and public key's modulus N arrays - in our case 32.
- **e_bits** - Determines how many bits are used by the public keys' exponent E - in our case, 17, because we test mainly with 65537.
- **hashLen** - Determines how many registers are used in the hash array - in our case, 4, because we test with SHA256.
- **npk** - Determines the number of public keys present in the group.

9.4.2 Circuit Execution

This circuit creates an instance of an RSA Signature Verification circuit slightly different from the one mentioned in the previous chapter, which will be explained in this subsection.

This instance is created with the same configurable parameters, and verifies that the signature provided is verifiable when using the provided Public Key.

Afterwards, for each Public Key, we create an instance of another circuit, *ComparePublicKeys*, which we will also describe in detail in this subsection.

These instances are stored in an array called *Verifications*, and are responsible for verifying if the given secret Public Key matches another Public Key. Their output is 1 if the public keys match and 0 otherwise. Thus we can determine if the given secret public key is part of the known array of public keys, if the sum of all outputs is equal to 1.

Since the Public Key used to sign is kept secret, and the signature as well (to preserve the identity of the signer) we decided to output a unique hash of the signature using MiMCSponge the same way the Strong implementation does, do deter people from submitting multiple proofs using the same signature, and to make it easier to distinguish different ones.

This execution flow is present in figure 9.5 and described in detail in this subsection.

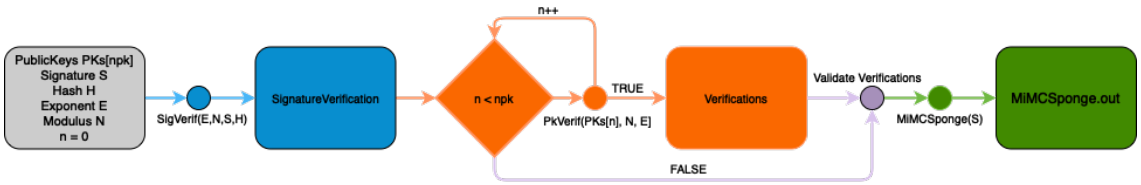

Figure 9.5: Group RSA Signature Verification Circuit "Efficient" - Execution Flow

9.4.2.1 RSA Signature Verification

The Efficient Group Signature Verification circuit's first step is to determine if the Public Key provided can validate the signature.

The previously discussed RSA Signature Verification circuit from chapter 8, which is also used in the Strong Group Signature Verification implementation, makes use of the *IsEqual* template to compare the *EM* obtained from *PowerMod* to the expected EM.

It goes over the EM array iteratively comparing each 64-bit element to the expected value, and by doing so it ends up storing a sum of all IsEqual outputs. This sum, if equal to 32, makes it so that the circuit outputs 1 (in case of a valid signature) or 0 (otherwise). However, this makes it so that running that circuit on its own will always constitute a valid proof, as no constraint is being made regarding that output (i.e. out === 1). This is a design choice made contemplating the execution of the Strong implementation, hence this constraint is only made when in conjunction with it.

This slightly different RSA Signature Verification implementation, which we will call "Fast" RSA Signature Verification, is meant to be used standalone and in conjunction with the Efficient implementation, as it only verifies the signature with one public key. Because of this, it does not require the IsEqual templates, and directly makes constraints regarding the value of each element of the EM output, as seen in figure 9.6. By doing so, any mismatch will result in failed proof immediately. If this circuit is successful, then the Efficient circuit will go on as expected.

Because this circuit uses less templates, and therefore less constraints in total, it is considered a faster implementation of the RSA Signature Verification circuit discussed previously.

```
signal powermodOutput: bigint[32] = PowerMod.output;
signal input hashed: bigint[4];

powermodOutput[0] === hashed[0];
powermodOutput[1] === hashed[1];
powermodOutput[2] === hashed[2];
powermodOutput[3] === hashed[3];

powermodOutput[4] === 217300885422736416;
powermodOutput[5] === 938447882527703397;
powermodOutput[6] === 18446744069417742640;
powermodOutput[7..30] === 18446744073709551615;

powermodOutput[31] === 562949953421311;
```

Figure 9.6: Efficient - Fast RSA Signature Verification

9.4.2.2 Public Key Comparison

The circuit's second step is to determine if the secret Public Key is part of the public set of Public Keys.

The previous implementation does not need this, because this validation is already contemplated when using the RSA Signature Verification circuit.

Our RSA Signature Verification, Fast, does not contemplate this and so we need to verify it using another strategy. That strategy is comparing the secret public key with every public key from the group, which is public.

To do so, we use a circuit called *ComparePublicKeys* which we have developed. This circuit takes as configurable parameters w and nb, just like we have done so far for every other circuit, to define the representation of our large integers.

After taking as input signals the exponents and the moduli of the public keys it's going to compare, it proceeds to create $nb * 2$ instances of the IsEqual template to compare each element of the nb arrays. The modulus comparisons are stored in a nb-length array named *modulusEquals* and the exponent comparisons are stored in a nb-length array named *expEquals*, as shown in figure 9.7.

```
signal input modulus_1[nb]; signal input modulus_2[nb];
signal input exp_1[nb]; signal input exp_2[nb];
signal output out;
component modulusEquals[nb]; component expEquals[nb];

for (var i = 0; i<nb; i++):
    modulusEquals[i] = IsEqual();
    modulusEquals[i].in[0] <== modulus_1[i];
    modulusEquals[i].in[1] <== modulus_2[i];

for (var i = 0; i<nb; i++):
    expEquals[i] = IsEqual();
    expEquals[i].in[0] <== exp_1[i];
    expEquals[i].in[1] <== exp_2[i];
```

Figure 9.7: Public Key Comparison - IsEqual instantiation

Afterwards, the comparisons are evaluated and stored in two intermediate array signals, one for the exponent and the other for the modulus, which maintain the cumulative sum of the corresponding comparison outputs up until the respective element, as seen in figure 9.8.

This means that the last element of both arrays will have a value of nb if all registers match for both keys. By creating two instances of the IsEqual template, we can effectively check if that is the case both for the exponent and the modulus. If the sum of both IsEqual outputs is 2, then both the exponents and the moduli match. We output 1 if that's the case, and 0 otherwise, ending the ComparePublicKeys circuit, as seen in figure 9.9.

Returning to the Efficient RSA Group Signature circuit, an instance of the Compare-PublicKeys circuit is created for every public key present in the group, comparing it with the secret Public Key. These instances are stored in an array called *Verifications* to gather their outputs more easily.

Then, by using an intermediate signal array, we once more make an accumulative sum for each output of the Verifications array, effectively determining at the last position if the secret Public Key is part of the group. If the value of the last element is 1, then a match was found and we can be assured that the group signature is valid. Otherwise, we know that the signer isn't part of the group.

```
signal modulusInterm[nb]; signal expInterm[nb];

modulusInterm[0] <== modulusEquals[0].out;
for (var i=1; i<nb-1; i++):
    modulusInterm[i] <== modulusInterm[i-1]+modulusEquals[i].out;
modulusInterm[nb-1] <== modulusInterm[nb-2]+modulusEquals[nb-1].out;

expInterm[0] <== expEquals[0].out;
for (var i=1; i<nb-1; i++):
    expInterm[i] <== expInterm[i-1]+expEquals[i].out;
expInterm[nb-1] <== expInterm[nb-2]+expEquals[nb-1].out;
```

Figure 9.8: Public Key Comparison - Cumulative Sum

```
component modulusVerifies = IsEqual();
modulusVerifies.in[0] <== modulusInterm[nb-1];
modulusVerifies.in[1] <== nb;

component expVerifies = IsEqual();
expVerifies.in[0] <== expInterm[nb-1];
expVerifies.in[1] <== nb;

signal total <== modulusVerifies.out + expVerifies.out;

component outVerifies = IsEqual();
outVerifies.in[0] <== total;
outVerifies.in[1] <== 2;

out <== outVerifies.out;
```

Figure 9.9: Public Key Comparison - Verification

This verification algorithm can be found in figure 9.10 where we verify, through the IsEqual template, if the accumulative sum is 1 on the last element of the Verifications array.

```
component verifications[npk];
for (var i=0; i<npk; i++):
    verifications[i] = ComparePublicKeys(w, nb);
    verifications[i].exp_1 <== publicKey[0];
    verifications[i].modulus_1 <== publicKey[1];
    verifications[i].exp_2 <== publicKeys[i][0];
    verifications[i].modulus_2 <== publicKeys[i][1];

signal interm[npk];
interm[0] <== verifications[0].out;
    for (var i=1; i<npk-1; i++):
        interm[i] <== interm[i-1]+verifications[i].out;
interm[npk-1] <== interm[npk-2]+verifications[npk-1].out;

component groupVerifies = IsEqual();
groupVerifies.in[0] <== interm[npk-1];
groupVerifies.in[1] <== 1;
groupVerifies.out === 1;
```

Figure 9.10: Efficient - Group Verification

This algorithm is essential to determine if the public key associated with the signature is part of the known group. Otherwise, if we don't do this check, as long as the signature is valid, the prover could be from any group or none at all.

9.4.2.3 MiMCSponge

Just like the Strong implementation, an hash of the signature is created using MiMCSponge which is later output to easily determine if the proof provided is not a duplicate and unique. The implementation of this signature has been already described in figure 9.4.

10

Evaluation and Results

This chapter describes the evaluation methods used and the platforms in which they were conducted. Afterwards, it goes over the results obtained, proceeding with their analysis and key observations. Finally, we discuss the significance of the results and their implications.

10.1 Evaluation Methods

To assess the requirements of our solution, particularly for proof generation and validation, we conducted benchmarks on two distinct platforms to evaluate their feasibility on modern home computers, similar to those used by the provers. Our two benchmarks were performed using two platforms:

- Windows Subsystem for Linux 2 (WSL2) with Ubuntu 22.04.3 LTS on Windows 11, featuring an 8-core Ryzen 7 7800X3D CPU and 32GB of DDR5 RAM.
- MacOS Sonoma 14.4.1 featuring an 8-core Apple Silicon M1 processor and 16GB of LPDDR4X RAM.

We evaluated the performance of five operations directly associated with proof generation and verification:

- The circuit compilation (done by the verifier) involves the generation of the R1CS and *Wasm* files (and the witness generation scripts). The R1CS file is used in the construction of the proving and verifying keys (by the verifier) and the Wasm file is used in the proof generation (by the prover).
- The circuit-specific phase 2 of the trusted setup (done by the verifier) involves the generation of the proving and verifying keys. It requires the phase 1 Powers of Tau file and the R1CS file, along with a random factor contribution.
- The witness generation (done by the prover) requires the Wasm file mentioned previously and the inputs that satisfy the constraints, producing the Witness (*wtns*) file.

- The proof construction (done by the prover) requires the Witness file and the proving key, producing the proof along with the public inputs.
- The proof verification (done by the verifier) requires the verifying key, the proof, and the public inputs

The evaluation of these operations was conducted for both the Efficient and the Strong circuits, to make comparisons on how they perform against each other. Additionally, we varied the group size in the group signature to assess the limitations and, based on the results, theorized about the progression of requirements for larger group sizes, even though these larger sizes were not directly tested. We conducted tests with the following circuits and group sizes:

- Efficient - 50/500/5000/500000
- Strong - 5/10/20

The reason for the difference in group sizes is due to resource exhaustion. The Strong circuit exhausted the systems resources early on in the evaluations. Therefore we decided to decrease the group sizes and compare how shorter group sizes in this circuit perform versus the larger group sizes from the Efficient Circuit.

We analyzed key benchmarks, including execution time, memory usage, and storage requirements, to assess both the feasibility and potential limitations of the solution in modern home computers.

10.2 Setup

This section describes how we setup our tests, to ensure they are reproducible by other people and for future reference.

10.2.1 Proof Input

To properly test our solution we developed some notebooks written in Python to generate a varying number of RSA Keys used in the group. We also provide functions to sign any chosen data and to output all the circuit's necessary inputs in the required format.

There are a total of 3 notebooks, which can be found here [64], and they are described as follows: .

- RSA PKCS#1 v1.5 Sign - Generates (or reads) a RSA key used to sign a certain given input using the PKCS#1 v1.5 scheme. It generates a file with the public key information, signature and hash as big integers.
- RSA Generate - Generates a file with a list of a chosen number of RSA public keys, including a given public key corresponding to the previously mentioned signature

used for the proof input. It also enables you to create placeholder values for testing purposes, since generating a large number of RSA keys can be resource-intensive.

- Circom Input Builder - Reads a file with a number of RSA keys, and another file with the signature and the hash, and produces the proof input file, properly formatted.

We developed and ran these Python scripts using Python 3.8.19 and *Pycryptodome* 3.20.0 library on an Anaconda environment.

10.2.2 Circuit Compilation

We provide a ready to use shell script that compiles the circuits and generates an additional output file with the execution time and memory used, for easier automation and benchmark reporting. The *circomcompile.sh* script is available here [62], and can be executed with the command present in figure 10.1:

```
sh circomcompile.sh {.circom file location} {output directory}
2>&1 | tee {output directory}/circomCompileLog.txt
```

Figure 10.1: Circuit Compilation - Script execution

This script outputs the R1CS file, the Wasm file and the Javascript script required to compute the witness. It also outputs the log of the compilation, reflecting execution time and memory usage, and the *computewitness.sh* script used to facilitate the witness creation.

10.2.3 Trusted Setup Phase 2

To generate the proving (*.zkey*) and verifying (*.json*) keys we must execute the phase 2 of Powers of Tau cerimony, which is circuit-specific. For this effect, we provide the *ppowersoftau.sh* script which takes the R1CS file and an appropriate *.ptau* file, which should be the smallest ptau file with a number of points greater than the number of constraints of the R1CS file. This script outputs an additional file containing the logs of the computation, along with an execution time and memory usage report. It can be executed in the following manner, as presented in figure 10.2:

10.2.4 Witness Generation

To generate the witness we use the previously generated *computewitness.sh* script, which makes use of the Javascript script to generate the witness, the Wasm file and the provided input (which must be present in the same subdirectory as a *input.json* file). Finally, it outputs a *.wtns* file used in the proof generation. Additionally, we output a log file

```
sh ppowersoftau.sh {.r1cs file location} {.ptau file location}
{output directory} 2>&1 | tee {output directory}/ptauPhase2Log.txt
```

Figure 10.2: Trusted Setup Phase 2 - Script execution

containing the execution time and memory usage. The proper use of the shell script is as shown in figure 10.3:

```
sh computewitness.sh 2>&1 | tee {output directory}/computeWtnsLog.txt
```

Figure 10.3: Witness Generation - Script execution

10.2.5 Proof Construction

To generate the proof we just provide the proving key and the witness, and by executing the command from 10.4 we obtain the *proof.json* and *public.json* files. We also get the log containing the execution time along with the memory usage.

```
usr/bin/time -v node $(which snarkjs) groth16 prove
{.zkey file} {.wtns file} proof.json public.json
2>&1 | tee {output directory}/proofGeneration.txt
```

Figure 10.4: Proof Construction - Command execution

10.2.6 Proof Verification

To verify the proof we provide the aforementioned public.json and proof.json files, along with the verification key. As always we return an additional output file with metrics regarding execution time and memory usage. To verify the proof we execute the command shown in figure 10.5.

```
usr/bin/time -v node $(which snarkjs) groth16 verify
{verification key .json file} public.json proof.json
2>&1 | tee {output directory}/proofVerification.txt
```

Figure 10.5: Proof Verification - Command execution

10.2.7 Additional Instructions

10.2.7.1 Memory & Swap

Because some executions require more memory than others, some steps had to be taken to increase the capacity of our systems. As such, we first started by creating a *Swap* partition, whose size varies from 32GB to 128 GB. We done so using the commands present in figure 10.6.

```
$ sudo fallocate -l 32G /swapfile
$ sudo chmod 600 /swapfile
$ sudo mkswap /swapfile
$ sudo swapon /swapfile
```

Figure 10.6: Swap Partition Allocation - Command executions

In some instances we had to increase the the maximum number of memory-mapped areas a process can have. Memory-mapped areas are used by applications to map files into memory for faster access. We did so using the command from figure 10.7.

```
sysctl -w vm.max_map_count=655300
```

Figure 10.7: Increase Memory-Mapped Areas - Command execution

10.2.7.2 Node

To further optimize the performance of Node you can follow the instructions presented here.

10.2.7.3 Top

In some cases the *time* (*gtime* for MacOS) command used to gather metrics only displays the total reserved memory used (RAM), not accounting for the *Swap* partition usage. Therefore, in order to properly evaluate the memory consumption of the processes running on our machines, we often resorted to the *top* terminal command 10.8 running alongside them.

```
# MacOS
$ top -stats pid,command,cpu,time,vsize,rsize
# Linux
$ top
```

Figure 10.8: Top - Command execution

10.3 Results

In this section we present the results obtained from the metric gathering of all previously discussed operations. Each operation is accompanied its results in the form of a table and a bar graphic, illustrating differences across circuits and systems. We include observations and discussions regarding each result.

10.3.1 Circuit Compilation

The circuit compilation was conducted using a single machine (our most powerful one) as it isn't the main focus of the evaluation, since the burden of this operation is placed on the verifier. Thus, we conducted this operation using the Ryzen 7 7800X3D, with its 32GB of RAM. The results of the evaluation are present in table 10.1, which includes the number of constraints the circuit has, the time taken to finish the operation execution, the memory usage [1] and the size of the output files.

We include 3 graphs, one for comparing the constraint sizes 10.9, another for comparing compilation times 10.10 and finally one to compare memory usage 10.11, across the different circuits.

10.3.1.1 Observations

Constraints As can be observed in 10.9, and supported by table 10.1, for each circuit, the number of constraints increases as the number of the group increases as well. Both circuits exhibit a linear relationship between the growth of the group size and the increase

[1] Memory usage includes Swap partition usage

CPU	RAM (GB)	Circuits	Group Size	Constraints	Time (s)	Memory (GB)	Output Size (GB)
Ryzen 7 7800X3D	32GB	Efficient	50	535 807	10.09	0.46	0.13
			500	596 107	11.66	0.54	0.14
			5k	1 199 107	29.45	2.43	0.26
			50k	7 229 107	243.63	23.07	1.42
		Strong	5	2 783 170	27.00	1.59	0.48
			10	5 545 190	53.43	3.13	1.03
			20	11 069 230	107.60	6.20	2.05

Table 10.1: Circuit Compilation - Number of constraints, execution time, memory usage and output size

in the number of constraints. However, the Efficient circuit has a higher constant term compared to the Strong circuit, which results in an apparent exponential rise in graph 10.9. Note that, even though the group sizes are significantly smaller, the Strong circuit's constraint sizes are significantly bigger than the Efficient's.

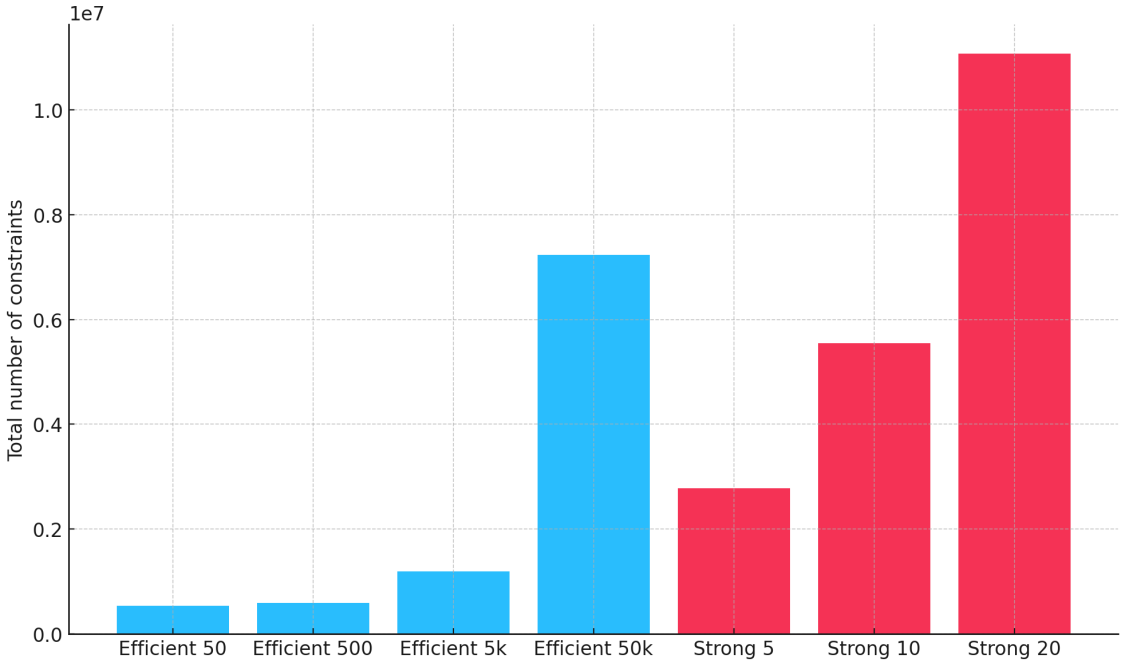

Figure 10.9: Constraints per Circuit

Time As the constraint size increases, the compilation time becomes longer, which is considered undesirable in comparison to shorter compilation times. Even though the Strong circuit's compilation time appears to exhibit a linear relationship with the constraint sizes, the same can not be said for the Efficient Circuit. Note that, despite having significantly lower group sizes, the Strong circuit surpasses the Efficient circuit's larger sizes in relation to the execution time, as evidenced in graph 10.10.

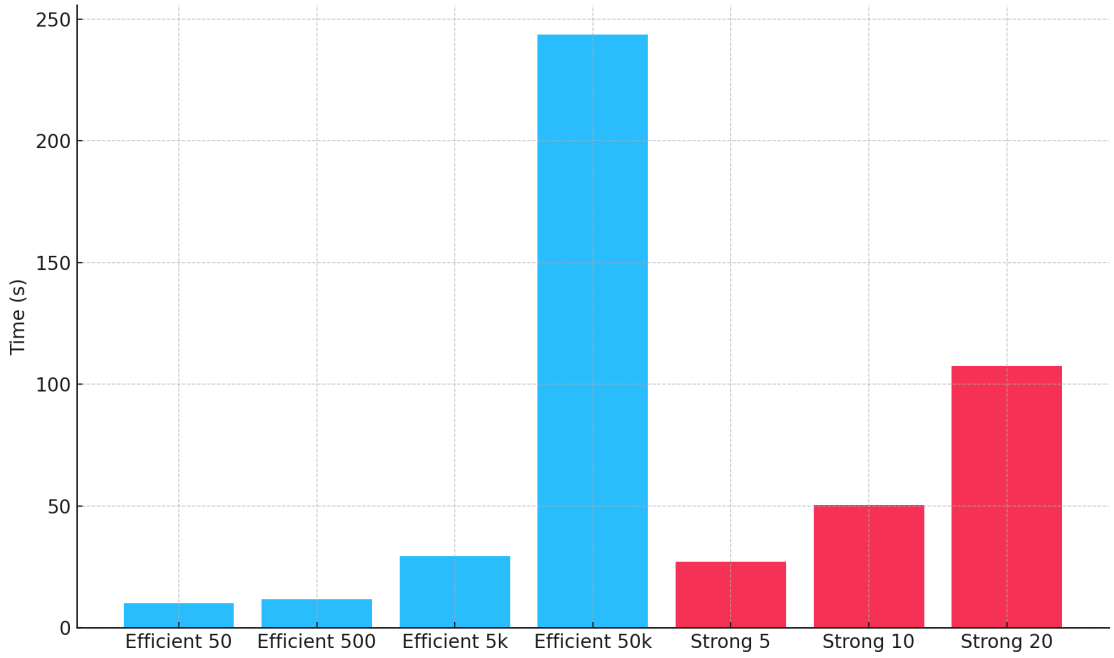

Figure 10.10: Compilation Time per Circuit

Memory Usage Similarly, as the number of constraints increases, the memory usage does so as well. We can observe that the Efficient circuit's memory usage progresses in a linear fashion, while the Strong circuit does. Less memory usage is better overall, and the graph 10.11 allows us to observe the differences between memory usage across different group sizes. Note that the Strong circuit's memory usage is substantially higher than the Efficient circuit's, with shorter group sizes (Strong 10 vs. Efficient 5k).

Output Files The output files increase in size as larger group sizes are used, across circuits, but their values aren't as big as the memory usage however. Their growth follows a linear relationship with the group size. Linearity is most evident in the Strong circuit, and not as much in the Efficient circuit, because the Efficient circuit has a higher constant term, which results in a seemingly exponential rise.

10.3.1.2 Discussion

Despite having larger group sizes, the Efficient circuit outperforms the Strong circuit in every metric. This shows that the Efficient circuit scales better with a growing group size than the Strong circuit.

Even though it is not present in table 10.1, earlier Strong circuit evaluations using the same group sizes as the Efficient circuit presented limitations due to system resource exhaustion. For context:

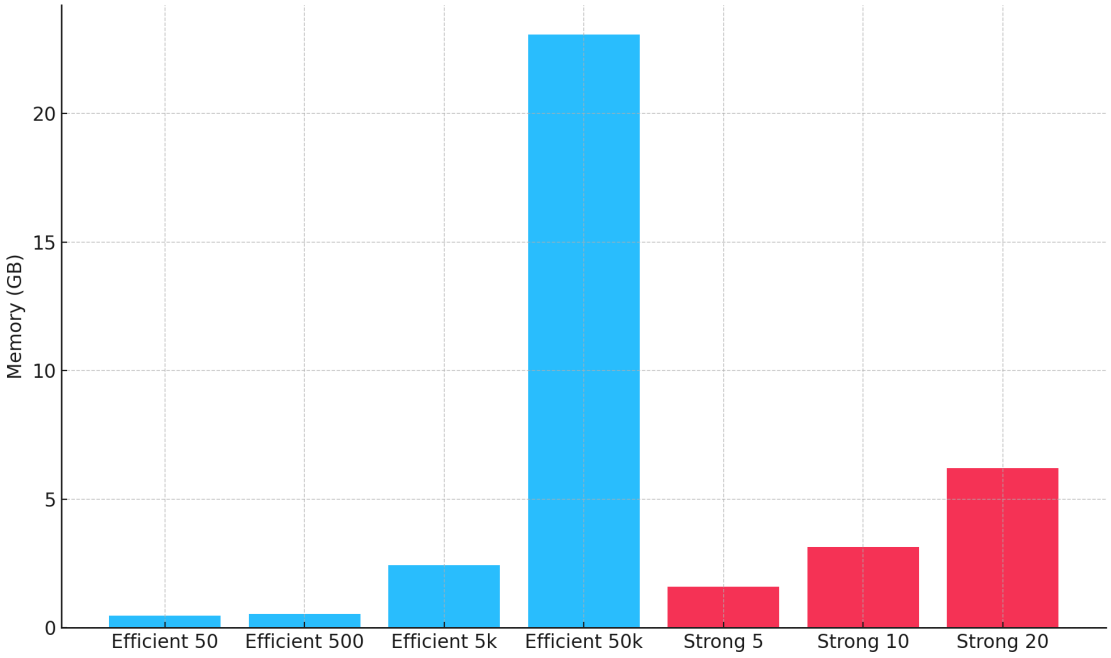

Figure 10.11: Compilation Memory Usage per Circuit

- Strong 50 compilation outputs 26835020 constraints which is 100 fold the Efficient 50's constraints, using 20GB of RAM.
- Strong 100 compilation outputs 53648920 constraints and uses 45GB of memory (including Swap).
- Strong 500 exhausted the system's resources by using over 280GB of memory.

Because these compiled files wouldn't follow the flow of the evaluation methods, as will be seen further on, we decided to shorten the group sizes as they allow for a proper comparison in terms of feasibility nonetheless.

In order to properly compile circuits as large as these, developers often rely on Cloud Provider machines, equipped with high core CPU's, large quantities of RAM (in the interval of 256GB-976GB) and large Swap partitions (400GB-1TB).

10.3.2 Trusted Setup Phase 2

The Phase 2 of the Trusted Setup was also conducted using a single machine, as the burden of the operation is placed on the verifier. The results from the evaluation are present in table 10.2, which includes execution time, memory usage and the output size. The table also includes information regarding the *Ptau* files used in the execution, namely their number of points and size.

We include 2 graphs, which compare execution time 10.12 and memory usage 10.13 across different circuits.

CPU	RAM (GB)	Circuits	Group Size	Ptau	Ptau Size (GB)	Time (s)	Memory (GB)	Output Size (GB)
Ryzen 7 7800X3D	32GB	Efficient	50	2^{20}	1.21	177.40	4.86	0.68
			500	2^{20}	1.21	184.55	5.20	0.76
			5k	2^{21}	2.42	332.73	7.65	1.78
			50k	2^{24}	19.33	2864.25	29.26	12.02
		Strong	5	2^{22}	4.83	402.00	20.02	3.08
			10	2^{23}	9.66	1107.60	34.31	6.13
			20	2^{24}	19.33	3354.13	73.13	12.25

Table 10.2: Trusted Setup Phase 2 - Execution time, memory usage and output size

10.3.2.1 Observations

Ptau The Ptau column present in table 10.2 displays the number of points the Ptau file can consider. To advance in the execution of Phase 2, the setup requires a number of points higher than the number of the circuit's constraints. Usually the smallest file with a greater number of points is used, but in some instances a bigger file might be required (if the protocol demands so).

We observe that as the group sizes increase, the number of points increase as well. Recall the the number of constraints also increases as the group sizes increase. Moreover, the Strong circuit requires a higher number of points even though its group sizes are smaller than the Efficient circuit's.

The size of the Ptau file also increases as the number of points increase as they are directly related.

Time The more time it takes to finish the phase, the worse. We observe that the time required to execute the Phase 2 grows as the group sizes grow as well, in their respective circuit. However the increase in time is not linear in either circuit. Moreover, it is evident that the time to complete the phase 2 is greater in all Strong circuits than it is on the Efficient circuits, with the exception of Efficient 50k, which is only surpassed by Strong 20. Graph 10.12 helps us to visualize this comparison. The worst case scenario in our evaluations is 2864s which corresponds to around 47 minutes.

Memory Usage Much like the time, memory usage also increases as the group sizes increase, somewhat linearly. The less memory usage, the better, and the Strong circuit uses substantially more memory than the Efficient circuit, even resorting to swap when the group sizes are 10 and 20, with 34GB and 73GB of memory usage respectively. Only the Efficient 50k circuit can contest, as it almost achieves maximum memory. Graph 10.13 illustrates the differences in memory usage across circuits.

Output Size The output contains the proving and verifying keys, along with intermediate keys that are generated amidst the execution. We can observe that executions that include

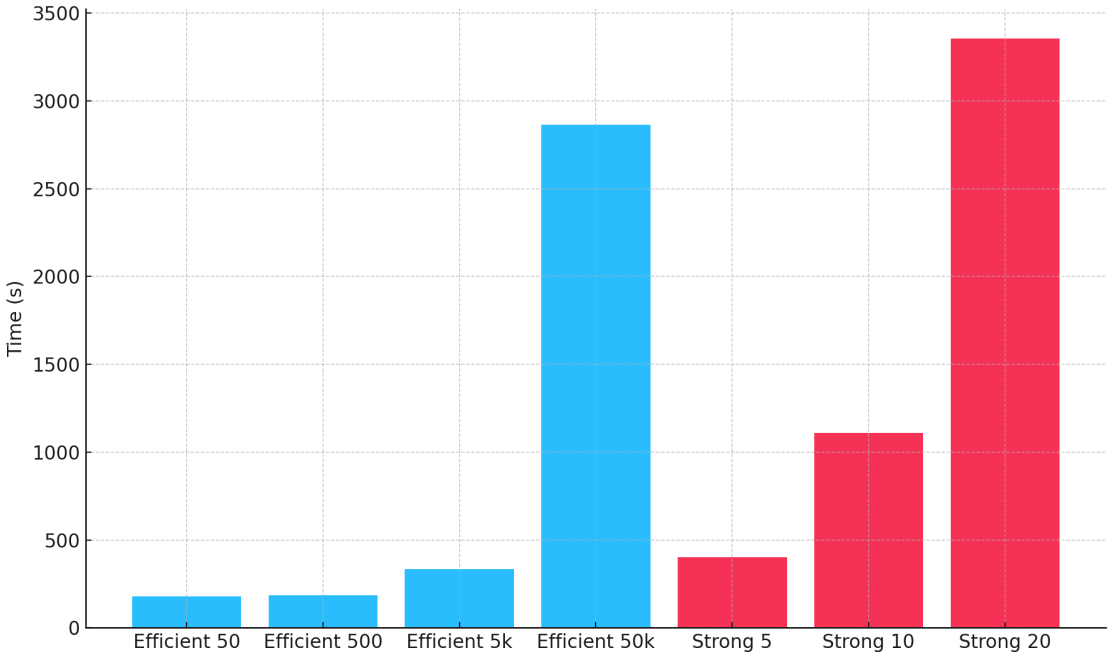

Figure 10.12: Execution Time of Trusted Setup Phase 2 per Circuit

circuits with higher constraints, and bigger ptau files, constitute bigger keys. We also observe that executions where the same number of points is used constitute similar output sizes.

10.3.2.2 Discussion

A higher number of points is associated with the higher number of constraints. As these constraints grow in size, the memory usage increases and therefore it is expected that the time taken to execute the Phase 2 increases as well. This becomes even more evident when the memory usage surpasses the system's maximum RAM, resorting to swap. The addition of swap operations increases the time significantly due to the disk being used more often during the execution.

The Strong circuit resorts to the swap partition substantially earlier than the Efficient circuit, due to its higher number of constraints. If a system is not configured to permit the usage of swap, or does not have enough swap available, it will fail the computation. Therefore, memory consumption is a highly significant metric to be aware of when conducting this operation.

Like the circuit compilation, it is advisable that this operation is conducted on more powerful machines with larger quantities of RAM and swap space when using circuits with a high amount of constraints.

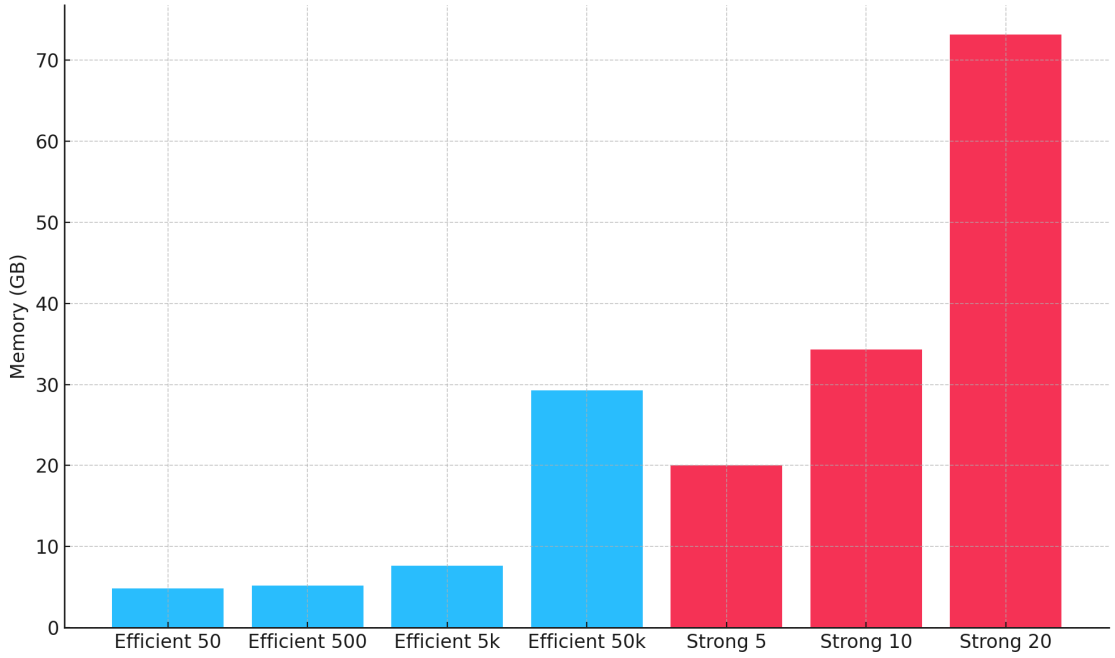

Figure 10.13: Memory Usage of Trusted Setup Phase 2 per Circuit

10.3.3 Witness Generation

The witness generation was conducted using two machines to better represent the use cases of the solution. Since the burden of this operation is placed on the prover, it is crucial to determine its feasibility in common home computers. Thus we conducted the operation using the aforementioned Ryzen 7 7800X3D, with its 32GB of RAM (representing high-end home computers) and an Apple Sillicon M1, with 16GB (representing a less powerful, more standard home computer). The results of the evaluation are present in table 10.3, which includes the time taken to finish the operation execution, the memory usage, and the size of the input and output files.

We include 2 graphs that compare the performance of all circuits with varying group sizes across the different machines, focusing on execution time in graph 10.14 and memory usage in graph 10.15.

10.3.3.1 Observations

Time The higher the time, the worse, and as evidenced in the table 10.3, the execution time is higher on the Apple M1 across all circuits, than it is on the Ryzen 7. In both systems the execution time increases as the group size increases in each circuit. The comparison of the execution times is made clear in graph 10.14, where it is evident that the difference in execution time across systems grows significantly as circuits with higher constraints are used.

CPU	RAM (GB)	Circuits	Group Size	Time (s)	Memory (GB)	Output Size (GB)
Ryzen 7 7800X3D	32GB	Efficient	50	4.85	0.09	0.02
			500	5.03	0.13	0.02
			5k	6.20	0.31	0.05
			50k	18.32	2.18	0.33
		Strong	5	25.14	0.26	0.09
			10	50.88	0.47	0.17
			20	105.31	0.90	0.34
Apple Silicon M1	16GB	Efficient	50	8.16	0.10	0.02
			500	8.71	0.12	0.02
			5k	9.92	0.32	0.05
			50k	26.12	2.31	0.33
		Strong	5	42.56	0.28	0.09
			10	84.68	0.50	0.17
			20	171.06	0.94	0.34

Table 10.3: Witness Generation - Execution time, memory usage and output size

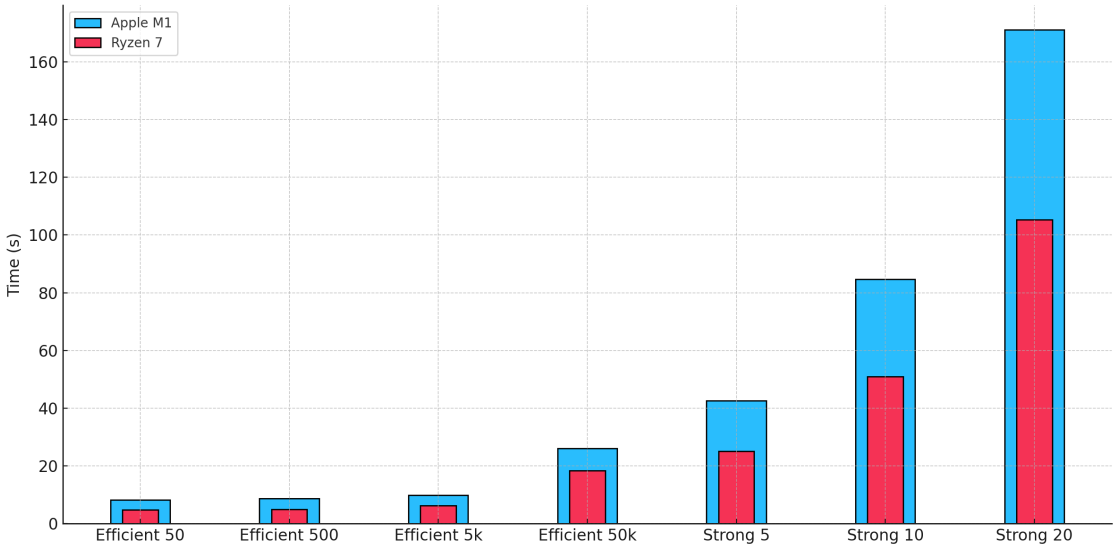

Figure 10.14: Execution Time of Witness Generation across Systems per Circuit

Memory Usage Lower memory usage is better, however it continues to increase as more complex circuits are used. The Apple M1 competes with Ryzen 7, having similar memory usages. The memory usage remains below 2.4GB across both systems and circuits. Only the Efficient 50k circuit stands out with its higher memory usage, which is made clear in graph 10.15.

Output Size The output size remains relatively small across all systems and circuits, but nonetheless, it increases as more complex circuits are used. However, the size of the outputs is the same regardless of the system used.

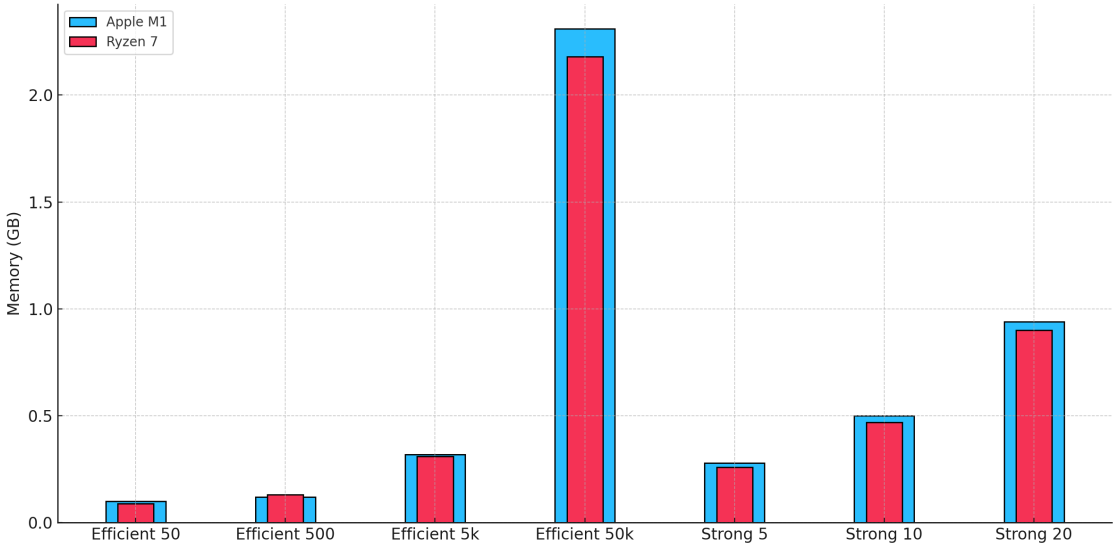

Figure 10.15: Memory Usage of Witness Generation across Systems per Circuit

10.3.3.2 Discussion

Considering that we use two different systems, which differ in performance substantially, we considered that differences in performance would be expected. Even though Ryzen 7 7800X3D outperforms the Apple Sillicon M1 in execution time, the Apple M1 succeeds in producing outputs in under 3 minutes, in comparison to Ryzen's 2 minute ceiling.

Additionally, since variation in memory usage is minimal and doesn't surpass any of the system's RAM limit, we can discard the use of the swap partition. Note that besides the difference in size, the Apple Silicon M1's RAM is slower than the Ryzen's, which contributes to the added execution time.

The output size does not differ regardless of system used, and remains considerably small when compared to the previous operations' output files.

Overall we find that this operation is very time efficient and consumes an amount of resources that is expected to be found in a common home computer.

10.3.4 Proof Construction

The Proof Construction was conducted on two systems as well, Apple Silicon M1 with 16GB RAM and a Ryzen 7 7800X3D with 32GB RAM, as its burden is also placed on the prover. Therefore, by testing across many systems we can determine the feasibility of this operation in common home computers. The results of the evaluation are included in table 10.4 which includes the execution time, the memory usage and the output size.

We also include 2 graphs for easier comparison of execution time across both systems and circuits, namely graph 10.16 and likewise the memory usage in graph 10.17.

CPU	RAM (GB)	Circuits	Group Size	Time (s)	Memory (GB)	Input Size (MB)	Output Size (MB)
Ryzen 7 7800X3D	32GB	Efficient	50	10.65	5.22	0.10	0.05
			500	11.24	5.40	1.00	0.47
			5k	25.12	11.10	10.40	5.10
			50k	346.02	42.10	104.10	49.00
		Strong	5	45.37	22.40	0.01	0.01
			10	97.64	28.15	0.02	0.01
			20	250.14	50.23	0.04	0.02
Apple Silicon M1	16GB	Efficient	50	19.75	4.25	0.10	0.05
			500	20.78	4.64	1.00	0.47
			5k	44.92	6.59	10.40	5.10
			50k	385.26	30.02	104.10	49.00
		Strong	5	85.63	14.20	0.01	0.01
			10	174.52	24.52	0.02	0.01
			20	376.35	39.50	0.04	0.02

Table 10.4: Proof Construction - Execution time, memory usage and output size

10.3.4.1 Observations

Time The shorter the execution time, the better, and as evidenced in table 10.4 simpler circuits have lower execution times, across both systems. The execution time grows significantly when constructing proofs with Efficient 50k and Strong 20 across systems, and when constructing a proof with Strong 10 using the Apple M1.

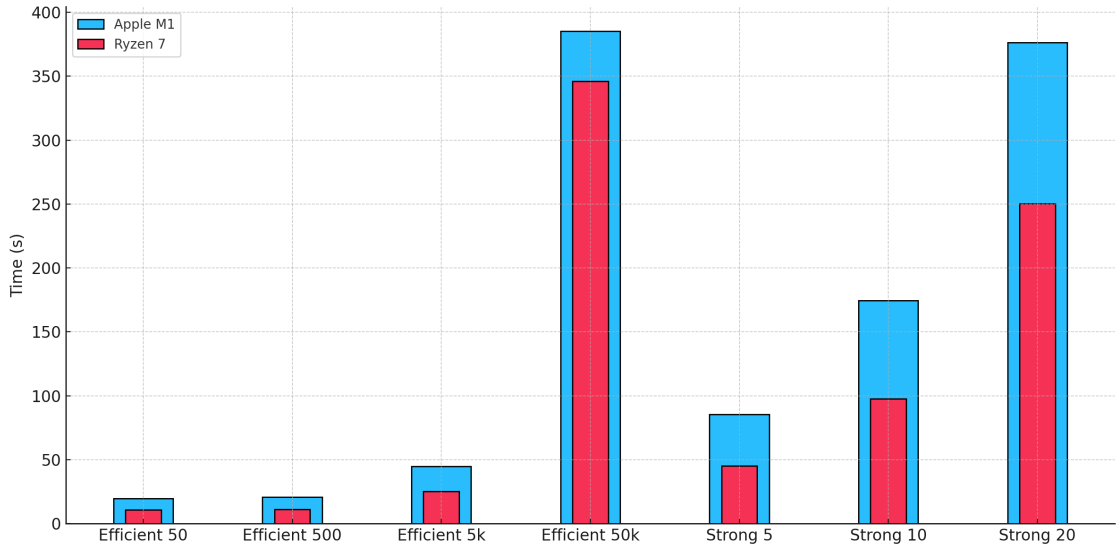

Figure 10.16: Execution Time of Proof Construction across Systems per Circuit

By visualizing the graph 10.16, the difference in execution time between systems becomes more apparent, increasing in higher complexity circuits such as Efficient 50k and the Strong circuits. Still, the Apple M1's execution times are approximate to those of the Ryzen using the Efficient circuits.

Memory Usage Higher memory usage is worse, and increases as the complexity of the circuit increases as well. However, unlike previous operations, the memory usage of the Apple M1 is less than that of the Ryzen's, for the same circuit.

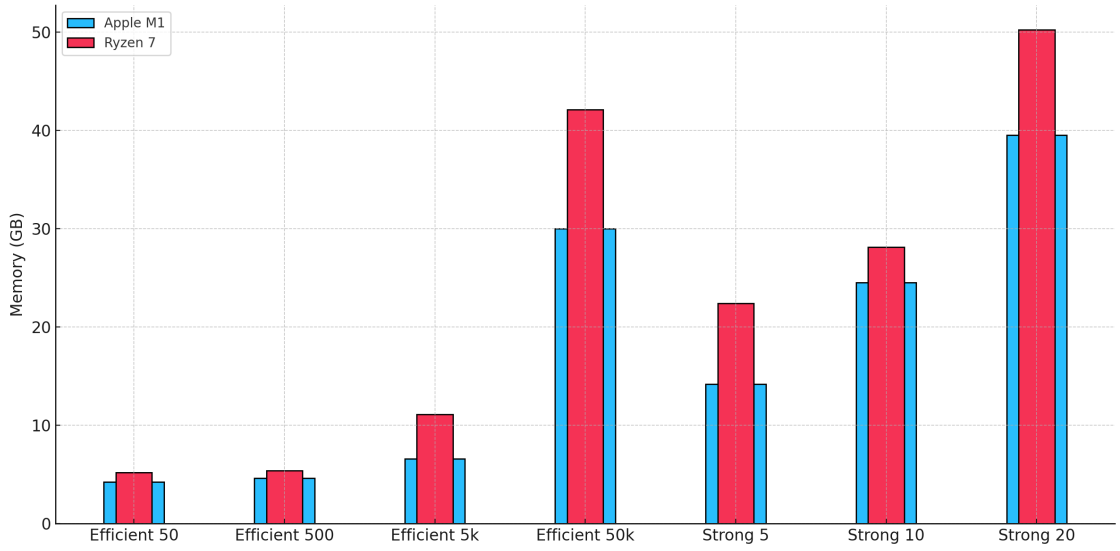

Figure 10.17: Memory Usage of Proof Construction across Systems per Circuit

The amount of memory used increases the most from the Efficient 500 circuit onwards, for both systems. In some instances, the memory usage surpasses the system's maximum amount of RAM, relying on swap space.

Input Size The Input Size refers to the size of the input signals provided by the prover. It grows linearly with the group size regardless of circuit or system, since it contains the group itself.

Output Size The Output Size refers to the size of the proof file, and the public input file. Like the input file, it grows linearly with the group size regardless of circuit or system, as it contains the group itself as well.

10.3.4.2 Discussion

We expected differences in execution time to differ substantially, but that was not the case for the Efficient circuits. The execution time of the Apple Silicon M1 is on par with the Ryzen 7 7800X3D, being slightly slower. Their performance differences become more apparent in the Strong circuits. Overall the Apple M1 succeeds in producing outputs in under 6:25 minutes, in comparison to Ryzen's 5:45 minute ceiling.

However, what we did not expect was the memory consumption of running this operation, and even more surprising was the lower memory usage from the 16GB RAM

Apple M1. The high memory usage can discourage many individuals from constructing proofs with large group sizes, especially when considering the Strong circuit's memory usage, and the Efficient circuits with group sizes above 500. Swap space was definitely used when evaluating Efficient 50k and Strong 20 (and Strong 10 in the case of the Apple M1), which could be the reason for the execution time substantial increase, regardless of the system used.

We can only assume why the Apple M1 consumed less memory. It may be due to differences in system architecture, as it is an ARM-based SoC (System on a Chip) with integrated RAM, as opposed to the Ryzen, which is a x86-based CPU with discrete (external) RAM. Besides, the Apple M1 operates on its host OS, while the Ryzen operates on a guest OS. Regardless of the reasons, the memory usage is still substantial considering the use case.

10.3.5 Proof Verification

The proof verification was conducted using two machines, even though it is an operation done by the verifier, to determine if the verification is indeed fast across different systems. We used two systems, an Apple Silicon M1 with 16GB of RAM and a Ryzen 7 7800X3D with 32GB of RAM. The results of the evaluation are included in table 10.5 which includes execution time and memory usage. These metrics can also be viewed in graphs 10.18 and 10.19, respectively, for easier comparison across systems and circuits.

CPU	RAM (GB)	Circuits	Group Size	Time (s)	Memory (GB)
Ryzen 7 7800X3D	32GB	Efficient	50	0.35	**0.05***
			500	0.52	0.46
			5k	2.13	1.98
			50k	-	-
		Strong	5	0.82	0.22
			10	0.87	0.22
			20	0.89	0.22
Apple Silicon M1	16GB	Efficient	50	0.30	0.17
			500	0.50	0.35
			5k	2.36	1.32
			50k	-	-
		Strong	5	0.32	0.15
			10	0.32	0.15
			20	0.32	0.15

Table 10.5: Proof Verification - Execution time and memory usage

10.3.5.1 Observations

Time Higher execution times are worse, and as observed in table 10.5 the execution time grows as the group size grows as well, regardless of circuit or system.

The Apple M1 has a lower execution time than the Ryzen 7 overall, only being slower in Efficient 5k, which happens to be the most time-consuming circuit, as evidenced in graph 10.18

No results are presented for Efficient 50k regardless of system used.

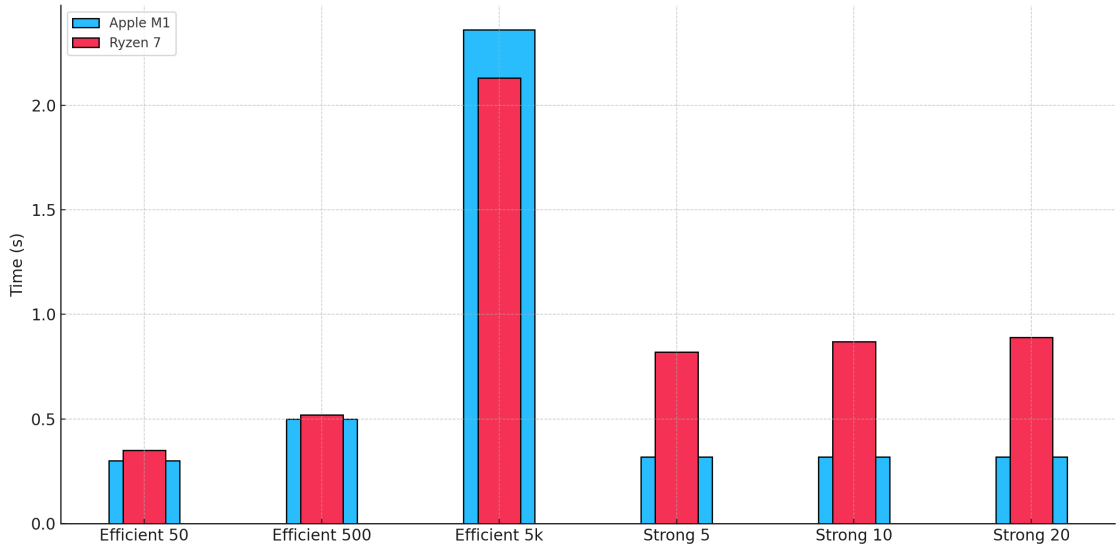

Figure 10.18: Execution Time of Proof Verification across Systems per Circuit

Memory Usage Memory usage grows as the group size grows as well, remaining under 2GB across all circuits and systems.

Overall, the Apple M1 uses less memory, except for the **Efficient 50** execution, where it uses three times as much as the Ryzen.

Efficient 5k stands out as the most memory-consuming circuit, regardless of system. It also has the biggest difference in memory usage between systems, as shown in graph 10.19.

10.3.5.2 Discussion

Overall proof verification is a fast operation as expected, performing under in 3 seconds for all circuits and platforms.

The memory usage is also low, only standing out with the Efficient 5k circuit. This is due to the verification key used to verify the proof, as it grows in size as the complexity and group size do too as well.

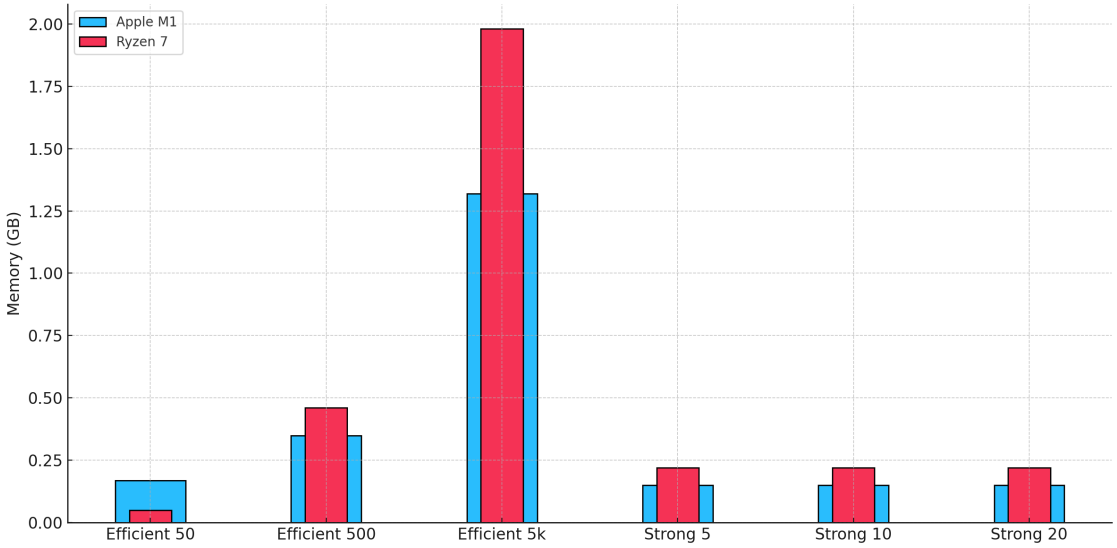

Figure 10.19: Memory Usage of Proof Verification across Systems per Circuit

These results show that the verification is fast regardless of circuit or system used, which can be beneficial for fast accounting of proofs.

However, you may have noticed that there are no evaluations for Efficient 50k. The reason there are no results is because, due to the high number of public inputs (group size), the verification key generated with this specific circuit has a size of 571MB. Unfortunately, *snarkjs*' proof verification implementation can only read verification keys with a size up to 512MB.

This presents a limitation regarding the number of public inputs we can provide. Even though using a group signature involving 50 thousand public keys decreases the probability of guessing the signer to 0.002%, it does not satisfy our ambitions of providing group signature proofs of entire country populations, such as Portugal's 11 million individuals.

10.4 Results Analysis

RAM is critical in each of these processes, as they are very memory-intensive. If the system memory is under the required, the processes take much longer than expected or fail altogether due to errors such as bad memory allocation errors (which can be overcome with the addition or increase of swap partitions).

We had to limit the size of the group to 5 - 20 in the Strong circuit due to the high constraint number. The compilation of such a circuit caused our system to consume over 280GB of memory with just 500 individuals, having crashed multiple times due to RAM and storage limitations.

Developers commonly use cloud-provided machine instances with RAM ranging from 256GB to 976GB, and 400GB Swap partitions for large circuits with over 20M constraints. Therefore, a verifier that effectively wants to compile such a circuit, and create its proving and verification keys using the Trusted Setup Phase 2, must consider its requirements.

The Trusted Setup Phase 2 with Strong 20, a circuit with 11M constraints consumed over 70GB of RAM. This goes to show how badly the Strong circuit scales with the group size, as opposed to the Efficient 50k circuit, with 7M constraints, which consumed just 30GB with a 2500 times bigger group.

Moreover, the size of the files required to generate a proof is of such magnitude that users might be discouraged from providing proofs. The proving keys required to generate the proofs can be as high as 6GB in our most intensive circuits, and the Witness Generation can use as much as 50GB of memory, possibly exceeding the disk space through the use of swap, resulting in crashes.

Another fact to consider, is that our tests were conducted in systems with more than 16GB RAM. They do not account for systems where the RAM varies from 4-8GB, which are more common.

Overall we find that the Efficient Circuit is a much better circuit due to its lower number of constraints with higher group sizes, making it feasible for providing proofs using modern home computer resources.

However, it is not free of criticism, as its public inputs contribute to its own detriment. With a 50 thousand group size, the circuit generates a verification key whose size exceeds the capability of the verification algorithm, blocking further action and making it impossible, as of yet, to use group sizes that represent country populations.

11

Conclusion

11.1 Conclusions

The work developed during the dissertation resulted in a mechanism to properly verify the authenticity of RSA signatures while maintaining anonymity within a known group of individuals, through the means of Zero-Knowledge Proofs.

With Signed Identity Documents in mind, our solution can contribute to several sensitive protocols where individuals' legitimacy is of most importance, while preserving their anonymity, which promotes participation overall.

To achieve this goal, a set of tools was developed. To properly obtain digital signatures, we created a card reader program capable of reading identity card information and generating valid RSA signatures following the PKCS#1 v1.5 signature scheme, compatible with current versions of the Cartão de Cidadão.

Afterwards, resorting to standard ZKP protocols using trusted setups, we developed circuits capable of evaluating the validity of a group signature, without disclosing any information about the signer. The only information revealed is the group itself, for posterior verification.

Additionally, we provided mechanisms to aid the evaluation of the solution, namely scripts and Python notebooks.
Our Zero-Knowledge is non-interactive, as initially desired, and satisfies the completeness, soundness, and zero-knowledge properties defined in section 2.3.

- **Completeness**: As long as the verifier concludes that the group of public keys has not been tampered with, and is equal to a publicly available group, or follows pre-determined guidelines, the property is satisfied;
- **Soundness**: This property is intrinsically satisfied by design, as it is ensured by the *Groth16* protocol and the Powers of Tau ceremony;
- **Zero-Knowledge**: Like the soundness property, this property is also ensured by the Groth16 protocol and Powers of Tau ceremony.

Our solution is not free of criticism, however, and presents limitations regarding the implementation itself and the evaluation results.

One such limitation is the fact that the circuit itself considers only 2048-bit RSA keys, which was a choice made in anticipation of the Cartão de Cidadão test cards, which would be used during evaluation, but was never the case. Actual Cartão de Cidadão cards possess a 3072-bit RSA key used for signatures.

The results obtained from the evaluations make evident that, even though the proof construction is within reasonable execution times, the memory usage may discourage individuals from participating when group sizes are large, especially if they have a system whose RAM is under 16GB.

Moreover, the fact that the group signature proofs are limited by a 50000 group-size ceiling, makes it impossible to scale up further using the current protocols, which makes it not suitable for country-wide collaborations where group sizes reach millions.

Additionally, it is expected that the verifier can use machines with high amounts of RAM to successfully compile circuits and proceed with their key generations. This should not be a problem for enterprises or governments, which (ideally) possess powerful machines and data centers. However, for single individuals gathering signatures for a petition, it may be expensive, resource-wise and money-wise if they rent an instance to do the computations for them.

Despite not being fulfilling our ambition of being suitable for country-wide solutions as is, our circuits are suitable for smaller group sizes similar to the number of people in organizations or companies, making it possible to have internal democratic processes, or making statements regarding validity of the documents through certified signatures.

11.2 Future Work

Several aspects were identified during the development and evaluation of this solution that present opportunities for improvement. Throughout the process, potential modifications were considered to enhance the overall efficiency of the circuits, focusing on optimizing both the verifier and signer components. These adjustments aim to reduce computational overhead, streamline communication, and improve resource management. Implementing these changes would lead to better performance between the involved parties, making the solution more scalable and suitable for a wider range of individuals.

- **3072-bit RSA** - To properly cover Cartão de Cidadão signatures, improving the circuit to work with 3072-bit RSA keys would make it fully compatible with the card's signature scheme.
- **Memory Constrained Phase 2** - Certain implementations [1] of the Trusted Setup's Phase 2 have been optimized to lower memory usage, although at the cost of increased

[1] kobigurk/phase2-bn254

execution time. This trade-off offers verifiers an alternative for compiling circuits or generating keys on less powerful machines, enabling greater flexibility in resource-constrained environments.

- **Public Input Reduction**
  - **Group Partitioning** - Rather than relying on a single group, proofs can be constructed using partitions within the same group, which decreases the complexity of the circuit. This approach would minimize overall resource consumption and, consequently, reduce execution time. However, it is crucial that these partitions are randomized to prevent linking or identifying specific K signers within a given K-partition, ensuring anonymity and security.
  - **Group Hashing** - A common method to ensure the integrity of sets is through the use of a set hash, which enables faster verification compared to element-by-element comparison. To efficiently verify an element's membership within a set using a hash, *Merkle Trees* are employed. Integrating Merkle Trees into our solution would significantly reduce the circuit's complexity from $O(n)$ to $O(\log n)$, as the effective group sizes shrink. For instance, a group of 50,000 elements is reduced to a set of 16, while a group the size of a country's population — such as Portugal's 11 million — would require only 24 iterations for verification.
- **Full Integration** - Fully integrating all the components developed in this solution to form a complete and functional application would streamline the execution process. This integration would enable seamless interaction between the components, ensuring efficient communication and validation, and reducing the complexity of the interaction between user and machine.

The aforementioned developments did not account for emerging cryptographic threats such as post-quantum cryptography, thus there is a need to revaluate the developed components and underlying technologies, to identify potential vulnerabilities introduced by quantum adversaries, which can make our security assumptions obsolete. and make efforts to adapt the system to quantum-resistant cryptographic standards while mitigating potential performance trade-offs and ensuring long-term security.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[2] W. Du and M. J. Atallah. "Secure multi-party computation problems and their applications". In: *Proceedings of the 2001 workshop on New security paradigms - NSPW '01* (2001). DOI: 10.1145/508171.508174. (Visited on 2020-04-30) (cit. on p. 1).

[3] P. McDaniel and T.-M. Shih. *Sustainability is a Security Problem*. Zoom, 2022-11. URL: https://acm-org.zoom.us/rec/play/Qq2cWNOs7adzTqdHwlGac7WNK1Q5HRcd87_RuG_nPPxdSGM9ZqX2hmAVAjD6Q-I_UKeCz2RrTa7HRu_o.yzXRQo0DJkJE646v?startTime=1667925686000&_x_zm_rtaid=JdFqTUFwRe2aQtobT3IC_w.1675347503231.e3aebedc42d134a4fe7f8cd3e51c209d&_x_zm_rhtaid=200 (visited on 2023-07-04) (cit. on pp. 2, 18).

[4] W. Agahari, H. Ofe, and M. de Reuver. "It is not (only) about privacy: How multiparty computation redefines control, trust, and risk in data sharing". In: *Electronic Markets* 32 (2022), pp. 1577–1602. DOI: 10.1007/s1252502200572w. URL: https://doi.org/10.1007/s1252502200572w (cit. on p. 2).

[5] A. Yao. *Protocols for Secure Computations*. 1982. URL: https://crysp.uwaterloo.ca/courses/pet/W11/cache/www.cs.wisc.edu/areas/sec/yao1982-ocr.pdf (visited on 2023-06-07) (cit. on p. 5).

[6] O. Goldreich. "Secure Multi-Party Computation". In: *Manuscript. Preliminary Version* (1999-03) (cit. on p. 5).

[7] M. Naor, ed. *A general composition theorem for secure reactive systems*. Springer Berlin Heidelberg, 2004, pp. 336–354 (cit. on p. 5).

[8] O. Goldreich, S. Micali, and A. Wigderson. "How to Play ANY Mental Game". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: Association for Computing Machinery, 1987, pp. 218–229. ISBN: 0897912217. DOI: 10.1145/28395.28420. URL: https://doi.org/10.1145/28395.28420 (cit. on p. 6).

[9] Y. Ishai et al. "Zero-Knowledge Proofs from Secure Multiparty Computation". In: *SIAM Journal on Computing* 39 (2009-01), pp. 1121–1152. DOI: 10.1137/080725398 (cit. on p. 6).

[10] R. Cramer, I. B. Damgård, and J. B. Nielsen (aut). *Secure Multiparty Computation*. Cambridge University Press, 2015-07, pp. 16–17. URL: https://books.google.pt/books?hl=pt-PT&lr=&id=HpsZCgAAQBAJ&oi=fnd&pg=PR9&dq=Secure+Multi+Party+computation&ots=aPKfwiJ5uP&sig=4osn3H6iv0OXsrU9ri4bBxEvkIE&redir_esc=y#v=onepage&q=Secure%20Multi%20Party%20computation&f=false (visited on 2023-06-06) (cit. on p. 6).

[11] D. Rotaru. *awesome-mpc*. GitHub, 2022-10. URL: https://github.com/rdragos/awesome-mpc (cit. on p. 7).

[12] G. Lax, F. Buccafurri, and G. Caminiti. "Digital Document Signing: Vulnerabilities and Solutions". In: *Information Security Journal: A Global Perspective* 24 (2015-02), pp. 1–14. DOI: 10.1080/19393555.2014.998843 (cit. on p. 7).

[13] E. Tonkin and J. Allinson. *Signed metadata: Method and application*. International Conference on Dublin Core and Metadata Applications, 2006. URL: https://dcpapers.dublincore.org/pubs/article/view/861 (cit. on p. 7).

[14] C. Veloso and M. Almeida. *A Segurança da Informação no Cartão do Cidadão Português*. 2009-12 (cit. on p. 7).

[15] G. Português. *Specimen of Cartão de Cidadão*. URL: https://www.autenticacao.gov.pt/o-cartao-de-cidadao (visited on 2023-07-05) (cit. on p. 8).

[16] *Política de Certificado de Autenticação*. 2019-01. URL: https://pki.cartaodecidadao.pt/publico/politicas/PJ.CC_24.1.2_0011_pt_AuC.pdf (visited on 2023-07-02) (cit. on p. 8).

[17] IRN. *Política de Certificados da EC de Assinatura Digital Qualificada do Cartão de Cidadão*. 2022-01. URL: https://pki.cartaodecidadao.pt/publico/politicas/POL23_PC.ECAsC_1.0_Jan.2022_sig_INCM.pdf (visited on 2023-07-02) (cit. on pp. 8, 33).

[18] S. Boeyen et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. IETF, 2008-05. URL: https://datatracker.ietf.org/doc/html/rfc5280 (visited on 2023-07-02) (cit. on p. 8).

[19] W. Vercruysse. *What is a qualified signature seal creation device QSCD - eSignature Knowledge Base -*. ec.europa.eu, 2019-12. URL: https://ec.europa.eu/digital-building-blocks/wikis/display/ESIGKB/What+is+a+qualified+signature+seal+creation+device+QSCD (visited on 2023-07-02) (cit. on p. 8).

[20] *The knowledge complexity of interactive proof-systems*. Association for Computing Machinery, 1985, pp. 291–304. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178 (cit. on p. 8).

[21] O. Goldreich, S. Micali, and A. Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems". In: *Journal of the ACM* 38 (1991-07), pp. 690–728. DOI: 10.1145/116825.116852. URL: https://people.csail.mit.edu/silvio/Selected%5C%20Scientific%5C%20Papers/Zero%5C%20Knowledge/Proofs_That_Yield_Nothing_But_Their_Validity_or_All_Languages_in_NP_Have_Zero-Knowledge_Proof_Systems.pdf (visited on 2023-06-25) (cit. on pp. 8, 9).

[22] O. Goldreich and H. Krawczyk. "On the Composition of Zero-Knowledge Proof Systems". In: *SIAM Journal on Computing* 25 (1996-02), pp. 169–192. DOI: 10.1137/s0097539791220688. (Visited on 2021-03-04) (cit. on p. 8).

[23] *NP-complete problem | mathematics*. Encyclopedia Britannica, 2023-06. URL: https://www.britannica.com/science/NP-complete-problem (visited on 2023-06-28) (cit. on p. 8).

[24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. Fifth Printing (August 2001). CRC Press, 1996. URL: https://cacr.uwaterloo.ca/hac/ (visited on 2023-06-28) (cit. on p. 9).

[25] M. Blum, P. Feldman, and S. Micali. "Non-Interactive Zero-Knowledge and Its Applications". In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 103–112. ISBN: 0897912640. DOI: 10.1145/62212.62222. URL: https://doi.org/10.1145/62212.62222 (cit. on p. 9).

[26] A. De Santis, S. Micali, and G. Persiano. "Non-Interactive Zero-Knowledge Proof Systems". In: *Advances in Cryptology — CRYPTO '87*. Ed. by C. Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 52–72. ISBN: 978-3-540-48184-3 (cit. on p. 9).

[27] N. Bitansky et al. "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 326–349. ISBN: 9781450311151. DOI: 10.1145/2090236.2090263. URL: https://doi.org/10.1145/2090236.2090263 (cit. on p. 9).

[28] E. Ben-Sasson et al. *Scalable, transparent, and post-quantum secure computational integrity*. 2018. URL: https://starkware.co/wp-content/uploads/2022/05/STARK-paper.pdf (cit. on p. 9).

[29] M. Petkus. "Why and How zk-SNARK Works". In: (2019-06). DOI: 10.48550/arxiv.1906.07221. (Visited on 2023-07-03) (cit. on p. 9).

[30] Chainlink. *zk-SNARK vs zkSTARK - Explained Simple | Chainlink*. Chainlink Blog, 2023-02. URL: https://blog.chain.link/zk-snarks-vs-zk-starks/ (visited on 2023-07-03) (cit. on p. 9).

[31] P. Team. *zk-SNARKs vs zk-STARKs — Comparing Zero-knowledge Proofs*. Panther Protocol Blog, 2022-09. URL: https://blog.pantherprotocol.io/zk-snarks-vs-zk-starks-differences-in-zero-knowledge-technologies/#what-are-zk-snarks (visited on 2023-07-03) (cit. on p. 9).

[32] U. B. C. RDI. *Lecture 10.3: What is a zk-SNARK?* Youtube, 2021-10. URL: https://www.youtube.com/watch?v=gcKCW7CNu_M&t=1006s (visited on 2023-06-09) (cit. on p. 10).

[33] ConsenSys. *Introduction to zk-SNARKS*. ConsenSys, 2017-03. URL: https://consensys.net/blog/developers/introduction-to-zk-snarks/ (visited on 2023-07-08) (cit. on p. 10).

[34] V. Tan. *Awesome Zero Knowledge*. GitHub, 2023-07. URL: https://github.com/ventali/awesome-zk (visited on 2023-07-03) (cit. on p. 10).

[35] J. Groth. "On the Size of Pairing-based Non-interactive Arguments". PhD thesis. University College London, UK, 2016. URL: https://eprint.iacr.org/2016/260.pdf (visited on 2024-09) (cit. on pp. 10, 11).

[36] P. Barreto and M. Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order". PhD thesis. Escola Politécnica, Universidade de São Paulo; Lehrstuhl für Theoretische Informationstechnik, Rheinisch-Westfälische Technische Hochschule Aachen, 2006. URL: https://eprint.iacr.org/2005/133.pdf (visited on 2024-09) (cit. on p. 10).

[37] P. Barreto, B. Lynn, and M. Scott. "Constructing Elliptic Curves with Prescribed Embedding Degrees". PhD thesis. Laboratório de Arquitetura e Redes de Computadores (LARC), Escola Politécnica, Universidade de São Paulo, Brazil; Computer Science Department, Stanford University, USA; School of Computer Applications, Dublin City University, 2002. URL: https://eprint.iacr.org/2002/088.pdf (visited on 2024-09) (cit. on p. 10).

[38] S. Bowe et al. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. 2019. URL: https://eprint.iacr.org/2017/1050.pdf (cit. on p. 11).

[39] V. Nikolaenko et al. "Powers-of-Tau to the People: Decentralizing Setup Ceremonies". In: *Applied Cryptography and Network Security*. Ed. by C. Pöpper and L. Batina. Cham: Springer Nature Switzerland, 2024, pp. 105–134. ISBN: 978-3-031-54776-8 (cit. on p. 11).

[40] privacy-scaling-explorations. *GitHub - privacy-scaling-explorations/perpetualpowersoftau*. GitHub, 2024. URL: https://github.com/privacy-scaling-explorations/perpetualpowersoftau (visited on 2024-07) (cit. on p. 12).

[41] Iden3. *Circom*. GitHub, 2022-05. URL: https://github.com/iden3/circom (visited on 2024-05) (cit. on p. 12).

[42] Iden3. *snarkjs*. GitHub, 2022-05. URL: https://github.com/iden3/snarkjs (visited on 2024-05) (cit. on p. 13).

[43] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Commun. ACM* 21.2 (1978-02), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342 (cit. on p. 13).

[44] K. Moriarty et al. *RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2*. IETF Datatracker, 2016-11. URL: https://datatracker.ietf.org/doc/html/rfc8017 (visited on 2024-05) (cit. on pp. 14, 15).

[45] D. Chaum and E. Van Heyst. "Group signatures". In: *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10*. Springer. 1991, pp. 257–265 (cit. on p. 15).

[46] R. L. Rivest, A. Shamir, and Y. Tauman. "How to Leak a Secret". In: *Advances in Cryptology — ASIACRYPT 2001*. Ed. by C. Boyd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 552–565. ISBN: 978-3-540-45682-7 (cit. on p. 16).

[47] U. Fiege, A. Fiat, and A. Shamir. "Zero Knowledge Proofs of Identity". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: Association for Computing Machinery, 1987, pp. 210–217. ISBN: 0897912217. DOI: 10.1145/28395.28419. URL: https://doi.org/10.1145/28395.28419 (cit. on p. 17).

[48] T. Beth. "Efficient zero-knowledge identification scheme for smart cards". In: *Advances in Cryptology—EUROCRYPT'88: Workshop on the Theory and Application of Cryptographic Techniques Davos, Switzerland, May 25–27, 1988 Proceedings 7*. Springer. 1988, pp. 77–84 (cit. on p. 17).

[49] M. Burmester, Y. Desmedt, and T. Beth. "Efficient Zero-Knowledge Identification Schemes for Smart Cards". In: *The Computer Journal* 35.1 (1992-02), pp. 21–29. ISSN: 0010-4620. DOI: 10.1093/comjnl/35.1.21. eprint: https://academic.oup.com/comjnl/article-pdf/35/1/21/1116430/35-1-21.pdf. URL: https://doi.org/10.1093/comjnl/35.1.21 (cit. on p. 18).

[50] M. Rosenberg et al. *zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure*. Cryptology ePrint Archive, Paper 2022/878. https://eprint.iacr.org/2022/878. 2022. URL: https://eprint.iacr.org/2022/878 (cit. on p. 18).

[51] W3C. "Decentralized Identifiers (DIDs) v1.0 becomes a W3C Recommendation". In: *W3C* (2022-07). URL: https://www.w3.org/press-releases/2022/did-rec/ (visited on 2023-07-05) (cit. on p. 18).

[52] J. Camenisch and G. M. Zaverucha. "Private Intersection of Certified Sets". In: *Financial Cryptography and Data Security*. Ed. by R. Dingledine and P. Golle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 108–127. ISBN: 978-3-642-03549-4 (cit. on p. 19).

[53] S. Garg et al. "Credibility in Private Set Membership". In: Springer-Verlag, 2023. DOI: 10.1007/978-3-031-31371-4_6 (cit. on p. 19).

[54] F. Zhang et al. *DECO: Liberating Web Data Using Decentralized Oracles for TLS*. 2020. URL: https://arxiv.org/pdf/1909.00938.pdf (cit. on p. 19).

[55] A. Faz-Hernández, W. Ladd, and D. Maram. "ZKAttest: Ring and Group Signatures for existing ECDSA keys". PhD thesis. Cloudflare, Inc; Cornell Tech, 2021. URL: https://eprint.iacr.org/2021/1183.pdf (visited on 2024-09) (cit. on p. 20).

[56] A. Davidson et al. "STAR". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022-11. DOI: 10.1145/3548606.3560631. URL: https://doi.org/10.1145%2F3548606.3560631 (cit. on p. 22).

[57] D. Boneh et al. *Lightweight Techniques for Private Heavy Hitters*. 2023. arXiv: 2012.14884 [cs.CR] (cit. on p. 22).

[58] T. Morais. *Cartao-cidadao-data-signer*. GitHub, 2024-08. URL: https://github.com/tmorais-fctunl/Cartao-cidadao-data-signer (visited on 2024-09) (cit. on p. 32).

[59] amagovpt. *GitHub - amagovpt/autenticacao.gov: Middleware Oficial de Identificação Eletrónica em Portugal - Cartão de Cidadão, da Chave Móvel Digital e Sistema de Certificação de atributos profissionais*. GitHub, 2019-09. URL: https://github.com/amagovpt/autenticacao.gov (visited on 2023-12-20) (cit. on p. 32).

[60] jacksoom. *circom-rsa-verify: Zero Knowledge Proof for RSA*. GitHub, 2023-04. URL: https://github.com/zkp-application/circom-rsa-verify (visited on 2024-05) (cit. on p. 36).

[61] iden3. *Background in ZK - Circom 2 Documentation*. Circom.io. URL: https://docs.circom.io/background/background/#zero-knowledge-proofs (visited on 2024-04) (cit. on p. 36).

[62] T. Morais. *Circom-group-rsa-verify*. GitHub, 2024-08. URL: https://github.com/tmorais-fctunl/Circom-group-rsa-verify (visited on 2024-09) (cit. on pp. 44, 56).

[63] M. Albrecht et al. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity*. Cryptology ePrint Archive, Paper 2016/492. 2016. URL: https://eprint.iacr.org/2016/492 (cit. on p. 47).

[64] T. Morais. *RSA-Keys-Generator-and-Signature*. GitHub, 2024-08. URL: https://github.com/tmorais-fctunl/RSA-Keys-Generator-and-Signature (visited on 2024-09) (cit. on p. 55).