Zero Knowledge Proofs with MPC-in-the-Head

António Maria Ribeiro Ferreira Santos da Cunha
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2023/2024

Zero Knowledge Proofs with MPC-in-the-Head

António Santos da Cunha

Dissertation carried out as part of the Master in Information Security

Department of Computer Science

2023/2024

Supervisor
Bernardo Portela, Assistant Teacher, DCC-FCUP
Co-supervisor
Hugo Pacheco, Assistant Teacher, DCC-FCUP

Dedicated to my parents who've brought me this far, my success and all future accomplishments are theirs as well

Abstract

Imagine a world where you can ensure to a bank you have enough money to withdraw a certain amount without ever sharing your balance with the bank.

That is the promise of Zero Knowledge Proofs (ZKPs), a cryptographic technique that allows for a *prover* to successfully convince a *verifier* of the truthness of a given statement without ever disclosing any information about said statement other than it's truthness. Now, although very useful in contexts where privacy, security and integrity are paramount, these proofs are traditionally very costly and inefficient, making them unreliable in real world applications where scalability and performance are of the utmost importance.

There are many approaches that aim to solve this issue, but in the work described in this document we're going to dive in the concept of introducing simulating Multi-Party Computation techniques in the proving mechanisms of traditional ZKPs. This approach, called MPC-in-the-Head, is a rather promising technique that has shown great impact in improving traditional ZKPs, turning them into scalable and highly efficient proofs that can be used in 'real-world' scenarios.

As such we'll be instantiating a specific ZK protocol that uses MPCitH, ZKBoo as well as a circuit generator with the goal of addressing a real use case, of proving citizenship on a given country without revealing any information about the national ID. This use case reflects the problem of proving private set membership (PSM) in a ZK fashion, and it is our goal to test the suitability of the instantiated protocol to address this kind of real-life scenarios.

Keywords: Zero-Knowledge (ZK), Zero-Knowledge Proofs (ZKP's), Multi-Party Computation (MPC), Multi-Party Computation in-the-Head (MPCitH), Private Set Membership (PSM)

Table of Contents

List of Figures

1. Introduction

1.1. Motivation

Zero-Knowledge Proofs (ZKPs) have emerged as an exciting cryptographic technique with the promise of allowing one party, the prover, to prove to another party, the verifier, the truthness of a given statement without revealing any additional information about the statement itself during the process. This unintuitive property is crucial in scenarios where privacy and confidentiality are of the utmost importance.

The main advantage of ZKPs is their ability to enable secure computations without compromising the privacy of the involved parties. By leveraging on ZKPs, it becomes possible to perform rather complex calculations on sensitive data while guaranteeing the underlying information remains confidential. This has a vast number of implications in various domains, such as:

- Financial Transactions: by allowing anonymous payments where the buyer would convince the seller he could afford the payment without revealing any information about he's balance or personal account.
- Electronic Voting: by allowing voters to cast their ballots without revealing their preferences or identity to anyone.
- Medical Data Analysis: One of the main issues of medical studies consist on the anonymization of the patient, with ZK techniques, no identifiable information about the patient would ever be disclosed, allowing for the analysis of the medical data without ever knowing the underlying patient information.

However, traditionally these proofs are computationally costly and quite inefficient, making ZK often regarded only out of theoretical intrest and unfit to be adapted and introduced in real-world problems.

While there are some efficient approaches designed to address this issue, these are often conceived to deal with very specific contexts instead of being of general-purpose.

That's where MPCitH comes in. Multi-party computation (MPC) is another cryptographic technique that enables multiple parties to jointly compute a function over their private

inputs without revealing any individual inputs. MPC-in-the-Head is then the result of combining ZKPs with a local simulation of a regular MPC protocol. By doing this we unlock the possibility to create even more efficient and scalable, general-purpose protocols.

By leveraging ZKPs with MPCitH, we unlock several benefits:

- Efficiency: MPCitH can reduce the computational complexity and communication overhead associated with traditional MPC protocols, making the solution more practical for real-world applications.
- Flexibility: MPCitH allows for the ZK protocol to be able to efficiently address any general-purpose computation as long as it can be described and executed in a MPC protocol.
- Optimised Verification: MPCitH often generates efficient proofs, making these verifiable by other parties in a rather efficient time, helping the protocol become more scalable.
- Simple Design: MPCitH stands out for its ease of representation and understanding, making it a simple and easy way

Finally, in this document we aim to take this improvement and apply it onto a real-world use case of proving citizenship for a given country. This use case is described by the private set membership (PSM) problem in the ZK context, a fundamental cryptographic task that involves determining whether a given element belongs to a private set without revealing the contents of the set or the element itself. This problem has significant applications in various domains, including IAMs[1], proof of ownership, cryptocurrency, *etc.*...

Keeping in mind that given the fact that the protocol here instantiated is a general-purpose one, our approach can easily be adapted to deal with any other problem or context by changing only the circuit being generated.

In conclusion, the combination of ZKPs with MPC in the head offers a promising approach to solving the PSM problem. By leveraging the power of these cryptographic techniques, we can develop efficient, secure, and privacy-preserving solutions that have the potential to transform various domains and applications.

[1] Identity Access Managers

1.2. Objectives

The work described in this document aims to investigate the usage of ZKPs with the optimisation introduced by MPCitH to address the limitations of traditional proofs, specifically when creating and solving multiple instances of the PSM problem.

As such, the goals established for this work were to:

- Explore ZKP constructions and the impact of introducing MPCitH to check how efficient these are in a real use case.
- Emulate an example instance of ZKBoo with a prover and verifier that'll emulate the participants in our use case.

1.3. Report Outline

This work we'll be exposed and organised in the following chapters:

1. Preliminaries: This will be the chapter that will provide the reader with the necessary background knowledge to better understand the concepts further mentioned.
2. State of the Art: This is the chapter where we'll go over some key approaches from the last 10 years highlighting the improvements at each step as well as key implementation aspects.
3. ZKBoo and Circuit Generation: This is the chapter where we'll closely look into the cornerstones necessary to achieve a viable solution to the proposed use case.
4. Experimental Results: This is the point where we have a critical analysis regarding performance and possible improvements given the obtained experimental results.
5. Conclusion and Future Work: Finally we'll conclude by summarizing the key findings of the developed work and highlight possible future studies with this as a starting point.

2. Preliminaries

2.1. Zero-Knowledge

Zero-Knowledge proofs have emerged as a rather powerful and promising cryptographic tool with a vast number of applications. First proposed in 1985 on the paper ”The Knowledge Complexity of Interactive Proof-Systems” by Goldreich, Micali, and Wigderson [1].

Their vision was to allow for a prover to be able to convince a verifier of the thruthness of a public statement without revealing any information about the statement itself. As such the innovation brought by these aproaches lies in their ability to separate the verification part of the protocol from the disclosure of its underlying details.

The best way to start grasping how these kind of proofs work is one of the examples proposed by Quisquater *et al.* on their short paper ”How to Explain Zero Knowledge Protocols to Your Children”[2] as an attempt to easily illustrate how ZKPs work. In this paper, the authors describe many practical and illustrative examples to introduce the concept of a ZK protocol in a high-level and easy to understand format. The example in question is known as ”The Strange Cave of Ali Baba”(as illustrated in 2.1) and it takes place in a cave that has 2 paths that meet in the middle, separated by a locked door. In this scenario, Alice wants to prove to Bob the truthness of the Statement ”I have the key to the door”.

However, since we wants to do this in a Zero-Knowledge Fashion, she establishes the following protocol:

1. Alice randomly chooses a path (either A or B).
2. Bob, not knowing which path Alice chose, randomly chooses a path for Bob to come out from.
3. Alice finally returns using the path chosen by Bob.

Notice that by following this protocol, Bob ends up with the exact same amount of information he started the experiment with, however he can say with an $1 - \epsilon$[1] amount of certainty that he is properly convinced that Alice owns, in fact, the key.

[1] probability of fooling Bob

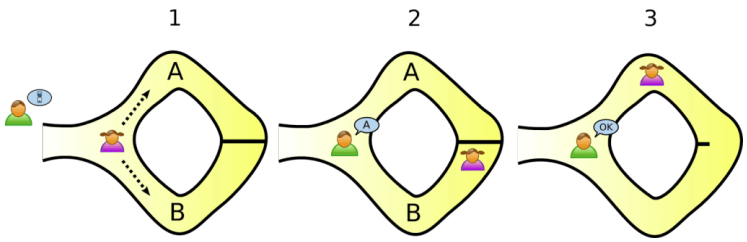

Figure 2.1: The Strange Cave of Ali Baba Experiment[3]

This is due to the fact that if we repeat the experiment just once there is 1 possible way of Bob going in and out the same path, thus not proving he actually owns the key. The more this experiment is repeated the less likely Bob can trick Alice.

Being ϵ the probability of Bob convincing Alice he owns the key without actually owning it, we have:

$$\epsilon = \frac{1}{2^n} \tag{2.1}$$

with *n* being the number of rounds we repeat this same experiment.

It is important to highlight that most of the traditional proofs are safe when assuming an attacker with bound computational resources, meaning someone with enough computational power could convince the verifier of a wrong statement.

2.2. Multi-Party Computation

Diving now into the 2nd building block of the approach here being explored we have Multi-Party Computation, another cryptographic primitive that allows for multiple parties to jointly compute one or more functions over their private set of inputs without revealing any actual individual input.

The primary advantage of this technique is then to enable executing a function $f(x)$ in a private fashion, meaning we can run it like $f(x_1, x_2, x_3)$ where each x_i comes from a different participant, so that all of the 3 participants end up with the output of f but none knows any other input besides their own.

2.2.1 Additive Secret Sharing

Now, although there are many ways to accomplish MPC, the one we'll be focusing on is by using a property called additive secret sharing. Additive secret sharing is a cryptographic technique used to divide a secret (SHARE) x into n secrets (for n participants to compute on), or shares, so that only specific subsets of these shares can reconstruct (UNSHARE) the original secret.

Take into consideration the secret value $x = 15$, if we break it into $x_1 = 5, x_2 = 7, x_3 = 3$ we manage to break it in such way that no party knows the original value and we can easily reassemble it and have $x_1 + x_2 + x_3 = x(15)$. With these kinds of shares we manage to perform regular computations (like adding, multiplying, dividing and subtracting) while preserving the original secret.[4]

These kind of mechanisms are possible due to the fact that we're working in a fixed field Z_r (r, the range of the field, being usually a large prime for arithmetic circuits and 2 for boolean circuits).

Algorithm 1 SHARE(x)

Require: $r \geq 0$
$x_1 \leftarrow rand(r)$
$x_2 \leftarrow rand(r)$
$x_3 \leftarrow y - (x_1 + x_2) mod(r)$

Return $\rightarrow (x_1, x_2, x_3)$

Algorithm 2 UNSHARE(x_1, x_2, x_3)

Return $\rightarrow (x_1 + x_2 + x_3) mod(r)$

This way we ensure we get the same values as when calculating with the original secrets but having the guarantee that the original secrets, each party's input and intermediate values stay private.

While addition operations may be quite simple (as for the associative trait of addition), things can escalate in complexity when we're dealing with operations like multiplication (as for their distributive trait). The example pictured in 2.2 describes a simple multiplication between 2 inputs x and y and unrolls as such:

1. We first break both private values (x, y) each into 3 shares for each of the 3 participants (P_1, P_2, P_3) which makes $x \times y$ is the same as $(x_1 + x_2 + x_3) \times (y_1 + y_2 + y_3)$.

2. Then, due to the distributive character of traditional multiplication, we'll end up with parts like $x_1 y_2$, which no party can compute by themself as no one owns shares of other parties. As such, each party sends their shares to the next one.

3. Finally, after the product is computed by each party, the resulting shares of the operation are fed into the UNSHARE function to reassemble it into the actual output.

In this scenario it is noticeable that even though parties are communicating between them they are never able to reconstruct the original secrets as everyone will always still be missing one final pair of shares.

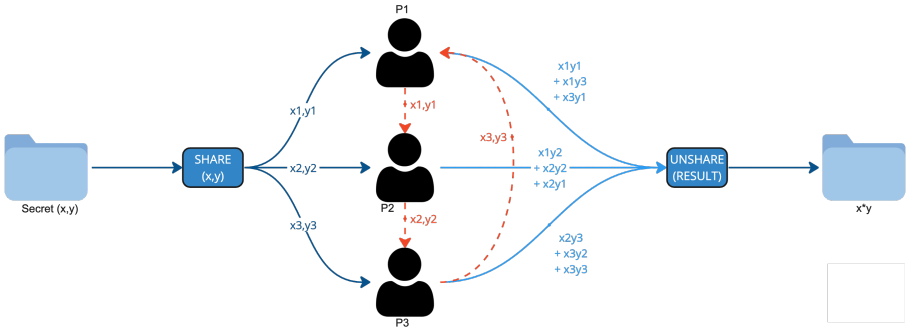

Figure 2.2: Example of a MPC multiplication

To be able to safely perform these, when drafting a MPC protocol we need to ensure 2 key properties:

1. Correctness: A protocol is correct if and only if the probability of the function computed through MPC outputing the same as if normally computed is 1:

$$\Pr[f(x) = \text{MPC}(f, x)] = 1 \tag{2.2}$$

2. d-Private: An MPC protocol is d-private, if there exists a possibility for someone to simulate a party's view of the execution without knowing the secret. Meaning the real execution never reveals any information about the secret and that it is safe to $d-1$ corrupt parties, meaning it can only be subverted if all d parties were to collude.

Then, the level of complexity and additional concerns can vary according to the chosen assumption on the parties' behaviour. We have 2 different assumptions on malicious behaviour:

- Active: we assume that a malicious actor or corrupt party can behave freely to try and corrupt the protocol by obtaining additional information.
- Semi-honest: we assume that parties will properly follow the protocol but they still might be curious and try to infer additional information by looking into the exchanged messages.

By assuming a semi-honest perspective, one concern we have is that of introducing randomnesses r_1, r_2, r_3 in the messages send between parties in such way that they mask

the value being sent but also $r_1 + r_2 + r_3 = 0$ thus ensuring they do not impact the computation of f.

2.3. MPC In-The-Head

Taking the previous section we now look into the ”in-the-head” concept, an approach that simply put is a technique that resorts to the assurances of traditional MPC protocols to be able to simulate this process locally and still guarantee that:

- Each of the generated views, by themselves, do not reveal any information about the original secret(s) (Zero Knowledge Property).
- All of these same views can not be consistently created without knowledge of the original secret(s) (Soundness Property).

This way, the goal of integrating this approach with traditional ZKPs will be to have the prover generate a set of views of a simulated MPC protocol and have the verifier challenge this by opening a subset of these. Like in the above mentioned ”The Strange Cave of Ali Baba”, there is still a slim chance of the prover to get lucky and being able to forge one of the chosen views, however, as it is not possible to forge them all (given the soundness property above mentioned), the more this experiment is repeated, the slimmer the prover’s odds to fool the verifier become.

2.4. Bristol Circuits

Bristol Circuits are the final building block to be able to assemble the implementation described ahead.

Circuits are used to illustrate MPC computations by offering a formal representation of the computations to be performed by the participating parties. Usually defined as a DAG (Directed Acyclic Graph) where nodes represent computational operations (Gates) and edges the flow of data between these operations (Wires). This representation when well structured offer us safety guarantees for both the inputs and the operations being performed, allowing for each and every function f to be presentable into an admissible version of MPC.

The way we chose to present these circuits is by using the Bristol Format, one of various acceptable ways of describing circuits in text formats, and taken as a standard, that traditionally follows the below mentioned structure:

- A Header, composed by:
  - A line defining the number of gates and then the number of wires in the circuit.
  - Then two numbers defining the number *n1* and *n2* of wires in the inputs to the function given by the circuit (usually Bristol Format will describe a function with arity 2).
  - Then on the same line comes the number of wires in the output *n3*.The wires are then ordered so that the first *n1* wires correspond to the first input value, the next *n2* wires correspond to the second input value. The last *n3* wires correspond to the output of the circuit.
- The body, composed by a list of gates defined by:
  - Number input wires.
  - Number output wires.
  - List of input wires (indexes).
  - List of output wires (indexes).
  - Gate operation.

So given the above mentioned structure, we can have a simple logic circuit turn into an easily parsed text file like this:

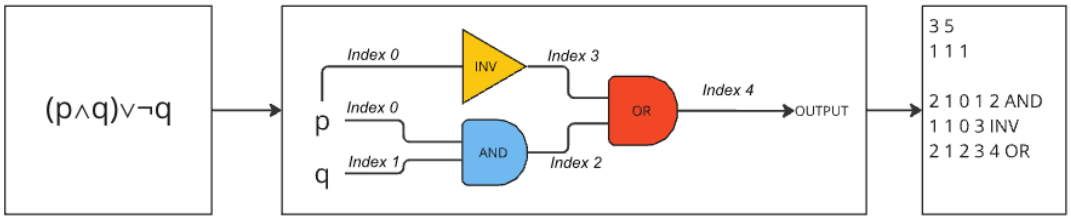

Figure 2.3: Example on how to convert a simple boolean circuit to the bristol format

In this specific report however, we'll be dealing with Bristol Fashion Circuits, a newer, more modern and tidier format proposed by professor Nigel Smart [5] that adapts the syntax to accommodate for more complex operations and generic problem solving. The new format differs from the old one by changing the header to have 2 new fields:

- The number of input values *niv* (e.g. if doing z=a+b mod p we have *niv*=3, one input each for a, b and p). Followed by *niv* sizes (in wires) respective to the number of inputs.
- The number of output values *nov* (e.g. if doing z=a+b mod p we have *nov*=1, corresponding to z). Followed by *nov* sizes (also in wires) respective to the number of outputs.

This format allows for the creation of functions with multiple outputs such as MANDs (multiple AND gate that takes *2n* input wires and produces *n* outputs result of the application of the traditional AND to each pair of input wires) making circuit definition more concise and execution way more efficient.

2.5. Private Set Membership

Set membership is the problem of checking whether an element belongs to some private set S without disclosing which element it is.

It manifests in a vast amount of contexts, mostly in applications with large datasets where efficiency and privacy (either of S or a given element $x \in S$) are at stake. Some examples are scenarios where someone needs to prove citizenship, like a Bank proving to regulators a new client is citizen of a given country. In this scenario the list of citizens might be public but the bank might not want to disclose information that identifies the client himself.

More recently this problem has also emerged in the blockchain context mainly in cryptocurrency design to track the Unspent Transaction Outputs (UTXO[2])[6].

In the context of the proposed use case, of proving citizenship of a given country, this problem reflects as a citizen proving he belongs to a given set S of a country's population without disclosing any information about himself. For this scenario, where we deal with large datasets that aim to represent an entire population worth of data, 2 concerns arise about both scalability but also privacy.

2.5.1 RSA Accumulators

RSA Accumulators represent a promising way to prove PSM in an efficient and computationally cheap fashion. The way this technique approaches this problem is by representing

[2]A set containing all the coins that haven't been already spent and are therefore eligible to be spent in a new transaction

the set publicly as an accumulator A of its elements. So that:

$$A \leftarrow G^{\prod_{i=1}^{n} e_i} mod N \tag{2.3}$$

Where G is a generator from $\mathbb{Z}_N^*$ that represents the ring of non-negative elements modulo N and e_i primes belonging to the ring.

From here, we know that a prover P knows an element $x \in S$ iff he knows a witness W so that $W_i^e = A$. However, this approach is not enough to ensure the correctness of the proof as a malicious party might simply use $W = \sqrt[n]{A}$, thus being able to easily compute A without necessarily knowing an element of the set.

That's where Pedersen Commitments[3] become handy. By having the prover ensure that he commits the element he knows into a public Pedersen commitment, then we ensure the locking and blinding of the proof, guaranteeing a proper trust-worthy execution. The proof will then consist in proving knowledge of (e_i, W, r) such that:

$$A = W^{e_i} mod N \wedge C_e = g^{e_i} h^r \in \mathbb{G} \tag{2.4}$$

Where $\mathbb{G}$ is the group of prime order where the element e_i is committed and g and h 2 generators of the said group.[6] In this way, we ensure an efficient way of proving PSM in ZK as described in the work of Camenisch and Lysyanskaya [7], who managed to design a ZKP protocol that follows this approach.

[3] Pedersen commitments are a type of cryptographic commitment scheme that allow a party to commit to a value without revealing it. They are based on the discrete logarithm problem and offer strong security properties.

3. State of the Art

In this chapter we'll deep dive over the evolution of related works over the past 10 years, analyse the contributions introduced by every new protocol and how that helped shape future work.

3.1. ZK-SNARKS

SNARKs (Succint Non-Interactive Argument of Knowledge) are proof systems designed to efficiently prove integrity of results for large computations.

By breaking down this system's name we get the following definitions that already paint us a picture on the way these proofs work [8]:

- Succint: the size of the proof is very small compared to the size of the statement or the witness (the size of the computation itself *e.g.*).
- Non-Interactive: a concept already covered that means it does not require multiple rounds of interaction between the prover and the verifier.
- ARgument: meaning it is assumed to be secure only for provers that have bounded computational resources, which means that provers with enough computational power can convince the verifier of a wrong statement.
- Knowledge (bound): meaning it is assumed to be impossible for the prover to construct a valid proof without knowing a certain witness for the statement.

One of the disadvantages of these proof systems is the need of a trusted setup phase to safely exchange the keys involved in the proof generation and validation.

By introducing the ZK concept in SNARKs, we can add a ZK property that enables the proof to be done without revealing anything about the intermediate steps. These new systems, called ZK-SNARKs, can preserve the original ones properties while eliminating the risk of privacy breaches during setup.

A traditional ZK-SNARK can so be described using 3 algorithms[8]:

- *Gen* is the setup algorithm, generating a necessary string *crs* used later in the proving process as a public binding parameter and some verification key *vrs*, sometimes assumed to be secret to the verifier only. It is typically run by a trusted party.

- *Prove* is the proving algorithm that takes as input the *crs*, the public statement *u* and a secret witness *w* and produces the proof π.
- Finally, *Verify* takes the verification key *vrs*, the statement *u* and the proof π as an input and returns *True* or *False*.

3.2. ZK-STARKs

This new approach (Zero-Knowledge Scalable Transparent Arguments of Knowledge) follows a transparent proof system that enables for efficient verification while maintaining privacy. These systems differ from standard ZK-SNARKs for the ability to be either interactive or non-interactive, and they improve on the traditional proof system in the following fields [9]:

1. Transparency:
   - ZK-SNARKs require a trusted setup which involves a phase to generate and share public parameters generated with non-public randomness. This lack of transparency constitutes a single point of failure as if this is compromised the security of the whole system can be as well.
   - ZK-STARKs on the other hand are completely transparent and do not require a trusted setup phase, making them more robust again vulnerabilities associated with these trusted step processes.[1]
2. Proof Size and Verification Time
   - ZK-SNARKs produce rather shorter proofs when compared with ZK-STARKs (around 1000 times smaller) allowing for quite quick verification times.
   - ZK-STARKs generate larger proofs, often several megabytes long, but these can be verified in a time-efficient manner, making them way more scalable.
3. Computational efficiency: Due to the cryptographic techniques use by ZK-SNARKs, their generation can usually be quite more computationally intensive when compared with ZK-STARKs.
4. Practical Applications:

[1] ZK-STARKs are designed to have verification times that are often smaller than the naive running time of computations since they manage to have the complexity of the verifier bein independent of the prover's complexity

- ZK-SNARKs are widely used in applications requiring short proofs such as privacy-preserving cryptocurrencies (Zcash e.g.).
- ZK-STARKs on the other hand represent a promising alternative for decentralised systems that require scalability and transparency.

3.3. ZKBoo

ZKBoo, first mentioned in ”ZKBoo: Faster Zero-Knowledge for Boolean Circuits” by Irene Giacomelli, Jesper Madsen and Claudio Orlandi [10] describes a proposal for practically efficient ZKPs especially optimised for Boolean circuits (although it can be adapted to arithmetic ones as well, as it is a general purpose protocol). It leverages the MPCitH approach, enhancing the practicality of zero-knowledge proofs.

This innovative approach constitutes a generalization of the IKOS (Ishai et al. acronym) protocol by extending it into enabling faster Σ-protocols[2] suitable for various soundness parameters. In addition to this, it also dables in function decomposition introducing a simplified method for achieving linear decompositions for arithmetic circuits, aiding in the construction of more efficient ZK protocols. This leads to the above-mentioned outstanding performance benchmarking with experimental results indicating that ZKBoo outperforms existing systems like Pinocchio and ZKGC in proving times, especially for practical circuits such as SHA-1 and SHA-256

This is a non-interactive protocol (meaning it can be executed with only 1 exchange, and does not need back and forth communication) that takes the approach to ZK proposed by Ishai *et al.* (IKOS construction [11]) as a strating point and uses MPCitH to increase its efficiency and performance in a prover/verifier protocol.

This protocol is designed to deal with statements of the structure ”I know x such that $y = \phi(x)$ holds, where ϕ is a public circuit and y a public value. The way this works is by following the below described structure for a prover P and a verifier V.

In the figure 3.1 we can observe that for the public parameters C, a public boolean circuit that represents a function f, and y a public output, usually an expected output we have a prover P who wants to show that she knows a x for which $f(x) = y$ with y being a public value. To achieve this, she'll follow the following approach:

[2] A Sigma (Σ) protocol for L is a 3-move public coin interactive proof system that allows a prover to convince a verifier that he knows a witness w of a public instance x without disclosing w.

1. P will run f in MPC, using the secret input x and breaking it into 3 shares (we're assuming MPC with 3 different parties).

2. By playing the role of these 3 parties, P will generate 3 different individual views of what each of this fictitious parties would "watch" in a real MPC scenario. (As previously stated, none of these views will reveal any information about x and it is impossible to generate them consistently without knowing x).

3. After generating these, P will commit these 3 views by hashing them[3] and send them to the verifier V, locking the generated views and giving V the guarantee that those won't be able to be changed without V noticing.

4. V then chooses an index $e \in 1,2,3$ so that P opens the commits e and $e+1$.

5. P then sends them the opened commits (the original views).

6. Finally V accepts the proof iff:

   (a) The received views are consistent with the commits (*Hash*(views) = commits).

   (b) He can reconstruct the MPC execution described in $view_{e+1}$ using the views he got.

   (c) UNSHARE(final shares) = y.

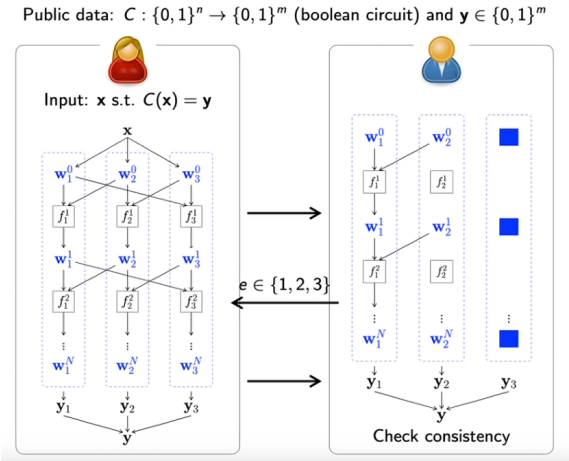

Figure 3.1: Illustration of the execution of ZKBoo[10]

Imagine now, that we want to prove that we have a private key p_k used for digital signatures. In this scenario, $f(p_k)$ would consist in creating a digital signature and verifying its

[3] Thus ensuring the blinding property that guarantees the commitments do not reveal any information from the views

validity using a public key integrated in the circuit. In this scenario, proving $f(p_k) = \textit{true}$ means a valid signature was produced.

While ZKBoo exhibits a linear scaling of proof size with circuit complexity, its verification times are comparable to SNARKs, and the proving process is significantly faster—up to 1000 times quicker than some traditional methods.

Despite all of the mentioned advantages, ZKBoo's proof sizes increase quite a lot with circuit complexity, which may be a trade-off in certain applications with large datasets.[10]

3.4. NIZPoK

Non-Interactive Zero-Knowledge Proofs of Knowledge (NIZPoKs) are a rather modern and innovative approach proposed by Jonathan Katz, Vladimir Kolesnikov and Xiao Wang (KKW) back in 2018 that leverages on symmetric-key primitives, enhancing both security and efficiency. It refines the previously introduced concept of MPCitH enabling the generation of way shorter proofs while maintaining computational efficiency.

This protocol applies a 5-round public-coin proof of knowledge, which can be compressed to three rounds [4] while retaining zero-knowledge properties. This facilitates the creation of NIZKPoKs for arbitrary circuits where the Prover simulates an MPC execution of these same circuits. This technique is designed to help manage circuit complexities while ensuring security against both honest and malicious verifiers.

This proposed protocol shows rather significant improvements when compared to protocols that came before it, such as ZKBoo, particularly when it comes to proof length and efficiency. As such, the advances proposed in the original article position NIZKPoKs as a noteworthy contribution in the still evolving world of ZKPs in particular for post-quantum applications.[12]

3.5. Ligero

Ligero represents a novel zero-knowledge argument protocol for NP problems, which innovatively reduces communication complexity to the square root of the verification circuit size.

This protocol stands out due to its concrete efficiency and reliance solely on black-box symmetric-key primitives, specifically collision-resistant hash functions. Unlike some of

[4] A public-coin proof of knowledge is a cryptographic protocol that allows a prover to demonstrate knowledge of a secret (or witness) to a verifier in an interactive manner. This type of proof is characterized by its use of public coins, where the verifier's challenges are made using random values that are publicly available.

the previously mentioned protocols, Ligero does not require any trusted setup or complex public-key operations, making it accessible for practical applications.

Some of Ligero's most significant contributions and innovations include:

- Sublinear Communication: Ligero achieves communication complexity that is proportional to the square root of the size of the verification circuit, which is a significant improvement over prior protocols that often had linear or worse complexities.
- Use of Symmetric-Key Primitives: Ligero exclusively employs symmetric-key techniques, which simplifies the implementation and reduces potential vulnerabilities associated with public-key cryptography as well as communication overhead.
- Concrete Efficiency: The authors of the original paper (Ames et al.) provide an implementation demonstrating the protocol's efficiency, particularly in scenarios requiring multiple evaluations of the same NP verification circuit, thereby improving amortized communication and verification times (see 3.2).
- Lightweight Design: The protocol is designed to be lightweight, making it suitable for applications with limited computational resources while still maintaining robust security features (like privacy preserving cryptocurrency or blockchain technologies).
- Optimised to deal with multi-instance complexity: In multi-instance settings, Ligero shows enhanced performance, allowing for efficient verification of several instances of NP problems simultaneously.

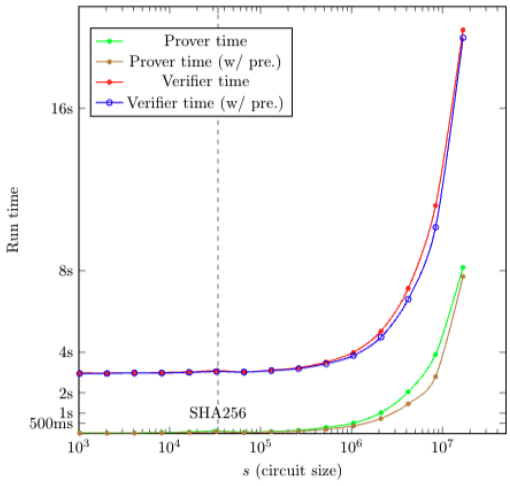

Figure 3.2: Prover and verifier running times for verifying a single instance of different circuit sizes

As such, Ligero is currently positioned as a really fierce competitor when faced against earlier works regarding prover running time, especially when dealing with larger verification circuits, representing itself as a significant advancement in the ZK arguments field, however, all of this efficiency comes with some associated costs such as[13]:

- Design complexity: Ligero's design is more intricate than most protocols, which can complicate implementation and understanding for developers and researchers.
- Communication complexity: Although Ligero achieves sublinear communication complexity, this efficiency diminishes for smaller circuit sizes. For circuits smaller than 3 million gates, the communication complexity may not outperform simpler approaches (making it hard to scale).
- Prover and verifier performance: While Ligero demonstrates competitive prover running times for large verification circuits, the prover's overhead is logarithmic in relation to the circuit size, which can constitute a disadvantage in scenarios that demand quick proof generations.
- Limited Flexibility: Ligero is optimised for specific applications such as verifying SHA preimages. In these contexts its performance is quite robust, but in more generalised protocols this performance usually gets limited.

4. ZKBoo and Circuit Generation

In this section we'll go over 2 key aspects of the final implementation, both ZKBoo's prover and verifier scheme and the Bristol Circuits generation and how it works in MPC (for both boolean and aritmetic implementations).

4.1. ZKBoo

As previously described, ZKBoo is a ZK protocol optimised for Boolean Circuits (even though it can be adapted to be general purpose) that utilises a commitment-hybrid model, this is a property derived from using MPC techniques, a commitment-hybrid model integrates the concept of commitments with the functionality of secure computation, enabling parties to collaboratively compute a function while maintaining the privacy of their inputs to enhance efficiency.

The way this protocol works is by having 3 different phases (when in an interactive form):

1. Commit Phase: The prover (P) samples random tapes [1] and runs, in an MPC fashion, the specific computation described by the public circuit. Then, P generates n views (that reflect each of the parties' view over their own execution of the circuit) and commits them by using a computationally strong cryptographic function (usually SHA-256) sending those, after, to the verifier (V)

2. Challenge Phase: V randomly selects an index i challenging P to reveal 2 of the commitments (i and $i+1$) corresponding to the views with the same indexes.

3. Verification Phase: Finally, V receives the views (or openings) from P and accepts the proof by running 3 different checks: Consistency Check, Execution Check and Expected Result Check (as mentioned in **??**).

This patter of communication classifies as a Σ- protocol, ensuring, by definition, the following key properties:

- Completeness: If both P and V are honest and $y \in L$ then Pr[$(P, V)(y) =$ accept] $= 1$.

[1] These tapes are a way to separate the randomness from the rest of the process making it deterministic. this will later be used to allow us to repeat processes that are, otherwise, probabilistic

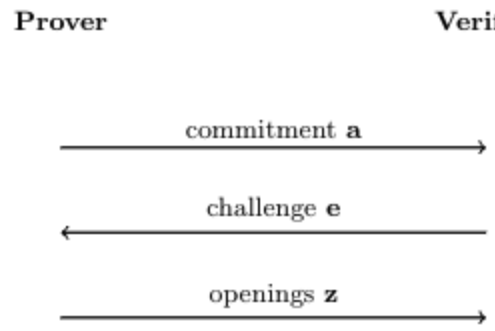

Figure 4.1: Communication pattern of a Σ-protocol

- s-special soundness: This property implies a bound of $\frac{(s-1)}{c}$ on the soundess error (with c being the cardinality of the choice set, in this case, 3) of the protocol.

For this communication to take place, the first step is exactly to sample the n tapes for the n fictitious parties simulating the MPC execution.

As such, the first concept to be studied and implemented on this work was exactly MPCitH, starting with Additive Secret Sharing.

4.2. Proof Creation (Bristol Circuits + MPCitH)

As previously stated, the circuit syntax here used was a modern twist on traditional Bristol Format called Bristol Fashion[5].

For our use case, we need to represent a relation $R(x)$ that reflects the knowledge of preimage of HASH(x). As such, for a set S of n elements, we want our circuit to verify if we know one of the $n-$many preimages of a set C of hashes, meaning it will hash the input and repeatedly test it against every hash of the commitment group returning *True* if it matches with any of the values being compared.

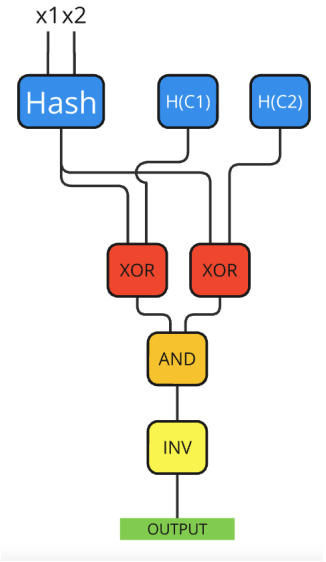

Figure 4.2: Small illustrative example of a circuit that follows the mentioned approach

The chosen approach was so to represent each party using a *Map<Integer,Integer>* so that we were able to dynamically had indexes that didn't necessarily exist yet. By doing this we would always ensure that as long as we ordered the secret sharing positions in the map by their key we'd be able to have every party index aligned and ensure for a smooth computation with their keys acting as indexes.

This leaves us the ability to quickly perform operations with the gate information we can read from the circuit like so:

```
1  public class Circuit{
2  (...)
3  private void AddOp (int w1_index, int w2_index, int w3_index){
4      p1.add(out_index, (p1.get(w1_index)+p1.get(w2_index))%module);
5      p2.add(out_index, (p2.get(w1_index)+p2.get(w2_index))%module);
6      p3.add(out_index, (p3.get(w1_index)+p3.get(w2_index))%module);
7      }
8  (...)
9  }
```

Whilst the above mentioned approach may look quite simple even when executing in MPC, operations with communication involved can be slightly trickier, as it requires randomness generation to mask the sent shares but also operation decomposition.

Take a look at the case of the AND operation, if we look into this operator's truth table

it becomes apparent that in the field $\mathbb{Z}_2$ this operation works the exact same way as a regular multiplication. As such, having $\textit{share}(x) = (x_1, x_2, x_3)$ and $\textit{share}(y) = (y_1, y_2, y_3)$ we have:

$$\begin{aligned} x * y &= (x_1 + x_2 + x_3) * (y_1 + y_2 + y_3) \\ &= x_1y_1 + x_1y_2 + x_1y_3 + x_2y_1 + x2_y2 + x_2y_3 + x_3y_1 + x_3y_2 + x_3y_3 \end{aligned} \tag{4.1}$$

Shares like x_1y_2 involve communication between parties, in this case, from p_1 to p_2, as such, we associate randomnesses per party (r_1, r_2, r_3) leaving us with the following:

```
1  public class Circuit{
2      (...)
3
4      private void andOp(int w1_index, int w2_index, int out_index) {
5          int rand12 = gen1.nextInt(module);
6          int rand23 = gen2.nextInt(module);
7          int rand31 = gen3.nextInt(module);
8          p1.put(out_index,align(((p1.get(w1_index)*p1.get(w2_index)
9                                  +p1.get(w1_index)*p3.get(w2_index)
10                                 +p3.get(w1_index)*p1.get(w2_index)
11                                 +rand31-rand12) % module)));
12         p2.put(out_index,align(((p2.get(w1_index)*p2.get(w2_index)
13                                 +p2.get(w1_index)*p1.get(w2_index)
14                                 +p1.get(w1_index)*p2.get(w2_index)
15                                 +rand12-rand23) % module)));
16         p3.put(out_index,align(((p3.get(w1_index)*p3.get(w2_index)
17                                 +p3.get(w1_index)*p2.get(w2_index)
18                                 +p2.get(w1_index)*p3.get(w2_index)
19                                 +rand23-rand31) % module)));
20     }
21     (...)
22 }
```

Notice that the added randomness does not impact the calculations as when added up (UNSHARE) everything adds up to 0, not influencing the result of the computations. Notice also that in both examples given, we add every operation result to each party's tape. This is necessary to introduce the above-mentioned concept of "Trace". Traces represent the perspective from a single party from its own circuit execution. It represents the corner

stone necessary to build the views and commitments (consequently). A Trace is defined as such:

```
1   public class Trace{
2       private int seed;
3       private Map<Integer, Integer> shares = new HashMap<>();
4
5       public int getSeed() {
6           return seed;
7       }
8
9       public Map<Integer, Integer> getShares() {
10          return shares;
11      }
12  }
```

By providing the used seed in the randomness generation we're able to replicate that exact same randomness when verifying the proof and the protocol execution.

Once we have this Trace generated we can proceed with both the views and commits generation. A commit will then be defined simply by hashing the originated view (plain trace).

4.3. Proof Validation

Given the above-mentioned description on the prover stage with the circuit execution, we get 4 files, 3 commit files and 1 file containing the final share from each party (not committed). As such, now, on the verifier end we'll choose one index $i \in \{1, 2, 3\}$ so that we can have access to the views i and $i + 1$.

Once we have access to this on the verifier side we'll run 3 checks as mentioned before:

4.3.1 Consistency Check

This consistency check as explained in a previous chapter is obtained by applying the follow steps:

Algorithm 3 Consistency Check ($View_1, View_2$)

$HashView_1 \leftarrow$ SHA-256$(View_1)$

$HashView_2 \leftarrow$ SHA-256$(View_2)$

Return $\rightarrow HashView_1 == Commit_1 \wedge HashView_2 == Commit_2$

Here, we're simply calculating the received view hash and comparing it with the corresponding public commit. This serves to check if the view has been adulterated compared to the moment it was first committed. It offers the verifier the insurance that the sent execution, either trustworthy or not, has not been tampered with after creation.

4.3.2 Execution Check

This secondary check can be slightly trickier, its goal is basically to ensure that the given 2 views reflect a trustworthy execution of the public circuit being executed.

The idea here is to take the first n shares of the view $i+1$ and populate them as input for a new party (let's call it p_r) that will try to reconstruct p_{i+1} execution. It's necessary to have 2 views so we can perform the operations that require inter-party communication. As such, after providing p_r with p_{i+1} first n shares, we leave it as is until it is time to check everything properly matches.

From there onwards we redo every step of the public circuit and fill the shares of p_r using the same randomness generated on the Prover's execution and finally compare each and every share of the the obtained new trace with the original p_{i+1} from the opened view.

Algorithm 4 Execution Check ($Trace_1, Trace_2$)

while $i \leq n$ **do**

$p_r \leftarrow Trace_2.get(i)$

end while

$Reconstruct(c, p_r, Trace_1)$ ▷ Were c is the public circuit in Bristol Fashion

Return $\rightarrow p_r == Trace_2$

4.3.3 Result Check

After running the previously mentioned checks, we check if the final shares provided by the Prover add up to the expected result (*True*, or 1):

Algorithm 5 Result Check($final_1, final_2, final_3$)

Return $\rightarrow$ UNSHARE$(final_1, final_2, final_3) == 1$

If this is the case we can then assume that the proof:

- Has not been tampered with.
- Reflects a trustworthy execution of the public circuit.
- The witness held by the prover holds for the relation we're trying to prove, meaning the Prover has indeed the knowledge to do so.

4.4. PSM

Having covered how the prover/verifier structure works and was implemented to ZKBoo for basic boolean operations, it's time to look at some proper circuits and focus on the scope of the problem, proving private set membership.

4.4.1 Approach

As before-mentioned, the problem consists in proving, in ZK, belonging to a given set. This represents the ability for a prover P to prove to a verifier V that he has ownership of a witness w such as $w \in S$, S being a private Set.

As you were able to read in Chapter 3 there are many approaches to solving this problem, but they all have one goal in common, to be modular, thus allowing them to easily build up to different problems and protocols.

In this work we've decided to obtain benchmark results by following the "naive" approach where having the following:

- S the private set we're trying to prove belonging.
- C a public commit group that is obtained by hashing private group S.
- x the piece of knowledge necessary to solve the problem. $x \in S$ and it'll be our witness in assembling the proof.

With this parameters we basically want to prove knowledge of a pre-image that belongs to C:

$$H(x) \in C \tag{4.2}$$

without (1) revealing x itself and (2) any additional information (like the index of C that x corresponds to).

This is easily achievable by masking the comparisons between the generated hash with each hash of the public commitment (c_i) group like so:

$$\neg(H(x) \oplus c_1 \wedge H(x) \oplus c_2 \wedge (...) \wedge H(x) \oplus c_n) \tag{4.3}$$

The above logical equation will evaluate to *True* iff $x \in C$. Now, to be able to feed this approach to our ZKBoo implementation we have to represent into a circuit that performs these operations on the bit level.

To be able to represent this into our protocol we'll need to be able to hash input but also add new operations that allow us to compare wires with fixed values (1 or 0) rather than other wire indexes.

4.4.2 SHA-256 Circuit and New Operations

As mentioned above, our first requirement is to be able to perform hashing in ZK. As such we've used an implementation by Nigel Smart et al. [5] where they define a circuit in their new Bristol Fashion that takes a bit-string of size 512 (message input) and returns its SHA-256 hash representation by using the set of boolean operations previously defined.

Then, to be able to compare every single output bit from the SHA-256 function with the hashes in the public commit we created a brand new operation called XORI defined similar to the previous ones with the following syntax:

- Number of Input Wires (always 2)
- Number of Output Values (always 1)
- Input Wire Index (can range from the any number in the wire range)
- Input Scalar Value (either 0 or 1)
- Output Wire Index

This leaves us with every necessary piece to assemble the final circuit that hashes and compares a given witness with the hashes of a public commitment.

4.4.3 Final Circuit Assembly

The problem in using a pre-existing circuit and building it into a new one is having to do the math beforehand to predict how the circuit's header is going to expand regarding both number of gates and wires.

The new header will so have the following increments for n elements in the set:

- $511n$ wires and gates for the 1st comparison stage: as we need to compare 256 bits from the hash output with 256 bits from the set C, we end up having for each comparison, 256 XORI gates + 255 OR Gates.
- $n-1$ wires and gates for the conjunction stage: after the 1st stage comparison we have to confirm that we've had 1 comparison turning back positive, as such we use conjunctions to group every result from the 1st stage, leading to having additional $n-1$ gates and wires.
- 1 final gate and wire as this represents the *inv* gate we add at the end so we end up with *True* for when x actually belongs in S.

That being said, the new header will always have an additional $512n$ gates and wires being generated as such:

```
1     private static int putHeader(int count, FileWriter fw) throws IOException {
2         (...)
3         int finalWires = numWires + count*shift; // count = n and shift = 512
4         int finalGates = numGates + count*shift;
5         (...)
6         fw.write(finalGates + " " + finalWires + "\n");
7         fw.write(finalInput + " " + finalState + " " + finalKey + "\n"); // These
      represent the input that'll be staying the same as we still just provide the
      element we want to hash
8         fw.write(finalOut + " " + finalBOut + "\n"); //Output wires are simply 1
     as we just want a 1 or a 0
9         (...)
10    }
11 }
```

From here we simply copy the whole hashing circuit so, given x as secret input it generates its hash value. Then we add the first stage of comparisons as described above:

```
1   private static void putCompare(int offset, int pivot, int i, FileWriter fw)
    throws IOException {
2       String TBC = pubGroup.get(i);
3       // ADDING XORIS
4       for (int j = 0; j < 256; j++) {
5           fw.write(2 + " " + 1 + " " + (offset + j) + " " + TBC.charAt(j) + " "
     + (pivot + j) + " XORI\n");
6       }
7       // OFFSET and PIVOT after XORIS
8       int xorEnd = pivot + 256; // End of the XORI results
9       // ADDING OR gates iteratively
10      int orOffset = pivot; // Starting point for OR gates input
11      int orPivot = xorEnd; // Starting point for OR gates output
12      for (int step = 256; step > 1; step /= 2) {
13          for (int j = 0; j < step; j += 2) {
14              fw.write(2 + " " + 1 + " " + (orOffset + j) + " " + (orOffset + j
     + 1) + " " + (orPivot) + " OR\n");
15              orPivot++;
16          }
17          orOffset = orPivot - (step / 2); // Reset offset to the beginning of
    the new layer of OR results
18      }
19  }
```

Finally we add the remaining AND gates and INV gate by applying the following algorithm to take into consideration when the number of elements is pair or not:

```
1   while(finORS.size()>1){
2           int lIndex = finORS.remove(0);
3           int rIndex = finORS.remove(0);
4           finORS.add(pivot);
5           fw.write(2 + " " + 1 + " " + lIndex + " " + rIndex + " " + (pivot) +
    " AND\n");
6           pivot++;
7
8       }
9       fw.write(1 + " " + 1 + " " + (pivot-1) + " " + pivot + " INV\n");
```

Once this script is executed for a given public commitment set of 256-long bit strings, we can feed it into the previously described implementation of ZKBoo and have the protocol run normally.

Taking this into a proper scenario, we can observe how a circuit grows for an increasing number of elements in a set S.

n	Gates	Wires
1	135585	136353
2	136097	136865
5	137633	138401
10	140193	140961
20	145313	146081
50	160673	161441
100	186273	187041
200	237473	238241
500	391073	391841
1000	647073	647841
10000	5255073	5255841

Table 4.1: Circuit Size for an increasing number of group elements

Note that the additional elements lead to an increase in the width of the circuit rather than the depth of it, meaning it can be parallelized to make the execution faster and more scalable.

5. Experimental Results

As previously mentioned, in this chapter we'll analyse the obtained results of the implemented instances of ZKBoo and PSM circuits and conclude on their suitability and how far these are to a real world application.

For a real world application we would expect for a fast protocol time, for both proving and verifying stages, that works with small and easily updatable general-purpose circuits (to reflect the forever changing character of a set in the PSM problem).

Inserting the instance in the context of the proposed use case, we can state that S represents the list or database containing all citizens of a given country (let's assume this list is represented by every national ID number of every citizen), x the individual aiming to prove citizenship and finally we mean to prove that $x \in S$ in the ZK fahion above described, so:

$$H(x) \in C \tag{5.1}$$

Where H represents the chosen hash function and C the public commitment group obtained by hashing each element of S.

When tackling a real world use case it's necessary to try and establish some constraints that reflect and frame the experimentation into an accurate (or scaled down) version of the reality.

In 2023, the average population per country in the world was 40.69 million, ranging from 1428.63 million people in India to 764 people in Vatican, as such, given the benchmark and ”naive” approach followed in this implementation, we've decided to obtain results up to $n = 10000$ as we've consider this to be a good scaled down threshold to represent quite small countries but also because with the intermediate values we believe we're reflecting some additional use cases' reality (like access control systems for some medium/large companies).

In the following sections we'll evaluate performance in 2 different stages:

- Offline Stage: This stange involves the preparation necessary for the proof to be built, in this case, the pre-computation of the public circuit that we'll be applying our secret input x that represents our national ID in the use case context.

- Online Stage: We refer to online stage, the stage of the protocol that ”requires”[1] communication between the prover and the verifier.

In this experiment we've considered the following setup:

- Device: Macbook Pro 14” 2021
- CPU: Apple M1 Pro
  - 10-core CPU
  - 16-core GPU
  - 16-core Neural Engine
- Language of Choice: Java
- Java Compiler: javac 17.0.6
- Java Version: OpenJDK 17.0.6

And for each value of n, performed 10 measurements, taking the average as a representative value for each of the set's sizes, regarding:

- User Time, that represents the elapsed time that the generation runs in ”user mode” without needing to access the underlying hardware.
- System Time which represents the elapsed time where accessing the underlying hardware is a necessity.
- Line Count (this one being constant regardless of the amount of measurements)

5.1. Offline Stage

In this stage we've obtained the below mentioned metrics for the following values of n:

[1] as ZKBoo can be made non-interactive as previously mentioned

N	User Time (s)	System Time (s)	Line Count
1	0,33	0,06	135589
2	0,32	0,04	136101
5	0,30	0,06	137637
10	0,32	0,05	140197
20	0,32	0,04	145317
50	0,34	0,07	160677
100	0,34	0,05	186277
200	0,38	0,06	237477
500	0,45	0,06	391077
1000	0,48	0,05	647077
10000	1,12	0,15	5255077

Table 5.1: Experimental results on circuit generation

We can easily notice that even following the naive approach, the produced results show some promising metrics, representing a linear growth with a really small growth factor, meaning that the instance of the circuit generator has a relative useful performance for even large sets, taking around 54 minutes to generate a circuit with an entire country's population ID (given the above mentioned average). Keep in mind that in the proving context, this can be considered a 1 time step as this public circuit does not need to be computed by any of the parties involved in the protocol.

Now, let's take a look at the experimental results regarding the generated circuit's size:

N	Size(MB)
1	3,6
2	3,6
5	3,6
10	3,7
20	3,8
50	4,2
100	4,9
200	6,3
500	10,3
1000	17,1
10000	147,8

Table 5.2: Experimental results on the generated circuit's size

As you can notice, although the size of the generated circuits also grows linearly, the growth factor in this case is quite bigger than in the circuit generation times. Taking this into consideration, to the average population per country, we would have a circuit with size around 57GB, making it quite large to be processed by the Prover and the Verifier and leading to rather slow proof times.

Some optimisations would offer some great improvement when addressing this size issue, such as:

- The use of MAND gates (described before, aggregate multiple AND gates into a single one).
- Using RSA Accumulators, that describe more complex yet less numerous computations, leading to shorter public circuits.

Looking now into the context of the problem where a citizen group is constantly changing for a variety of reasons, we can conclude with quite some level of confidence that the performance hereby demonstrated allows us to generate a daily circuit for proving citizenship. However, this also rises the need to optimise the generation of these to build them into smaller and more manageable files.

5.2. Online Stage

5.2.1 Prover

For the prover side of the communication we're measuring the performance on the process of:

1. Executing the Public Circuit and generating 3 Views with the traces of the 3 ficticious parties (heaviest part of the protocol as it involves running and recording the whole circuit execution)
2. Commiting these 3 views by hashing them and writing them into a file.
3. Opening 2 of these 3 commits.
4. Filesystem accesses that reflect the simulated ”communication” between prover and verifier as al the ”sent” information is simply written down in files to be later picked up.

For this stage we've obtained the following results:

N	User Time (s)	System Time (s)
1	0,99	0,08
2	1,04	0,09
5	0,98	0,07
10	1,00	0,08
20	1,01	0,07
50	1,03	0,09
100	1,20	0,09
200	1,33	0,11
500	1,86	0,16
1000	2,56	0,20
10000	15,98	1,22

Table 5.3: Experimental results on the prover stage

It is noticeable that once again, we're before a linear progression, this time however one with a quite rapid growth with a factor of $1,50E^{-3}$, leading to a proof generation time of around 16 hours for a group with $n = 4690000$ elements. It is worth noting, however, that for smaller datasets it actually performs quite well, taking only 2,56 seconds for a group with $n = 1000$ elements, making this a suitable approach for smaller countries such as the Vatican.

Some improvements that might help scale this approach into a more suitable one for big countries pass through:

- Shortening the circuit's size: by applying some of the improvements previously mentioned, we ensure smaller circuits and consequently smaller MPC executions, which lead to shorter and faster proofs.
- Using concurrent computation: by working with threads and computing in parallel, we enhance both performance (by taking better advantage of the CPU's usage and of the available computing power) as well as scalability (threads can be dynamically created and managed to efficiently utilise available resources, thus facilitating application scaling). We know that this work can be parallelised for as we saw in 4, the circuit grows essentially wider and not deeper.

5.2.2 Verifier

Finally, on the verification stage of the communication we're measuring the performance on the process of:

- Picking an index $i \in \{1, 2, 3\}$.
- Checking view consistency by hashing the views and comparing them to the original commits.
- Reconstructing the MPC execution resorting to the 2 opened views for 1 party only (heaviest part of the process as this involves reconstructing the whole circuit with the 2 provided views).
- Reconstruct the final shares to check if the obtained result is the expected one.

Our intuition here is that in this stage, has less computation cost than the proof assembly stage as we're reconstructing the execution just for 1 party and we do not need to break the secrets into shares.

N	User Time (s)	System Time (s)
1	0,78	0,06
2	0,79	0,08
5	0,78	0,07
10	0,77	0,05
20	0,78	0,06
50	0,86	0,07
100	0,87	0,06
200	1,01	0,08
500	1,39	0,12
1000	2,19	0,19
10000	14,60	0,87

Table 5.4: Experimental results on the verifier stage

Now although these results exhibit a quite similar behaviour as on the prover stage, we can notice a tendency to grow slower, making it more efficient with a smaller growth factor, leading to a verification time of around 1:48 hours for the average country population,

making it still not quite viable for real world scenarios of the proposed use case but definitely less costly than building the proofs themselves.

5.3. Comparison

Given the above mentioned experimental results, when putting side by side each of the three stages involved in this proof we get:

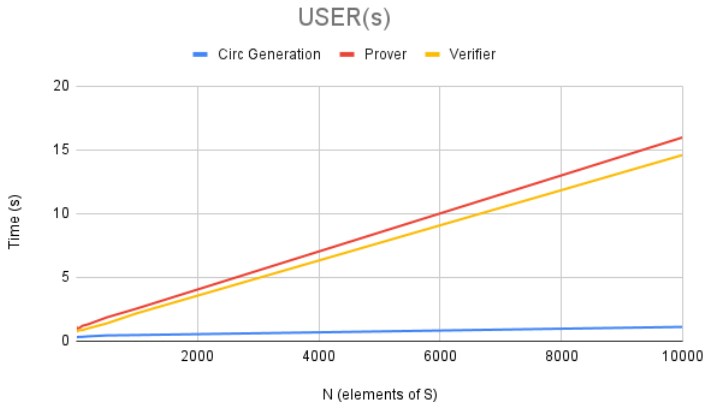

Figure 5.1: Time spent on each stage for $0 \leq n \leq 10000$

Through the chart represented in 5.1 it becomes apparent that even though this approach might be fit for small datasets, as n grows, the online stage's efficiency takes linearly more time to be executed, making it a good system for use cases with smaller sets, like privilege accesses in a file system or for resource accesses in an infrastructure, but not ideal for contexts with really large datasets as the proposed use case.

6. Conclusion and Future Work

In this final chapter, we'll look atthe results exposed in 5, conclude over its suitability in real world use cases like the one proposed in the beginning and finally understand what future work on this topic would look like taking the work described in this report as a starting point.

As the work described in this report reflects a benchmark ”naive” approach, it represents a good starting point to develop upon and experiment with more refined approaches, having a collection of protocols and optimisations, comparing their performance and conclude on their suitability to understand the best use cases for each chosen approach.

A proposed approach to follow up the work here would then be:

1. Start by introducing the optimisations described in 5 to have smaller proofs and fastest proving and verifying stages.
2. Experiment and instantiate more modern and intricate approaches like Ligero or KKW.
3. Compare the obtained results with the ones obtained in this benchmark, thus measuring the efficiency of the introduced optimisations.
4. Build the protocol into an actual application and test its suitability on an actual real world scenario (like a digital ID wallet or an identity and privilege manager)

References

[1] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof-systems,” in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC ’85. New York, NY, USA: Association for Computing Machinery, 1985, p. 291–304. [Online]. Available: https://doi.org/10.1145/22145.22178 [Cited on page 4.]

[2] J.-J. Quisquater, M. Quisquater, M. Quisquater, M. Quisquater, L. Guillou, M. A. Guillou, G. Guillou, A. Guillou, G. Guillou, and S. Guillou, “How to explain zero-knowledge protocols to your children,” in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 628–631. [Cited on page 4.]

[3] ”Nico”, “Zero-knowledge proofs decoded: A simple intro,” 2023. [Online]. Available: https://mightyblock.co/blog/zero-knowledge-proof/ [Cited on pages v and 5.]

[4] A. C. Yao, “Protocols for secure computations,” in *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 1982, pp. 160–164. [Cited on page 6.]

[5] N. S. et al., “’bristol fashion’ mpc circuits.” [Online]. Available: https://nigelsmart.github.io/MPC-Circuits/ [Cited on pages 9, 20, and 26.]

[6] D. Fiore, “Zero-knowledge proofs for set membership.” [Online]. Available: https://zkproof.org/2020/02/27/zkp-set-membership/ [Cited on pages 10 and 11.]

[7] J. Camenisch and A. Lysyanskaya, “Dynamic accumulators and application to efficient revocation of anonymous credentials,” in *Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22*. Springer, 2002, pp. 61–76. [Cited on page 11.]

[8] A. Nitulescu, “zk-snarks: A gentle introduction,” *Ecole Normale Superieure*, 2020. [Cited on page 12.]

[9] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Scalable, transparent, and post-quantum secure computational integrity,” Cryptology ePrint Archive, Paper 2018/046, 2018. [Online]. Available: https://eprint.iacr.org/2018/046 [Cited on page 13.]

[10] I. Giacomelli, J. Madsen, and C. Orlandi, “ZKBoo: Faster zero-knowledge for boolean circuits,” Cryptology ePrint Archive, Paper 2016/163, 2016. [Online]. Available: https://eprint.iacr.org/2016/163 [Cited on pages v, 14, 15, and 16.]

[11] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, “Zero-knowledge from secure multiparty computation,” in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, 2007, pp. 21–30. [Cited on page 14.]

[12] J. Katz, V. Kolesnikov, and X. Wang, “Improved non-interactive zero knowledge with applications to post-quantum signatures,” Cryptology ePrint Archive, Paper 2018/475, 2018. [Online]. Available: https://eprint.iacr.org/2018/475 [Cited on page 16.]

[13] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, “Ligero: Lightweight sublinear arguments without a trusted setup,” Cryptology ePrint Archive, Paper 2022/1608, 2022. [Online]. Available: https://eprint.iacr.org/2022/1608 [Cited on page 18.]