UNIVERSITÀ DI PARMA

Dipartimento di Ingegneria e Architettura
Corso di Laurea in Ingegneria Informatica, Elettronica e delle Telecomunicazioni

Protocolli Zero-Knowledge per Strategie Avanzate di Proof of Location

Advanced Proof of Location Strategies Based on Zero-Knowledge Protocols

Relatore:
Chiar.mo Prof. Michele Amoretti

Correlatore:
Ing. Gabriele Penzotti

Tesi di Laurea di:
Nicolò Zarulli

ANNO ACCADEMICO 2021-2022

To my parents, Bianca and Maddalena.

"If you want to keep a secret, you must also hide it from yourself."
George Orwell, 1984

Acknowledgements

Words cannot describe the gratitude I have for being in the position of writing this page and having the possibility to close this important chapter of my life. It has been a challenging, tumultuous, and never boring path but it has made me the person I am today, and I'm grateful for every single second of it. I would like to express my gratitude to my Thesis advisors, Ph.D. Prof. Michele Amoretti and Eng. Gabriele Penzotti, for their invaluable feedback and knowledge to pursue months of research and the writing of this Thesis. I have heavily thought about all the people in my past and present life, and each one of them has been part of this journey and has helped me attain the goals I could have never fathomed reaching. I would like to dedicate every single word of this Thesis to the people I love most. To my parents, Antonio and Catia, thank you for always believing in me and granting me unconditional love, for whatever path my life will take to be. To Antonella, Ivan, Jacopo, Mattia, Maura, Marcello, Simonetta, Roberto, Renzo, and to all family, thank you for allowing me to be me. To my not-by-blood sister Marta, since we were little you have always been a point of reference for me, and I will always be there for you. To my closest friends, Eros, Filippo, Luca, Noemi, Sofia, course-mates Davide, Ivan, and all the others I could not include in these paragraphs, I dedicate these words: you may not know it, but you made it possible, countless times, to not give up by always giving me a reason to get up from bed at my worst times; thank you. To the ones who were here, Bianca, Maddalena, Giuseppina, I would have loved for you to read these words, but I know in a way you are. You're always in my heart.

Le parole non possono descrivere la gratitudine che provo nel trovarmi ora a scrivere questa pagina, marcante la fine di un importante capitolo della mia vita. É stato un percorso stimolante, tumultuoso e mai noioso che mi ha permesso di essere la persona che sono oggi, e sono grato di ogni suo secondo. Vorrei esprimere un sentito ringraziamento al Professore Michele Amoretti e all'Ing. Gabriele Penzotti per il loro fondamentale supporto durante i mesi di ricerca e durante la stesura di questa Tesi. A lungo ho pensato a tutte le persone che hanno fatto parte e sono parte della mia vita, se sono arrivato alla conclusione di questo percorso, è grazie ad ognuna di loro. Vorrei quindi dedicare ogni singola parola presente in questa Tesi alle persone che amo di più. Ai miei genitori, Antonio e Catia, grazie per credere sempre in me e garantirmi amore incondizionato, qualsiasi percorso la vita mi porterà ad intraprendere. Ad Antonella, Ivan, Jacopo, Mattia, Maura, Marcello, Simonetta, Roberto, Renzo, e tutti i miei familiari, grazie per rendermi sempre possibile di essere me stesso. A Marta, da sempre sei il mio punto di riferimento, ed io per te ci sarò sempre; sei la sorella che non ho mai avuto. A tutti i miei amici più stretti, Eros, Filippo, Luca, Noemi, Sofia, ai miei compagni di corso Davide, Ivan e a tutti coloro che non ho potuto inserire in questi paragrafi, dedico queste parole: potreste forse non saperlo, ma avete fatto in modo che, nei miei momenti peggiori, io riuscissi ad alzarmi dal letto e non mollare mai; a voi dico grazie. A coloro che ora sono presenti solo nel mio cuore, Bianca, Maddalena, Giuseppina, avrei voluto fosse possibile per voi leggere queste mie parole, ma so che in qualche modo riuscirete a farlo; sarete per sempre parte di me.

Contents

Introduction

Nowadays, the value of personal data has grown exponentially thanks to the birth of new technologies that easily allow the extraction of valuable information and the consequential rise of new businesses. The interest in these data has grown so much that, the concept of *cybersecurity* cannot be overlooked. That is because it could be extremely easy to obtain the entire history of someone's private and sensible data when security approaches are not established and followed strictly.

A type of data that is highly used and in demand in the industry is the user's geographic information, which is necessary for Location-Based Services (LBS). Without such kinds of data, applications like navigation software, social networking services, tracking systems, and location-based advertising, media, or games could not exist. As such data are heavily required, it is fundamental to guarantee user privacy during the assessment of the Proof of Location (PoL) problem, which requires the use of special technologies to validate the geographic information managed.

It is therefore very important to viable solutions the problem of adding a privacy layer to PoL systems, which give a user the ability to prove to another user they lie in a specific location without disclosing their geographic information. Therefore, it is important to find a binding point between PoL and cryptography to reach such security goals.

This Thesis is the result of a period of curricular internship at the Distributed Systems Group (DSG) of the Department of Engineering and Architecture at the University of Parma, focusing on searching for a satisfactory answer to the previously stated problem. The cryptographic method that comfortably responds to the necessities of such a problem, is the Zero-Knowledge Proof (ZKP). ZKP is quite an unexplored but relevant cryptographic method that lets a user prove the knowledge of secret information with the use of a proof that does not reveal anything about the secret. ZKP consists of many protocols that implement this concept and many of them are mainly used in a decentralized environment, i.e., blockchains. Various strategies used to bind the PoL problem with ZKP Protocols have been studied and implemented.

The Thesis consists of four chapters that gradually take the reader from the theory that regulates these technologies up to the implementation of advanced PoL strategies based on ZKP Protocols. The chapters are summed up as follows:

1. **State of the Art**: Introduces the theory behind the PoL problem, ZKP, and the main ZKP Protocols.

2. **Zero-Knowledge Proof of Location Strategies**: Presents the generic structure of a zkPoL strategy and proposes specialized structures based on different ZKP Protocols.

3. **Implementation of zkPoL Strategies**: Outlines the implementation of two zkPoL strategies, which are the pyZKP library and the zkSNARK DApp.

4. **Performance Testing and Evaluation**: Reports the performances of the implementations and evaluates the results.

Chapter 1

State of the Art

The main goal of this Thesis is to find a binding point between the technologies that implement Zero-Knowledge Proofs with the Proof of Location problem. Based on that premise, it is important to introduce the theory behind these concepts and technologies. The “State of the Art” chapter wants to do just that, where each section takes one argument and tries to lay the foundations to better understand the implementation of such strategies assessed in the subsequent chapters.

Therefore, the section will evaluate the following topics:

- **The Proof of Location Problem** [**PoL**]: An introduction to the problem, why it is a relevant topic today, how it can be dealt with and how it relates to this Thesis.
- **Zero-Knowledge Proof** [**ZKP**]: What is ZKP, what its basic structure is, how does it work, and the basic classification of various models.
- **Zero-Knowledge Proof Protocols** [**ZKP Protocols**]: An analysis of actual and useful implementations of ZKP and how they can be implemented in various application domains.

1.1 The Proof of Location Problem

A *Proof of Location* can be seen as a digital certificate that attests to someone's presence at a certain geographic location and at a certain time, as it can be seen on a publication from *Amoretti et al.* [1]. The problem related to this concept is that, in such a context, it is essential to verify that the geographic locations claimed by users are trustworthy, but it is equally important to preserve user privacy preservation.

Generally speaking, a user has to obtain certain geographic information and must deploy them to a trusted system to obtain a PoL certificate. To obtain geographical information, the user could use various *Location Services (LSs)*, which also provide important location-dependent functionalities. Once the user has the geo-data needed, it has to go through a decentralized PoL system, which processes that data and computes a certificate.

The implementation of a decentralized PoL system may be either infrastructure dependent or infrastructure-independent. The cited publication [1] proposed a completely decentralized and infrastructure-independent PoL scheme for LBS-oriented peer-to-peer networks. The usage of a decentralized P2P system guaranteed the user a greater level of privacy, as there is no need for a central authority that knows both the geographic location of the user and the information which is being exchanged. Having said that a P2P architecture is needed, the perfect technology that suits this concept, and is going to be used as a strategic key point later in this Thesis, is the *blockchain*. The main features that blockchains bring to the table are, in fact, their decentralized nature, the property of immutability of the data stored in the blockchain, and the availability of a distributed database to the peers that contribute to the ledger.

Even though the PoL problem should be assessed in its entirety, for the purpose of this Thesis, the focus lies on the part of the computation of the PoL certificate through the implementation of strategies based on Zero-Knowledge Protocols. So, it is not essential to be sure *how* a user got the knowledge of certain geographical information, but it is more important to find a binding point, for the technologies involved, that lets the user compute a proof, which is going to be referred to as a certificate that proves to another actor of the protocol the knowledge of secret geographical information, without revealing any information about the secret itself.

1.2 Zero-Knowledge Proof

In an era in which technology evolves fast, the concept of Zero-Knowledge Proof (ZKP) finds its uses directly in the context of Computer Science: Cybersecurity, Blockchain Transactions, and Biometric Authentication are simple examples of how versatile and useful this technique can be. Indeed, it is essential to explain *what* Zero-Knowledge Proof is, and *how* it relates to the systems this Thesis is based on.

Zero-Knowledge Proof is best described as a cryptographic approach by which an entity, known as *Prover*, is able to prove to own sensible data, known as *Secret*, to another entity, known as *Verifier*, with the exchange of a *Proof*. The key point of this approach is that the demonstration mechanism between the two entities never reveals any relevant information about the secret itself, especially about the personal data owned by the entities at stake. This idea is found at the *Proof* level: it is directly linked to the secret, but must not reveal it by any means necessary. A basic Zero-Knowledge Proof scheme can be seen in Figure 1.1.

1.2.1 Structure

Following a brief introduction of what *Zero-Knowledge Proof* is, it is important to point out *how* this cryptographic method works and *how* a basic structure is designed, as the implementation of a *Zero-Knowledge Proof* technique can be defined as a *Zero-Knowledge Proof Protocol*. As the definition of protocol suggests, parameters and rules must be established in order to prepare a fair field for every party involved in the exchange. A generic *ZKP Protocol* can be summed up as an *Interactive Verification Protocol* with the

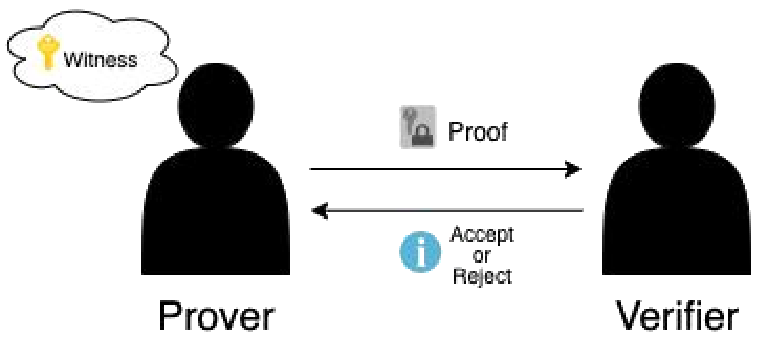

Figure 1.1: Zero Knowledge Proof Schema

following characteristics:

- Two entities are involved in the verification.
- Standard goals must be achieved.
- A synchronous message exchange is required.
- A three steps algorithm is performed.

The entities involved in the protocol were already briefly discussed; therefore a more detailed explanation of *what* role they play in the context must be then provided.

Prover : Knows a certain *secret* for which a reward is at stake. Its main priority is to try and obtain the reward without revealing the secret.

Verifier : Guarantees a reward for the knowledge of a personal secret and its priority is to be sure that a Prover really knows that secret. Before paying any kind of reward, the Verifier wants the Prover to reveal the secret at stake.

A clear behavioral pattern stands out: both parties will not give up their confidentiality before gaining something from the exchange, whether it be the reward or the secret; that is where the concept of *Zero Knowledge* steps in. With the integration of an *Interactive ZKP Protocol* into the communication protocol, a Prover would be now capable of exchanging *Statements*, associated with a secret, with a Verifier without taking a chance on leaking any sensitive information; moreover, the Prover would be able to obtain the reward without giving up the secret, and the Verifier would be sure that the Prover knows said secret and pay a reward for it.

Every *ZKP Protocol* must ensure three standard goals to instantiate a fair playground for each party involved, as listed below:

Confidentiality : The *Proof* sent from a Prover to a Verifier must prove the knowledge of a certain secret, but must not reveal any sensitive information about it. Moreover, no private data must be revealed at the communication level.

Integrity : The exchange of messages is carried on with the usage of *Statements*, both from the Prover and the Verifier. Each party involved in the protocol must ensure that its personal statements will not lose integrity from a malicious attacker.

Authenticity : A valid Proof can be computed from a party based on the premise of the knowledge of only a certain secret.

Having established the foundations of a standard ZKP Protocol, it is essential to outline the execution of such protocol, which follows a three steps algorithm to carry out the verification process [2], which can be seen in Figure 1.2 and listed below as:

1. **Witness Phase**: The Prover computes a **Proof**, on the basis of the knowledge of a *secret*, containing an initial *Statement*. The Proof is then sent to the Verifier as a starting point for the verification exchange messages.

2. **Challenge Phase**: After having obtained the Proof, the Verifier starts to ask questions, posed as *Challenges* to the Prover. This phase is

essential to the Verifier, as it serves as a tool to verify the intentions of the Prover.

3. **Response Phase**: The Prover receives said Challenges, one at a time, and computes an answer, or *Response*. The Response is then passed to the Verifier, which will assess the validity of the answer, and based on its own satisfaction, the Verifier will decide to account the Proof as **valid** or **reject** it.

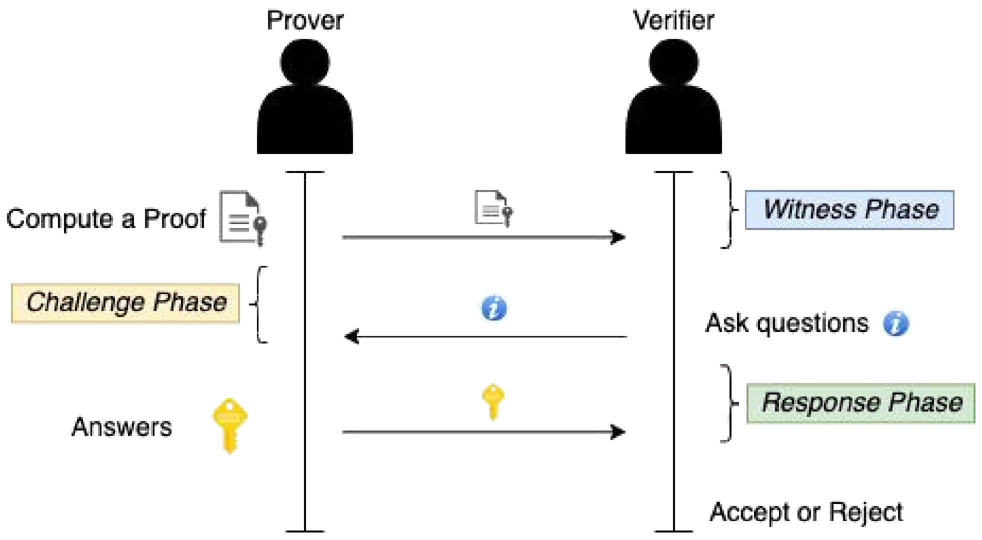

Figure 1.2: Interactive Zero Knowledge Protocol

It is essential to point out the direct consequences of this approach:

- If the Prover actually knows the secret the Verifier is verifying for, they will always be able to compute *true* statements in response to the challenges. Otherwise, in the event the Prover is malicious and does not own the secret, in the short term there is the possibility that it will be successful in computing a valid Proof, cheating the Verifier; however, in the long run, this probability decreases in such a way that it is highly unlikely (or impossible, based on the implementation) to fool the Verifier and make a successful Proof without knowing the secret.

- The Verifier can run a new cycle of the algorithm, if not completely satisfied with the response received.
- The computational resources involved on the Prover side could be costly; it depends on the mechanism used to implement the *Witness & Response* phases.

1.2.2 Properties

The concept of a *Zero-Knowledge Proof System* was first introduced in a published paper [3], from MIT researchers Goldwasser, Micali & Rackoff, where that said system is addressed with much more detail and cardinality. It can be said that by the means of an **Interactive Proof System** is defined as a model, by which a Prover exchanges messages with a Verifier to convince him that a certain *Mathematical Statement* in their posses is valid.

A sensitive topic, analyzed and dissected from the work of the researchers, concerns its focus on the property of *integrity* of the communication channel, which has to be kept valid to run a ZKP system. It is in fact possible that a *malicious* Prover could try to deceive an *honest* Verifier, providing a *false* Proof and living off of Probability Theory to play their cards with. The Interactive System previously described places the Prover in an advantaged position over the other party, the Verifier: *the Prover has nothing to lose, but everything to gain.* That is a key point in the study of ZKP, in regards to the implementation of an associated decentralized system. But what can be said about the Prover, can be equally inverted to the point of view of the Verifier: *What if a Prover could not be sure to trust a Verifier and exchange messages with them?* Both perspectives are equally important and **non-negligible**.

What was described in the last paragraph is the topic of **information leakage**, which deals with the assessment of how much detail needs to be included in the exchange of statements between the parties involved in the protocol. By bringing this concept to a ZKP Protocol, both parties must abide by and follow a set of three essential properties, namely:

Completeness : If the *statement* of the Proof is **true**, an honest Prover will always be able to convince an equally honest Verifier. Based on that premise, the probability associated with the successful execution of the algorithm is required, by design, to be as close as possible to 100%.

Soundness : If the *statement* of the Proof is **false**, a malicious Prover will never be able to deceive an honest Verifier into accepting said statement. Based on that premise, the probability associated with the successful deception of the Verifier is required to be as low as possible.

Zero Knowledge : Guarantees that the Prover won't leak any private and unnecessary information into the communication channel, to any Verifier. If the statement is found to be truthful, no Verifier will be able to know anything other than that data. Said property also gives the name to the cryptographic method.

1.2.3 Complexity

The same publication [3] discusses in the previous section, not only focuses on the definition of a set of strict properties that a ZKP Protocol must abide by, but also proposes a *hierarchy* of Interactive Proof Systems and assesses the concept of **knowledge complexity**. Said concept discusses the measurement of the amount of knowledge about the proof transferred from the Prover to the Verifier [4]. The researchers also outlined the first Zero-Knowledge Proof for a concrete problem, that of proving if a number is *quadratic non-residue modulo m*, which lies in the **NP-complete** subclass of Optimization Problems, as $NP\text{-}C \in NP$. This section aims to introduce the complexity class in which most ZKP Protocols operate, i.e., the class of *Non-Deterministic Polynomial Time Optimization Problems.*

Nondeterministic Polynomial Time Problems

An Optimization Problem R lies in the class of *Nondeterministic Polynomial Time (NP)* when, knowing an optimal solution to the problem, an optimal value can be computed in polynomial time. An equivalent definition[5] of NP can be linked to the concept of a *nondeterministic Turing machine*:

> An Optimization Problem belongs to the complexity class NP when its optimal solutions can be solvable in polynomial time by a non-deterministic polynomial Turing machine; in the presence of a pre-determined status and a pre-determined read character, said machine can perform multiple operations/transitions at the same time.

The NP complexity class represents a large set of problems: it does not matter with what complexity an algorithm computes an optimal solution, what it matters is that the computational time spent to obtain an optimal value from it *must* be polynomial. A different situation lives for the complexity class *Polynomial Time (P)*, which requires that an algorithm A computes a solution for an optimization problem R strictly at a polynomial time. It is essential to point out a relation between the two classes: $P \in NP$.

When talking about ZKPs, we are looking at a cryptographic primitive which is based on the computation of an *NP Statement*, an instance of a certain NP problem. Therefore, it can be said that a Prover P wants to prove to a Verifier V that they know a valid solution W for a certain public NP problem, omitting any information about the solution itself; thanks to the computation from P of a personal NP statement, that can be privately verified from V, the precedent hypothesis can be then proved true or false.

1.2.4 Classification

Up until this point, a ZKP Protocol has been described as an Interactive Proof System, taking for granted that this model would be optimal in any context, with any computational problem to be solved. An immediate flaw of the said model concerns the type of communication established between the two parties involved: it is generally used an exchange of *synchronous* messages, like a *question & answer* basic schema. With the progress made in the last two decades, and the shift towards a more decentralized structure of the Internet, said interactive structure poses a lot of issues, based solemnly on the property of **interaction**.

And by that said interaction, can be introduced a more concrete and persistent ZKP Protocol model, more suitable to the current times: it is a **Non-Interactive ZKP Protocol**.

That being said, the property of interaction makes it possible to obtain an essential, but brief, classification of Zero-Knowledge Proofs:

- **Interactive Zero-Knowledge Proof** [**I-ZKP**]: it is built on a *synchronous and direct communication* between two parties. As the model provides, the Verifier makes *tasks*, or questions, available to a Prover, who has to answer with *statements* to prove the knowledge of a Witness. The computational resources involved on the Prover side are much more than those on the Verifier one.

- **Non-Interactive Zero-Knowledge Proof** [**N-I ZKP**]: With this model, the Prover is able to prove statements to a Verifier by sending him a single message. There is no need to have a synchronous and direct communication channel, which requires the two parties to interact several times, which reduces the complexity of the channel itself and guarantees more privacy and security between the entities at stake. The model also introduces a new actor: it is a **trusted third party**, known as **Setup**, who can share personal software and additional computing resources with the Prover and the Verifier. Figure 1.3 outlines a basic N-I ZKP.

The latter approach is the most suitable when a decentralized system seeks the implementation of a ZKP Protocol because, thanks to the non-interactive nature of the model, it is not required of the parties to communicate at the same time, also making it possible that a Proof could be computed and verified at a later stage. There are various models available and each one of them has its own advantage and disadvantage. To analyze and choose a certain *N-I* model over the others, the most important property to be considered is the *knowledge complexity* weighed on each party: based on the goals of the system where a ZKP has to be implemented, it is fundamental to be sure to know which party has to consume the most resources to run the protocol; based on the algorithms and mathematical structure used, it could be either at the Prover side, at the Verifier one or directly involving the dimension of the Proof sent through the channel. The Non-Interactive models that are going to be analyzed and used throughout this Thesis, to lay the foundation of a *Zero-Knowledge Proof of Location* strategy architecture, are *zkSNARK & Bulletproofs*.

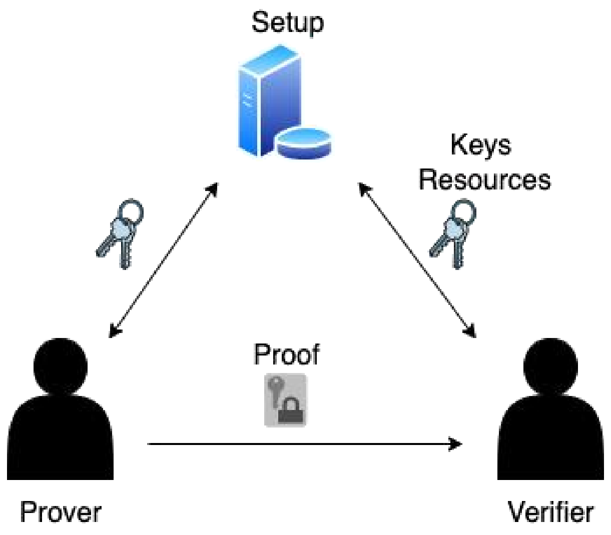

Figure 1.3: Non-Interactive Zero Knowledge Protocol

1.3 Zero-Knowledge Proof Protocols

It is now important to take a more strict focus on various and existing implementations of Zero-Knowledge Proof schemes. Each protocol that is analyzed in this section refers to either an Interactive ZKP architecture or a Non-Interactive ZKP one. As of today, the concept of I-ZKP remains more *theoretical*, whereas the N-I ZKP has multiple and concrete implementations, mainly aimed at decentralized systems. The Interactive-ZKP model is not as useful as an N-I ZKP can be when applied to the context of a decentralized system; nonetheless, before taking an in-depth look at the N-I ZKP protocols used in this Thesis, it is equally important to outline some cases in which an interactive approach could be applied, such as:

- **Knowledge of a k-coloring graph**: A k-coloring problem for undirected graphs is an assignment of colors to the nodes of the graph such that no two adjacent vertices have the same color, and at most k colors are used to complete color the graph. Such a problem belongs to the complexity class NP and it is based on *graph theory.* To approach this problem an Interactive ZKP model could be used, where a Prover and a Verifier share the knowledge of a public graph and the Prover wants to demonstrate to the Verifier the knowledge of an algorithm that solves this problem, without revealing how the algorithm works. The interaction between the two actors is essential: the Verifier has to be able to ask the Prover various statements that prove its knowledge of the secret.

- **Knowledge of a discrete logarithm of some residual modulo p**: Given a public prime number g, known as *generator*, a public prime number p, known as *modulo*, a certain *residual* r and a secret value x, that $g^x = y \pmod p$, where the value y is a primitive root of p. This problem is used to implement an interactive ZKP strategy, based on the *Discrete Logarithm Problem(DLP).*

1.3.1 zkSNARK

An improved mechanism based on the Non-Interactive ZKP Model is **zk-SNARK**, an acronym for *zero-knowledge Succinct and Non-interactive ARguments of Knowledge*. What is essential about this model is the introduction of a new property the protocol has to abide by it is the property of **succinctness**, which will be assessed in detail later on in this section. This model was first introduced in 2012, in a publication of *Bitanksy Nir et al.* [6], and since then has found major implementations in the world of Blockchain: the *Zerocash protocol* (www.zerocash-project.org) was the first to take advantage of the concept of zero-knowledge proof; by the usage of a zkSnark architecture, the engineers made it possible to nodes that participate into the protocol and posses or trade cryptovalues, to create and manage *mathematical proofs* to prove the knowledge of a related type of information, such as the possession of a certain amount of coins to carry out a transaction.

Preliminaries

zk-SNARKs are the most used ZKPs because they are short and succinct: the proofs can be verified in a few milliseconds. However, they require a trusted setup where some public parameters are generated. As it can be seen from a publication [7] by Salleras-Daza, these characteristics are inherited from the ZKP construction scheme used to guarantee basic parameters to run the protocol. The first scheme used to construct a zkSNARK was introduced in the BSCTV13 [8] protocol but was later improved by the zk-SNARK introduced in the Groth16 [9] protocol, which is still one of the most efficient zkSNARK schemes, especially when it comes to the verification algorithm. The Groth16 scheme is also used to construct a zkSNARK for the implementation of a Proof of Location strategy, assessed in the next chapters of this Thesis. Even though the construction scheme Groth16 brings advantages to zkSNARK, it requires the implementation of a *trusted setup*, which is a phase where some public parameters are generated and will be used to generate and verify proofs. The *soundness* property of this scheme relies on a specific **security assumption**, as each ZKP scheme does, and in the case of zkSNARKs, the construction uses a strong assumption called *q-Power Knowledge of Exponent (q-PKE).*

Regarding the security of the construction of zkSNARKs, it mainly relies on the security of elliptic curves. Breaking the security of the elliptic curve used by a specific construction would lead to being able to generate false proofs and thus, breaking the soundness property of the scheme. The zk-SNARK construction used for this Thesis requires a pairing-friendly elliptic curve E over a finite field $\mathbb{F}p$, where p is a prime number.

The q-PKE assumption requires three different technologies to construct a zkSNARK: an **Arithmetic circuit**, a **Rank 1 Constraint System (R1CS)** and a **Quadratic Arithmetic Program**.

- **Arithmetic circuit**: A circuit is a directed acyclic graph composed of different wires and gates, which lead to a set of equations relating the inputs and the outputs of these gates. The inputs and the output of this circuit, as well as the operations defined in the gates, are elements over a prime field $\mathbb{F}r$, where r is the order of E.

- **Rank 1 Constraint System (R1CS)**: A Rank 1 Constraint System (R1CS) is a system of equations that checks the correctness of all the operations in a circuit, by grouping them in constraints. Each constraint is composed of a multiplicative gate with its two inputs and its output. Having a statement u and a secret w, a R1CS is defined as a set of vectors (a, b, c) defining the circuit, with vectors $a, b, c \in \mathbb{R}^n$, n number of gates, whose solution is a vector $s = (u, w)$, with $u, w \in \mathbb{R}^n$, such that the following equation is satisfied:

$$< a, s > \cdot < b, s > - < c, s >= 0$$

- **Quadratic Arithmetic Program**: Quadratic Arithmetic Program (QAP) is a polynomial representation of the R1CS, which bundles all its constraints into one.

Protocol Structure

As said before, the interaction between a Prover and a Verifier is managed from a *trusted third party* known as **Setup**. The Setup not only serves as a communication bridge between the two parties involved in the protocol but, as the N-I model suggests, makes available to the parties additional circuit and software resources. In the case of zkSNARK, the model requires three parties to be involved in the protocol: a **Prover P**, a **Verifier V** and a **Setup S**. It also introduces a new concept that takes ZKP a step forward and makes the asynchronous communication possible: a pair of keys, known as **Proving Key [PK]** and **Validation Key [VK]**, are now needed to P and V respectively, to send or verify a Proof.

- **Proving Key [PK]**: used by P to compute a verifiable proof π.
- **Validation Key [VK]**: used by V to verify a certain proof π associated to a PK.

These keys are generated and distributed by S to the other parties involved in the protocol. The generation part of the key is carried out from a specific *generation GK algorithm*, which takes as input a relation $\mathcal{R}$ composed of an elliptic curve E over $\mathbb{F}p$, a secret randomness λ and a QAP (constructed from the knowledge of a public arithmetic circuit C). zk-SNARKs are composed of three main algorithms, each one available to a specific party:

+ **Key Generator KG [Setup]**: $KG(\mathcal{R})$
+ **Proof Generator PG [Prover]**: $PG(\mathcal{R}, PK, x, W)$
+ **Proof Validator PV [Verifier]**: $PV(\mathcal{R}, VK, x, \pi)$

A brief summary of the parameters that refer to formulas and expressions found in the previous paragraph and in the “Entities” section is now presented.

Summary:
F: is a multiplicative field of $\mathbb{Z}p$
λ: *secret randomness*
C: *arithmetic circuit, input & output* $\in F$
R: a relation composed of a an elliptic curve E over $\mathbb{F}p$, λ *and C.*
PK: *proving key,* $PK \in F$
VK: *validation key,* $VK \in F$
x: *P public input, hashed value* $x \in F$
W: *P secret input, witness value* $w \in F$
π: *proof computed from P, proof value* $\pi \in F$

Entities

It is essential to outline the behavior of each party involved in the zkSNARK protocol, as they are directly linked to the algorithms involved at each layer of the model.

Setup S : Generates a pair of keys (PK, VK) and then distributes them to a Prover P and a Verifier V. To generate the pair, it executes the generator algorithm GK, with inputs parameters that form a relation $\mathcal{R}$.

$$(\text{PK, VK}) = KG(\mathcal{R})$$

Prover P : Generates a **proof** π, by multiplying x and W by some polynomial provided from $\mathcal{R}$. The prover also needs to computethe coefficients h of $H(x)$, which can be achieved in $O(nlogn)$ using Fast Fourier Transform (FFT) techniques[7].

$$\pi = PG(\mathcal{R}, PK, x, W)$$

The proof is then forwarded to a Verifier. Said proof demonstrates to the Verifier that the Prover, who generated it, has the knowledge of a certain *secret W* and that it satisfies a *public arithmetic circuit C*, such as

$$C(x, W) = True/False$$

Verifier V : Receives a **proof** π from P, and a VK from S. With the knowledge of these two components it can verify π though the proving algorithm PV, such as

$$PV(\mathcal{R}, VK, x, \pi) = True/False$$

then, $PV(\mathcal{R}, VK, x, \pi) == (\exists W \mid C(x, W))$

Succinctness

The property of *succinctness* is what makes the zkSNARK protocol extremely useful blockchain-wise. The definition[1] of succinctness says

> *"the quality of being expressed clearly and without unnecessary words."*

Therefore, in relation to the zkSNARK protocol, said property requires that the Proof sent through the channel must be **compact** in size and **easily** verifiable. This property is essential when a Zero-Knowledge architecture expects a type of communication extremely costly, but it is also needed when a Verifier is weak in terms of computation resources.

An important characteristic of zkSNARK is that the protocol suits all problems in the complexity class NP and, actually, the practical zkSNARKs that exist today can be applied to all problems in NP, but it is unknown whether there are zkSNARK implementations for any problem outside of NP. It's easy then to find a generic zkSANRKS for all problems in NP: the only thing needed is to choose a suitable problem that belongs to the complexity subclass NP-Complete (NP-C), as $NP\text{-}C \in NP$. That is because the NP-C problems represent the hardest problems in NP. An NP-C problem is a problem for which the correctness of each solution can be verified quickly and can be used to simulate every other problem for which a solution can be quickly verified as correct; meaning that if some NP-Complete problem has a polynomial-time algorithm, all problems in NP do. Although a solution to

[1] definition from "https://dictionary.cambridge.org/dictionary/english"

an NP-complete problem can be verified "quickly", there is no known way to find a solution quickly. That is one of the reasons why, in a zkSNARK protocol, the complexity of the proving algorithm is always more expensive, in terms of resources, compared with the verification one, as can be seen in Table 1.1. For the reasons mentioned before, the property of succinctness is highly required when problems, belonging to the complexity class NP, are involved in the protocol.

Based on all of these premises and characteristics, the zkSNARK protocol suits really well the concept of decentralized architectures, as the three parties involved are not dependent on each other but communicate with one another to obtain a certain type of information to carry on a proving/verification process. The Blockchain domain benefited greatly from the publishing of this protocol, mainly because of its *succinctness* property, which gives the proof a very short size. Said size makes the exchange through nodes and a decentralized ledger extremely fast, compared to a general Interactive-ZK protocol; moreover, due to the fact that any Verifier in possess of a true Validation Key can verify a certain proof, it is possible for a Blockchain to store that proof and make it available to multiple verification parties, that consequently do not have the obligation to contact a Prover directly. Figure 1.4 delineates a simple zkSNARK schema.

Features

Based on the construction schema used to implement a zkSNARK, specific complexity requirements are outlined. The computing complexity of these algorithms depends on the number of operations that we do in the circuit, which is also the the number of gates n. In Table 1.1 are shown the complexity requirements for the zkSNARK construction schemes BSCTV13 and Groth16.

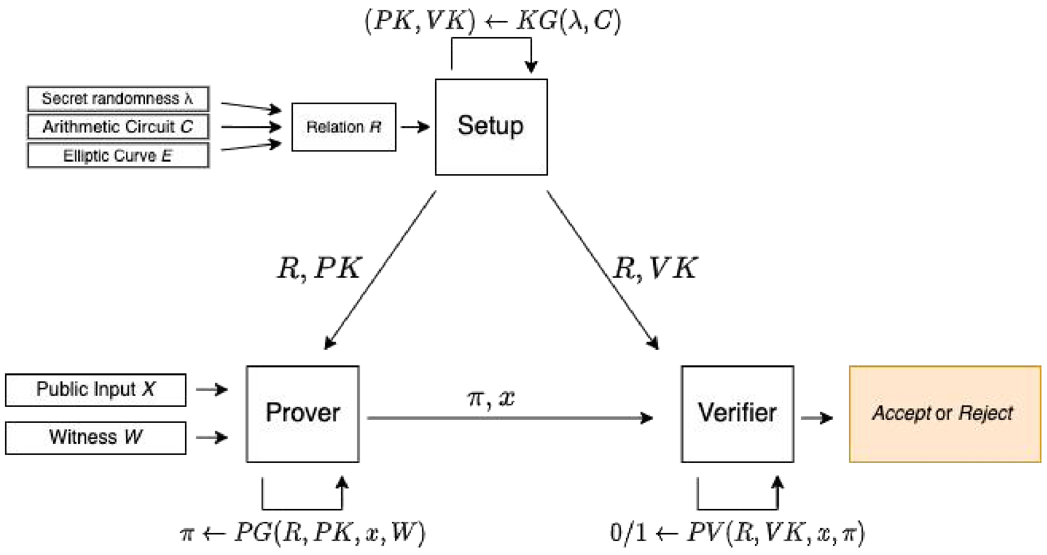

Figure 1.4: zkSNARK Scheme

zkSNARK Features					
Scheme	**TS**	**Prove**	**Verify**	π	**Security Assumption**
BSCTV13	yes	$O(n \log n)$	$O(1)$	$O(1)$	q-PKE
Groth16	yes	$O(n \log n)$	$O(1)$	$O(1)$	q-PKE

Table 1.1: zkSNARK Features Table

Using Groth16, the Setup has a complexity of $O(n)$, so the computation involves a number of elements that depend only on the size of the circuit. Regarding the Verifier, where the complexity time is constant $O(1)$, who only needs to compute 3 pairings and verify that an equation holds, which is not expensive in terms of power consumption. The operations which require more power consumption are done by the Prover when computing the proof.

1.3.2 Bulletproofs

An efficient Non-Interactive ZKP Protocol is *Bulletproofs.* The related paper is credited to *Bunz et al.* [10], and lays down the foundations for said schema as it is a groundbreaking step forward with respect to the efficiency of the proving/verification process. In relation to this Thesis, a key aspect that Bulletproofs introduces is the concept of *Zero-Knowledge Range Proof*, making it possible to prove that a secret committed value lies in a given interval. Bulletproofs have been recently implemented for a few privacy-oriented cryptocurrencies, including Monero (`www.getmonero.org`), to reduce the range proof sizes.

Preliminaries

Compared to zkSNARKs, Bulletproofs do not require a trusted Setup and are much more useful when a Prover needs to compute a Range Proof (it occupies at most 600 bytes) instead of an arithmetic circuit. The construction scheme used to implement Bulletproofs uses a better and lighter *security assumption*, confronted with the zkSNARK's q-PKE, which is the *Discrete Logarithm Problem (DLP).* Security-wise, the scheme works with any elliptic curve with a reasonably large subgroup size, as the fastest elliptic curves supported are the ones constructed through the Ristretto (`www.ristretto.group`) protocol. Even though Bulletproofs work extremely well when implementing Range Proofs, they also offer compatibility to generate proofs from arithmetic circuits: the protocol uses its own format for constraints computation which, with the use of linear algebra, can be easily converted to R1CS and back.

As it was briefly introduced in the previous section, the usage of such protocol is mainly addressed to the Blockchain domain, and more specifically to the concept of *confidential transaction (CT)*, in which every transaction amount involved is hidden from public view using a commitment to the amount. The key element is that with the implementation of this approach, the public validation of the blockchain seems to be extremely protected; with the addition in every transaction of a Zero-Knowledge Proof of Validity, an observer can check that the sum of transaction inputs is greater than the sum of transaction outputs and that all transaction values are positive without revealing anything about the transaction inputs.

Protocol Structure

The Bulletproofs protocol is built on the Non-Interactive Zero-Knowledge Proof model, but it does not require a trusted Setup to regulate the communication between a Prover and a Verifier. The Prover and the Verifier have to agree on the computation to be executed, and it must be converted to an arithmetic circuit that operates on a certain prime field $\mathbb{F}p$. The arithmetic circuit, which has the same characteristics as the model introduced for zkSANRKs, has to contain multiplication gates, addition gates, and scalar multiplication. It can be then said that Bulletproofs protocol allows proving statements of the form

$$C(I) = O$$

where C is a circuit, I describes the input variables and O describes the output variables produced by the computation. Each variable involved can be declared either a public constant or a private variable (only known to the Prover).

Bulletproofs can use any group of prime order where the discrete logarithm problem is hard and, as was seen in the previous section, the fastest of such groups are the elliptic curves Ed25519[11], which can be implemented with the *Dalek* scheme which uses the Ristretto protocol, a compressed group of Ed25519 points. The Dalek scheme supports both the natural Bulletproofs format for arithmetic circuits and the R1CS format.

As it can be seen in a publication by Khovratovich [12], a generic Bulletproofs protocol, constructed from the Dalek scheme, involving a Prover P and a Verifier V, for arithmetic circuits can be described with the following steps.

1. P and V agree on an arithmetic circuit C. Th circuit has n multiplication gates and manages its local variables a and external private variables v; C is described as a set of three vectors of field elements a_L, a_R, a_O which satisfy n multiplication constraints of the form

   $$a_{Li} \cdot a_{Ri} = a_{Oi}$$

   As this does not completely describe the circuit, q additional affine constraints are introduced as

   $$L_j(a_L, a_R, a_O, v) = 0$$

   These constraints can be expressed as the following matrix equation.

   $$W_L a_L + W_R a_R + W_O a_O + W_V a_v = c$$

   These notions are used to guarantee that a set of R1CS constraints of the generic form $L_1(a) \cdot L_2(a) = L_3(a)$ can be always converted to this singular representation of constraints used from the Bulletproofs protocol. The previous concepts define a method to generate R1CS code with a specific format type that Bulletproofs requires.

2. V creates a set of generators by calling two functions, provided by the scheme, *PedersenGens* and *BulletproofGens* and provides them to P.

3. P initializes the proof procedure by initializing the Prover class with the generators above.

4. P commits to private variables v by calling the *commit* function. The commitments are added to Prover's transcript. The transcript is used to create challenges.

5. P runs the R1CS generation code to create constraints of the form $L_1(a) \cdot L_2(a) = L_3(a)$.

6. P calls prove to finalize the Proof and pass it to V.

7. V initializes the verification procedure by creating the Verifier object and then imports the commitments from P.
8. V runs the same R1CS generation code but he does not know the values.
9. V checks the Proof by calling the *verify* function.

Features

The complexity requirements of a Bulletproofs implementation depend on the elliptic curve group chosen and on the optimization process used to guarantee the compatibility of R1CS code with the protocol itself. With the optimization proposed in the original publication[10], the complexity for proving a statement about a circuit with n multiplication gates is described at the Prover level, the Verifer level, and the Proof level, as:

- **Prover**: makes 6 group multi-exponentiations of length $2n$, each taking $O(n/\log n)$ time using the Pippenger algorithm, and makes $O(n)$ scalar multiplications in $\mathbb{F}p$.
- **Proof size**: is $8 + 2\log n$ group elements and 5 scalars.
- **Verifier**: makes 1 group multi-exponentiation of length $2n$, taking $O(n/\log n)$ time, and also makes $O(n)$ scalar multiplications. Benchmarks demonstrate that the Verifier running time is roughly 1/20 of the Prover running time.

The previous features are grouped in Table 1.2.

Bulletproofs Features					
Scheme	**TS**	**Prove**	**Verify**	π	**Security Assumption**
Ristretto	no	$O(n)$	$O(n)$	$O(\log n)$	DLP

Table 1.2: Bulletproofs Features Table

1.3.3 Applications

The concept of Zero-Knowledge Proof has been approached in the precedent sections with a certain level of detail, but each and every protocol addressed has its own application fields that better suit it. Generally speaking, a group of application domains can be pointed out to better clarify and expand the overall view of this cryptographic approach and see where it is already looked at and implemented. Before listing any kind of application domain ZKP could be suited for, it is important to delineate the main practical goals when a system needs the implementation of one of such protocols: a system that seeks the implementation of a ZKP Protocol could be looking for the **the protection of the data (privacy, confidentiality) when a communication channel between two parties is required**. Nowadays there are various scenarios in which it is convenient to implement a certain type of ZKP Protocol, such as:

- **Blockchain**: The level of transparency that is offered by public Blockchains, such as Bitcoin or Ethereum, ensures that some type of transaction validation can be carried out in a transparent and public way. However, this can result in *loss of privacy* for the users involved in the validated transactions. Therefore, the use of ZKP Protocols allows a higher level of privacy to be established for users and data involved in public Blockchains. A practical example is the cryptocurrency *Zcash* (`www.z.cash`), which publishes transactions on a public Blockchain but uses zkSNARKs to provide greater protection for sensitive transactions and user data.

- **Finance**: The ING Banking Group has introduced [13] a ZKP model that is based on the concept of *range proof*, and it is called ZKRP: this allows any customer of the ING Bank to be able to prove that they are in possession of a committed secret number which is contained within a predefined range. A simple application example of this model might offer the possibility for a bank customer, who wants to take on a mortgage, to prove that their personal salary can support the payments, without revealing the exact figure of the salary itself to the bank.

- **Anonymous Verifiable Online Voting**: The concept of *voting* is a fundamental component to ensure democracy in an election process, whether a country or a stockholding company is involved. There could be the possibility that such a voting process is structured upon a decentralized and online system, and that could cause many issues, such as a loss of privacy in respect of the voters. It could also be very complicated to ensure that the verification process of the voting results is reliable. ZKP is a possible solution that ensures the implementation of a fair election ground that possesses the properties of anonymity and integrity in the voting process. Based on the structure of ZKP, eligible voters can cast a ballot with confidentiality guaranteeing their anonymity. It is also possible to allow voters to request, from the institution responsible for the voting process, a verifiable Proof to be sure that their vote is in the final and decisive ballot box.

- **Authentication**: ZKP can be used for the implementation of a user authentication process, which prevents the user itself to exchange secret and sensitive information, such as passwords. An example could be remote biometric authentication.

  Secure Biometric Authentication is a method that can be used to identify the login of a certain user through his or her biometric characteristics, such as *fingerprints, facial images, iris images, or vascular pattern recognition.* ZKP could be implemented to solve a security problem in privatization and confidentiality of data forwarded to an untrusted third party, to perform authentication.

- **Machine Learning**: ZKP can allow the owner of a certain Machine Learning algorithm to convince other users about the results obtained from such a model, without revealing any sensitive information about the model itself.

To close this section, and the first chapter of this Thesis, it can be said that, by briefly analyzing these solutions via ZKP for real problems, the main field of work of said cryptographic approach concerns the privatization and confidentiality of a certain sensitive secret that is involved in a communication between two or more entities.

Chapter 2

Zero-Knowledge Proof of Location Strategies

The second chapter of this Thesis aims to extend the concepts introduced in the *State of Art* to lay down the foundations of basic strategies to bind the ZKP technologies with the Proof of Location problem. Therefore a generic structure of a zkPoL strategy is presented. Starting from that structure, more specific ones will be defined based on the ZKP Protocol used. The chapter is structured as follows:

- **Zero-Knowledge Proof of Location [zkPoL] Strategy**: The basic strategy is presented, assessing the *Point in Polygon* problem, how it can be solved and how a proof can be computed based on that knowledge.
- **Interactive ZKP Strategy**: A zkPoL strategy based on the Interactive Zero-Knowledge Proof Model.
- **Non-Interactive ZKP Strategies**: Multiple zkPoL strategies based on the Non-Interactive Zero-Knowledge Proof Model.

2.1 zkPoL Strategy

As it was anticipated in *Chapter 1*, the Proof of Location problem concerns various key points, such as the reliability of the strategies used to obtain certain geographic location data, and the user privacy preservation of the geo-data involved in the exchange. The zkPoL strategy does not assess the first problem but primarily focuses on the second one. That is because the usage of a ZKP makes sure to put privacy and protection into the secret data, in this case, a geographic location, with the computation of a proof that demonstrates its knowledge, but does not ensure how reliable is the strategy used by the user to obtain such information.

The goal of this strategy is that, with the implementation of a ZKP Protocol, a user, who is in possession of its secret geographic location and a publicly known perimeter, **is able to prove that its personal geo-location lies into that perimeter**, without disclosing any sensitive information about it.

2.1.1 Authenticity of the Geographic Location

Although the implementation of ZKP in the strategy allows the user to calculate a Proof that does not lose any private information in the verification-proving process, it is equally important that the user is able to provide a valid and certified geographic location that reveals its position, then fail to fake the position and try to generate a malicious Proof.

The generation of this secret is therefore fundamental for the protocol itself, in order to certify the authenticity of the location data owned by the Prover, which cannot be shared due to the nature of the ZKP strategy. It is therefore necessary that the possible applications of the results of this Thesis foresee the adoption of systems and mechanisms that ensure these conditions, depending on the use case in which one works, which will be briefly discussed below. First of all, what we call location data (the secret possessed by the Prover), can be made of different forms (e.g., using geographic or Cartesian coordinates, including altitude or not, etc) and different complexities, in order to increase its authenticity and integrity. For example, tokens or identifiers (based on various IT techniques such as Hashing) of the Prover or of the system from which the location data comes can be part of the secret.

Regarding the localization systems, they are various and have different applicability according to the use cases. For example, in the case of outdoor localization, the most used localization systems are geographic satellite navigation systems (GNSS), which, however, in addition to being prone to various malicious attacks (such as spoofing [14]), do not give guarantees that the final data is not intentionally manipulated by the Prover as the GNSS receivers are almost passive systems. Therefore, to guarantee a certain degree of authenticity and reliability of the location data possessed by the Prover, it is necessary to use or integrate other solutions, such as the installation of reliable infrastructural elements with short-range communications that provide a reliable location data, also based on some IT techniques that guarantee integrity such as Digital Signature. The data exchange between the location receiver and the location system could be based on short-range protocols, such as Bluetooth, or by using Near Field Communication (NFC) technologies, which therefore guarantee the presence of the receiver in a specific area.

That said, the design and implementation of mechanisms to guarantee the authenticity of the geographical position provided by the Prover are beyond the scope of this Thesis and will not be further discussed. For simplicity, the location data possessed by the Prover is a simple pair of Cartesian coordinates and the Prover itself is considered trusted.

2.1.2 Structure

The data managed by the zkPoL strategy derives from the PoL problem and is presented as:

- **Geographic Location [Private]**: a secret pair of (latitude, longitude) coordinates should be used as the secret information that has to be provided to compute a proof. Based on the goals of the system, and other reasons that will be explained later in the chapter, such geo-data will be considered, and processed, as a **point-in-space** of *(x, y)* cartesian coordinates.

- **Perimeter [Public]**: a public list of points-in-space that will be addressed to as a **polygon**.

The actors involved are extracted from the ZKP Protocol implemented. Generally speaking, each zkPoL must have:

- **Prover**: knows a secret point-in-space and a public perimeter. Computes a Proof, which demonstrates that they lie into the perimeter, and sends it to a Verifier. Has to implement a strategy to compute a verifiable Proof.
- **Verifier**: knows a public perimeter. Receives a Proof from a Prover and validates it. Has to implement a strategy to verify the Proof received.
- **Setup [Optional]**: Its implementation is based on the solely needs of the ZKP used. In the case a Setup is necessary, it manages the communication between the Prover and the Verifier, by also making available additional computation and software resources.

Based on these premises, a general structure can be outlined, as can be seen in Figure 2.1.

The structure proposed looks and behaves very similar to the ones already introduced in Section 1.2.1, but in this specific case, in order to compute a Proof and carry on the ZKP Protocol established, the Prover must be able to *locally* ensure that their secret point-in-space lies into the publicly know perimeter/polygon. This requirement is known as the Point-In-Polygon (PIP) problem. With the computation of this information, and based on the protocol implemented, the Prover will be able to assess its behavior in respect of the Verifier involved. To better clarify this concept, a distinction between the I-ZKP and N-I ZKP models must be made:

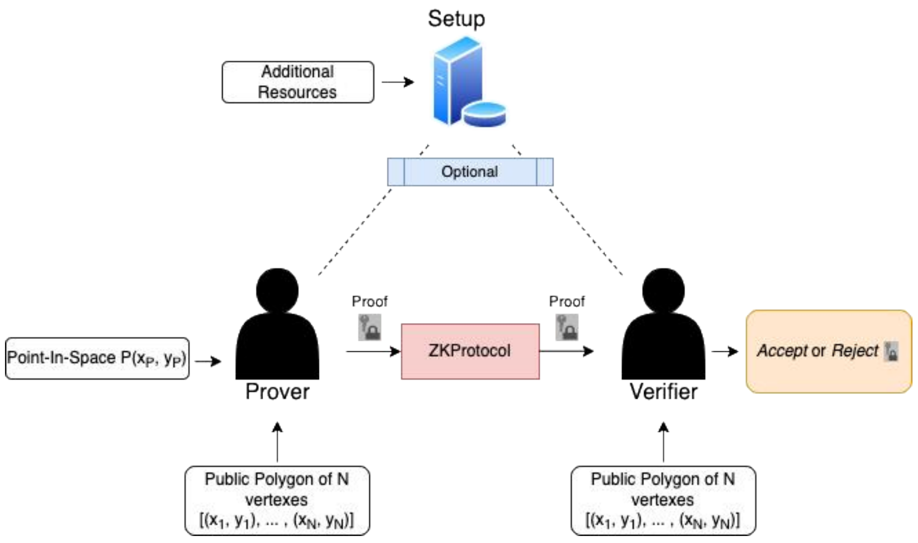
Setup
Additional Resources
Optional
Proof
Proof
ZKProtocol
Point-In-Space P(x_P, y_P)
Prover
Verifier
Accept or Reject
Public Polygon of N vertexes [(x_1, y_1), ... , (x_N, y_N)]
Public Polygon of N vertexes [(x_1, y_1), ... , (x_N, y_N)]

Figure 2.1: zero-knowledge Proof of Location Strategy Scheme

- *I-ZKP Prover*: based on the characteristics of the interactive approach, a Prover can choose to act as **trustworthy** or **malicious** towards a Verifier. That is possible because there is no other entity involved in the exchange that regulates the communication between the two actors; it is the Verifier who has to establish if the Prover is trying to fool them or not, on the basis of a property of reliability calculated with the responses from the challenges presented to the Prover. In relation to the zkPoL architecture, a Prover computes locally a verification of the Point-In-Polygon problem related to the data in their possession; based on the result they can either choose to establish a communication with malicious intent or not.

- *N-I ZKP Prover*: based on the characteristics of the non-interactive approach, the weak point established with the I-ZKP is not possible; that is because the Prover and the Verifier do not communicate directly with each other and, most of the times, there is a third-trusted party, known as Setup, that regulates their channel and does not allow the Prover to compute a Proof without knowing a valid secret geographic location.

To be able to deal with either the situation could be, the next section introduces the Point-In-Polygon problem and an algorithm that solves it, which must be implemented at the Prover level.

2.1.3 Point-In-Polygon

A *Point-In-Polygon (PIP)* problem [15] asks whether a given point-in-space p lies inside, outside, or on the boundary of a polygon; in our case, the point lying on the border and vertexes are considered to be inside. That is the exact situation in which a Prover that participates in a zkPoL strategy, finds itself: a proof, which demonstrates the knowledge of a geographic location (point-in-space) lying inside a perimeter (polygon), has to be computed. That is also the reason why, for the purposes of this Thesis, it is more useful to work and concentrate the algorithms and technologies not directly on the PoL fields, as the geographic location and perimeter are, but rather focus on the structural basis of the strategy and implement a working solution for the PIP problem. Then, at a later stage, the solutions can be easily adapted to the PoL fields.

This problem has to be solved at the Prover level of the zkPoL strategy; based on the ZKP Protocol chosen, an algorithm to solve the PIP problem must be implemented through the required technologies involved. For example, if a zkSNARK Proof is used in the zkPoL strategy, the PIP problem has to be assessed through the scripting of a custom arithmetic circuit that defines a solution for it.

To find a solution to the PIP problem, two common approaches can be implemented: the **winding number algorithm** and the **ray casting algorithm**. The latter is the one that was chosen to be used in the proposed zkPoL strategy.

Winding Number Algorithm

This algorithm uses the concept of a *winding number* to check if a point lies inside a polygon. In mathematics, a winding number represents the total number of times a closed curve in the plane travels counterclockwise around a fixed point; the winding number also depends on the *orientation* of the curve, and it is negative if the curve travels around the point clockwise. To check if a point lies inside a polygon, there is the need to compute the given point's winding number with respect to the polygon. The algorithm states that for any point inside the polygon, the winding number would be **non-zero**: a winding number of 0 means the point is outside the polygon, and other values indicate the point is inside.

Ray Casting Algorithm

The Ray Casting algorithm is the most common and efficient solution for the PIP problem. The basics of this algorithm state that, in order to check if a point lies inside or outside of a polygon, a checking on collisions between a *ray* and a polygon must be held. In this case, the ray is defined as a horizontal line segment that extends from the point checked to infinity, following a predetermined and fixed direction. If the point is on the **outside** of the polygon the ray will intersect its edge an **even** number of times; if the point is **inside** then it will intersect the edge an **odd** number of times. A detailed Ray Casting algorithm has also to implement a check on whether a point lies on a *segment* and/or on a *vertex* of the polygon.

The algorithm is based on a simple observation [15] that if a point moves along a ray from infinity to the probe point and if it crosses the boundary of a polygon, possibly several times, then it alternately goes from the outside to inside, then from the inside to the outside, etc. As a result, after every two "border crossings" the moving point goes outside.

The generic implementation of the Ray Casting algorithm follows the execution of three steps, on the basis of the knowledge of two fundamentals parameters: a *point* $P(x_P, y_P)$ and a *polygon* $[(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$ of n vertexes. The steps are:

1. Draw a ray that extends from the point to be checked to infinity, following a fixed direction.

2. Count the number of times the line intersects with the polygon edges.

3. A point is:

    - **Inside**: if the count of intersections is *odd* or if the polygon lies on either an edge or a vertex of the polygon.

    - **Outside**: if the count is *even* and none of the other conditions checks as true.

The Ray Casting algorithm is the solution adopted to approach the PIP problem in the zkPoL strategies. Algorithm 1 shows a detailed implementation of a Ray Casting algorithm for the PIP problem.

The algorithm successfully implements the steps previously introduced but adds a new concept: the property of *collinearity*. When a set of points lie on the same straight line they are said to be collinear. In relation to the algorithm, the set of points assessed is a triplet of points, which includes the point to be checked and the pair of points with which a segment of the polygon is defined. The property of collinearity is related to the concept of *orientation*, which is used to let the algorithm know the actual orientation of a triplet of points, and a mechanism to check when the number of intersections between the ray and a segment of the polygon is needed. To find the orientation of a generic triplet of points, a statement has to be computed. Given a set of three points A, B, and C, the orientation of the ordered triplet are found with the formula:

$$orientation = [(y_B - y_A) \cdot (x_C - x_B)] - [(x_B - x_A) \cdot (y_C - y_B)]$$

The value stored in the result represents the orientation of the ordered triplet of points.

The possible results are:

- **Clockwise**, when $orientation > 0$
- **Counter-clockwise**, when $orientation < 0$
- **Collinear**, when $orientation == 0$

Based on the results of this important addition to the Ray Casting algorithm, to find out if an intersection between the ray and the segment has occurred, a general case of intersection and four special cases can be tested, meaning that more complex polygons can be tested with the algorithm. Generally speaking, when checking the intersection of the ray with a segment, four points are tested: P1 and Q1, which are extracted from the segment (P1, Q1); P2 and Q2, which are extracted from the ray (P2, Q2). Four different orientations of triplets are computed:

- *o1*, as the orientation of the triplet (P1, Q1, P2).
- *o2*, as the orientation of the triplet (P1, Q1, Q2).
- *o3*, as the orientation of the triplet (P2, Q2, P1).
- *o4*, as the orientation of the triplet (P2, Q2, Q1).

The cases of intersection can be then tested.

- **General Case**: if $(o1 \neq o2) \wedge (o3 \neq o4)$ is true, there is an intersection.
- **Special Cases**:
  1. if o1 is *collinear* (o1 = 0) and P2 lies on the segment (P1, Q1), there is an intersection.
  2. if o2 is *collinear* (o2 = 0) and Q2 lies on the segment (P1, Q1), there is an intersection.

3. if o3 is *collinear* (o3 = 0) and P1 lies on the segment (P2, Q2), there is an intersection.
4. if o4 is *collinear* (o4 = 0) and Q2 lies on the segment (P2, Q2), there is an intersection.

This implementation of the Ray Casting algorithm is useful because it makes it possible to test the PIP problem with convex, concave, and more complex polygons. In Algorithm 1 the steps and basic logic of the algorithm are presented.

Algorithm 1: Ray Casting Algorithm

```
Data: point($x_P, y_P$), polygon[$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$], $n \geq 3$
Result: True if point is inside polygon, False if point is outside.
$infinity \leftarrow (value, point.yp)$ ;   /* value is an approximation */
$ray \leftarrow (point, infinity)$ ;        /* of infinity on the x axis */
$collisions \leftarrow 0$;
for each segment of polygon do
    if ray intersects with current segment then
        if point is collinear with current segment then
            if point lies on current segment or point lies on a vertex of
              current segment then
                return True
        $collisions \leftarrow collisions + 1$;
if (collisions % 2) == 1 then
    return True ;                           /* odd */
else
    return False ;                          /* even */
```

2.2 Interactive zkPoL Strategy

This section introduces the first strategy proposed to implement a zkPoL, which is based on an *Interactive ZKP Protocol*. As introduced in the previous section of this chapter, the objective of a zkPoL strategy is to compute a Proof, which describes the knowledge of a private point-in-space, as the personal geo-location of the Prover could be, that lies inside a publicly known perimeter, without revealing it.

2.2.1 Structure

This zkPoL strategy is based on an Interactive ZKP protocol known as *Discrete Logarithm Proving* [4] (DLP) algorithm, which is defined in more detail in the following section, so it is fundamental to delineate the parameters and technologies that each entity of the protocol has to implement to be able to participate in the verification process. Therefore, the entities involved are a Prover and a Verifier, and they are going to interact with each other through the rules defined in the DLP algorithm. Figure 2.2 shows the architecture of the Interactive zkPoL strategy presented in this section.

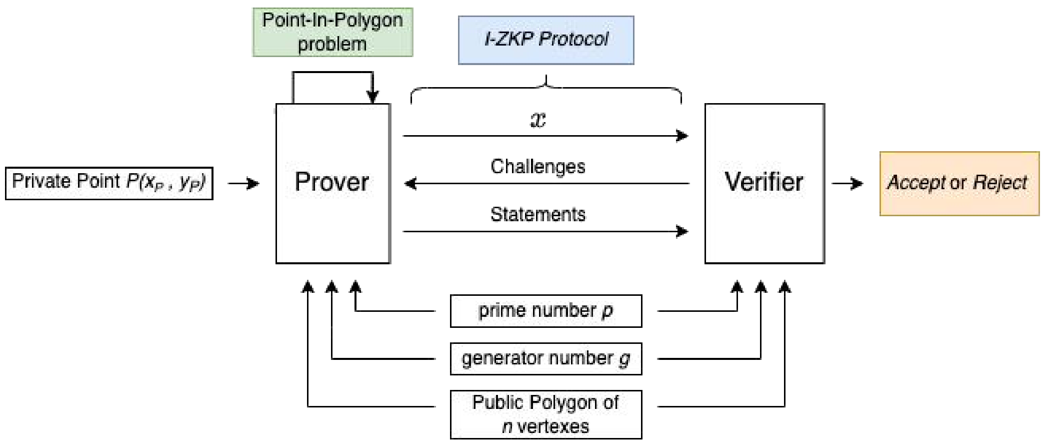

Figure 2.2: Interactive zkPoL architecture

- **Prover**

  (a) Parameters: a private point $P(x_P, y_P)$, a public polygon of n vertexes, a shared prime p and a generator $g \in \mathbb{Z}p$.

  (b) Technologies: implementation of the Ray Casting algorithm presented in Section 2.1.1, and the implementation of the DPL algorithm to compute a Proof and interact with a Verifier.

  (c) Proof: a secret value x generated from the correct validation of the PIP problem.

- **Verifier**

  (a) Parameters: a shared prime p and a generator value $g \in \mathbb{Z}p$.

  (b) Technologies: implementation of the DPL algorithm to verify a Proof and send challenges to the Prover.

A brief introduction of the Proof exchanged in the communication has been presented, but the mechanism that generates it is a custom one and has to be addressed individually. The DLP algorithm described later in Section 2.2.2, says that the Proof has to be a generated private value x.

Proof x : the objective is to compute a value on the basis that its correct computation is possible only with the knowledge of valid parameters to solve the PIP problem and to participate in the DLP algorithm, which are respectively:

- PIP: two values are extracted from the correct computation of the PIP problem; the first value is the **perimeter** of the public polygon, and the second value is the **sum of distances** from the private point to the edges of the public polygon.
- DLP algorithm: the *shared prime* p and the shared *generator* g.

These values are then summed togheter to form the secret value x, which describes the Proof generated by the Prover for this specific zkPoL strategy.

$$x = (\text{sum of distances}) + (\text{perimeter}) + p + g$$

The peculiar thing about this Proof computation process, is the fact that each parameter summed has to be known correct to compute a valid x. It attests that the Prover has solved the PIP correctly and will participate in the protocol with the correct setup values.

This interactive approach to the zkPoL strategy, which is carried on by a Prover and a Verifier, can be summed up with the following steps:

1. Prover knows a private point P and wants to prove to Verifier that P lies in a public polygon of n verteces.
2. Prover and Verifier agree on a prime p, a generator g, and the public polygon.
3. Prover solves the PIP problem. If P actually lies into the perimeter, then the Prover will generate a valid Proof. Otherwise, if P does not lie into the perimeter, can decide to generate a false Proof and try to fool the Verifier to gain permission. Prover passes the Proof to Verifier.
4. The DLP algorithm is executed to guarantee interaction between Prover and Verifier.
5. Based on the results from the DLP algorithm, Verifier decides to **accept** or **refuse** the Proof.

2.2.2 Discrete Logarithm Proving Algorithm

The DLP algorithm is used to implement an I-ZKP model, and the basic concept behind is that is used by a Prover to prove to a Verifier that they know the discrete logarithm of a given value in a given range group. For example, given a value y, a large prime p, and a generator g, the Prover wants to prove that they know a secret value x such that $g^x \pmod p = y$, without revealing x. The knowledge of the secret value x could be used also as a *Proof of Identity*, because the Prover could have such knowledge by choosing a random value that did not reveal to anyone, but could distribute the generated Proof to a group of potential Verifiers, such that at a later time, proving the knowledge of x is equivalent to proving identity as the Prover itself. With the implementation of this algorithm into the Interactive zkPoL strategy, the goal is to prove a *secret value* x, which can be computed correctly only with the knowledge of a valid solution for the PIP problem proposed, and with the correct knowledge of the other public parameters required to both parties by the protocol.

The algorithm involves a Prover P and a Verifier V, as P knows a secret value ρ and wants to prove its knowledge to V, without revealing it. The algorithm is implemented with the following three steps.

1. P and V agree on a prime p and a generator g, as g belongs to the multiplicative field $\mathbb{Z}p$.

2. (Proof) P computes the value

$$y = g^{\rho} \pmod p$$

   and transfers the value to V.

3. The following two steps are repeated a (large) number of times, and represent the interactive property of an I-ZKP model.

   (A) (Statement) P repeatedly picks a random value $r \in \mathbb{U}[0, p-2]$ and computes $C = g^r \pmod p$. Then, P transfers the value C to V.

   (B) V poses one of two challenges to P:

Challenge 1 : P has to compute the value $(\rho + r) \pmod{p-1}$ and transfer it to V.
V will then verify for $(C \cdot y) \pmod p \equiv g^{(\rho+r) \pmod{p-1}} \pmod p$.

Challenge 2 : P passes to V the value of r. V will then verify for
$C \equiv g^r \pmod{p}$.

The value $(\rho + r) \pmod{p-1}$ can be seen as the encrypted value of $x \pmod{p-1}$. If r is truly random and equally distributed in $[0, p-2]$, this approach does not leak any other information of the secret ρ.

2.3 Non-Interactive zkPoL Strategies

This section introduces different approaches to the zkPoL strategy that focuses on Non-Interactive ZKP Protocols. These represent a step forward from the Interactive zkPoL outlined in Section 2.2, removing the mandatory interaction between Prover and Verifier, in order to ensure more separation between the two. The proving mechanism will be executed with the exchange of a single message. These approaches also make it possible to extend such a strategy through the involvement of decentralized systems, where Blockchains can act as decentralized always-on Verifiers. The protocols that are involved are zkSNARK and Bulletproofs; due to their non-interactive nature, both require the implementation of similar technologies but need different structures.

2.3.1 zkSNARK-Based zkPoL Strategy

zkSNARK brings great benefices to the zkPoL strategy, making it possible to separate the responsibilities of the Prover and the Verifier and requires the implementation of a Setup process that functions as a regulator of the communication channel. Moreover, the characteristics of the zkSNARK protocol bring to the zkPoL strategy more security due to the usage of elliptic curves, as it was explained in Section 1.3.1. The protocol, being structured on the q-PKE assumption, requires the implementation of new parameters and technologies to generate and verify a zkSNARK Proof.

Structure

This approach extends the concepts introduced with the Interactive zkPoL strategy, with the addition of the requirements needed to implement zk-SNARKs. Figure 2.3 outlines the architecture of a generic zkSNARK zkPoL strategy.

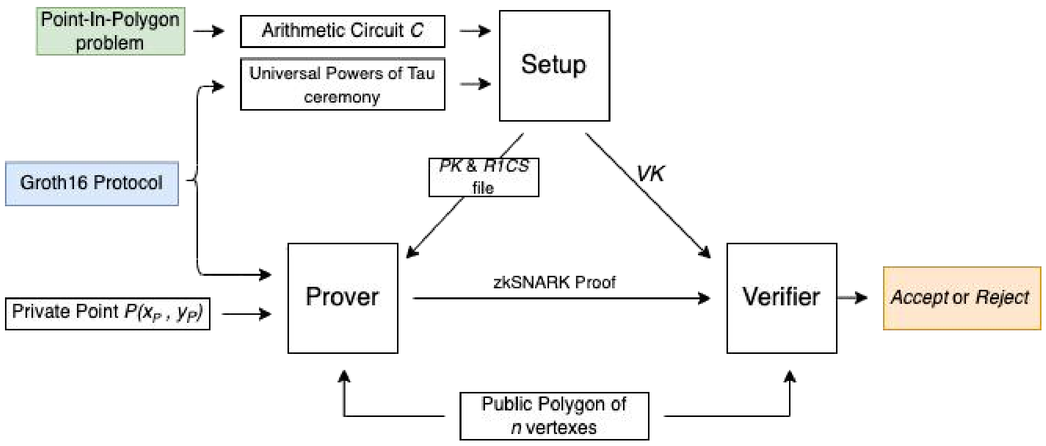

Figure 2.3: zkSNARK zkPoL architecture

The entire architecture is based on the zkSNARK Protocol. To construct a zkSNARK Proof, it is required that the architecture uses a ZKP construction scheme, and in this case, the scheme is the Groth16 protocol. This scheme is used to establish basic parameters for the computations that the entities at stake will have to perform. As can be seen in Figure 2.4, the strategy is a lot more complex and sees new concepts and technologies. The main changes are brought from the non-interactive nature of the protocol, with the addition of a key pair (PK, VK), and from the q-PKE security assumption, which requires the implementation of an arithmetic circuit that describes the computation problem assessed by the scheme. In this case, the Point-In-Polygon problem is going to be solved with the implementation of the Ray Casting algorithm through an arithmetic circuit. What stands out is a new technology that the Setup process needs, a *Powers of Tau ceremony*, established with the Groth16 protocol. This is a special cryptographic ceremony used to maintain security during the generation of the pair of keys, due to the fact that said process generates a piece of dangerous data, i.e., *toxic waste* that is taken care of and discarded by the ceremony.

Before discussing in more detail the technologies used by this strategy, it is important to define the behavior of each entity of the scheme.

- **Setup**: sets up the zkSNARK Protocol and the basic parameters needed to approach the zkPoL strategy.
  - (a) Parameters: an arithmetic circuit C, a contribution to a *Groth16 Powers of Tau (PoT) ceremony.*
  - (b) Technologies: the circuit has to implement an algorithm that solves the PIP problem. The PoT ceremony is used to maintain security during the generation of the key pair (PK, VK), as it generates dangerous data. The Setup also has to implement a process that takes care of the q-PKE security assumption, to generate an R1CS file from the circuit C.
  - (c) Outputs: generates a pair of keys (PK, VK), which are respectively used from a Prover and a Verifier to prove and verify a zkSNARK Proof. The R1CS file generated from the circuit is sent to the Prover to compute a valid secret from private and public inputs.
- **Prover**: it is responsible to generate a zkSNARK Proof.
  - (a) Parameters: a private point P, a public polygon, a Proving Key (PK), and an R1CS file.
  - (b) Technologies: it has to involve the biggest amount of computation resources, as can be seen in Table 1.1, due to the implementation of the process that generates a valid zkSNARK Proof from the knowledge of the parameters highlighted. To generate a Proof, a valid secret is required; the secret is generated from the valid

compilation of the circuit C through the R1CS file, with a valid set of private and public inputs that solve the PIP problem.

    (c) Outputs: a zkSNARK Proof.

- **Verifier**: it verifies a zkSNARK Proof.
    (a) Parameters: a public polygon, a Verification Key (VK), and a zkSNARK Proof.
    (b) Technologies: it receives a VK, associated to the PK used by Prover, to verify a zkSNARK Proof. The verification process also requires the public inputs used to compute the Proof.
    (c) Outputs: **accepts** or **rejects** the zkSNARK Proof.

Due to the involvement of a PoT ceremony, the strategy requires the implementation of a phase that sets up such a ceremony. This phase is necessary to establish a perpetual ceremony that generates basic parameters and can be used by multiple zkSNARK projects. The strategy is then split into two phases.

- **Phase 1 (Parameters generation)** [**Optional**]: establishes a perpetual Powers of Tau ceremony, that can be then used from different zkSNARK projects to guarantee secure parameters for the computation of the processes involved in the protocol.
- **Phase 2 (Circuit-specific)**: represents the actual zkPoL strategy, which accounts for an arithmetic circuit that implements a solution for the PIP problem, and lets Prover and Verifier proceed with the verification process.

It is important to say that Phase 1 is optional due to its perpetual property; if a PoT ceremony has already been established, the zkPoL strategy can then contribute to it without the need to generate from scratch a new ceremony. Otherwise, if no ceremonies are available, Phase 1 has to be executed.

More generally, this non-interactive zkSNARK approach to the zkPoL strategy can be summed up with the following steps:

- **Phase 1**

  1. Setup contributes to an already existing Groth16 PoT ceremony; if no PoTs are available, a new one has to be run.

- **Phase 2**

  1. Setup generates an R1CS file from an arithmetic circuit C that solves the PIP problem, and also produces a unique pair of keys (PK, VK).
  2. Prover that enters the protocol receives the key PK and the R1CS file from the Setup.
  3. Prover generates a valid secret with the compilation of the R1CS file from valid public/private input data.
  4. Prover generates a zkSNARK Proof π.
  5. Verifier that enters the protocol wants to verify π, receives the key VK from the Setup.
  6. Verifier verifies for π, and decides whether to **accept** or **reject** it.

Technologies

This zkPoL strategy differs from the interactive one through the requirement of new technologies. The main addition is the involvement of a Groth16 PoT ceremony, that has its specific purpose.

- **Groth16 Powers of Tau ceremony**: as it was introduced in the structure of this approach, this is a *special cryptographic ceremony* in which there are multiple participants taking turns to perform a computation. This perpetual ceremony is used to discard dangerous data produced from the computation process of the key pair (PK, VK) by the Setup. The final result of all the computations can be trusted as long as just one participant ensures that they securely discard their toxic waste. Earlier in this section, it has been said that the approach uses a Groth16 PoT ceremony, that is because the PoT also sets up the elliptic curve that is going to be used to construct a zkSNARK Proof, and the Groth16 Protocol, the default construction schema, defines a group of elliptic curves that can be used for zkSNARKs.

Another important addition to this approach is the possibility to involve Blockchains as a decentralized and always-on Verifier. That is because, as it can be seen in the paper by Reitwießner [16], zkSNARKs are very relevant to Blockchains, especially to Ethereum. With zkSNARKs, it becomes possible to not only perform secret arbitrary computations that are verifiable by anyone but also to do this efficiently. The way this concept is implemented is by exporting the Verifier process inside a precompiled Smart Contract by publishing it through the Ethereum Virtual Machine (EVM).

- **Smart Contract Verifier**: a Smart Contract that implements the Verifier process is published on-chain, and is related to the arithmetic circuit used in the strategy. This forces all Ethereum clients to implement a certain pairing function and multiplication on a certain elliptic curve as the precompiled contract is published. The benefit is that this is probably much easier and faster to achieve but, on the other hand, the drawback is that we are fixed on a certain pairing function, that generates (PK, VK), and a certain elliptic curve. Any new client for Ethereum would have to re-implement these precompiled contracts. Furthermore, if there are advancements related to zkSNARKs, as better pairing functions or better elliptic curves can be, we would have to publish on-chain new precompiled contracts.

2.3.2 Bulletproofs-Based zkPoL Strategies

Bulletproofs is a further improvement from zkSNARKs as it does not require a trusted Setup to regulate the communication channel and it is based on a lighter security assumption, the Discrete Logarithm Problem, that leads to lighter computation complexity required from the processes involved in the protocol. All these characteristics suit possible approaches that can implement a zkPoL strategy. Even though the Bulletproofs is still a new and emerging ZKP Protocol, this section aims to present two possible approaches for the zkPoL strategy. The two distinct approaches are structured on two separate methods that can be implemented via Bulletproofs. The first one uses an Arithmetic circuit to compute a Proof, and the second one uses the concept of Aggregated Range Proofs. Both strategies only differ in the mechanism implemented to compute the Proof, but they follow the same structural basis. Figure 2.4 outlines a basic Bulletproofs zkPoL architecture.

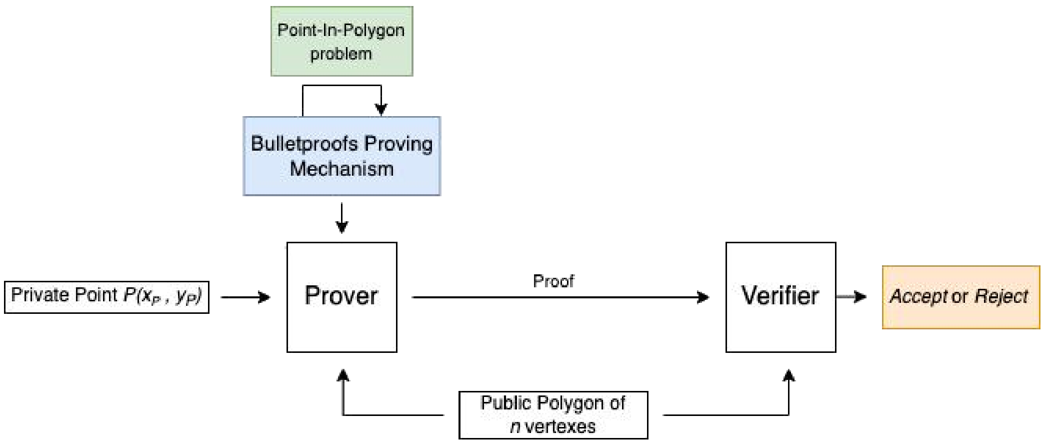
Point-In-Polygon problem
Bulletproofs Proving Mechanism
Private Point $P(x_P, y_P)$
Prover
Proof
Verifier
Accept or Reject
Public Polygon of n vertexes

Figure 2.4: Bulletproofs zkPoL architecture

Bulletproofs zkPoL with Arithmetic circuits

This strategy uses an Arithmetic circuit to implement and solve the PIP problem, in order to guarantee that the secret geo-location of a Prover lies inside a public perimeter. It follows the protocol established in Section 1.3.2

and, as it uses arithmetic circuits, can be directly compared to the zkSNARK zkPoL strategy. In this case, the Bulletproofs protocol makes the strategy more *lightweight*, as the computation complexity required for the Prover process is much faster as it is of linear growth $O(n)$, where n is the number of gates that define the arithmetic circuit. Moreover, the protocol does not need a Setup process, the arithmetic circuit is assessed at the Prover level, and the R1CS file generated will be used by both the Prover and the Verifier to compute and verify a Proof. A downside to this architecture is that at the current state, there are no supported libraries that implement Bulletproofs with complex arithmetic circuits. Moreover, the Bulletproofs protocol requires the translation of the R1CS representation of the arithmetic circuit used into a constraint format type custom to the Bulletproofs protocol (more details can be found in Section 1.3.2). Figure 2.5 outlines a Bulletproofs zkPoL architecture with Arithmetic circuits.

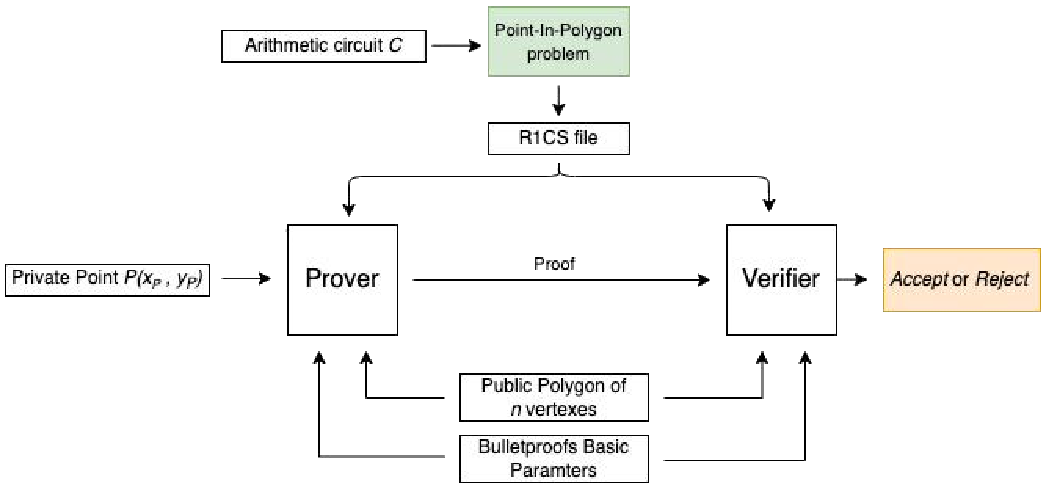

Figure 2.5: Bulletproofs zkPoL architecture with Arithmetic circuits

Bulletproofs zkPoL with Aggregated Range Proofs

The second strategy proposed with the Bulletproofs protocol involves the computation of an aggregated Range Proof that solves the PIP problem the zkPoL strategy introduces. This approach differs from the first one in the mechanism used to generate a valid Proof. The structure remains the same as the generic Bulletproofs one, but it uses the concept of Range Proofs. However, the usage of Range Proofs requires a Setup process to establish a commitment scheme that produces basic parameters that are grouped in a relation R and will be used from Prover and Verifier. Figure 2.6 outlines a Bulletproofs-based zkPoL architecture with aggregated Range Proofs.

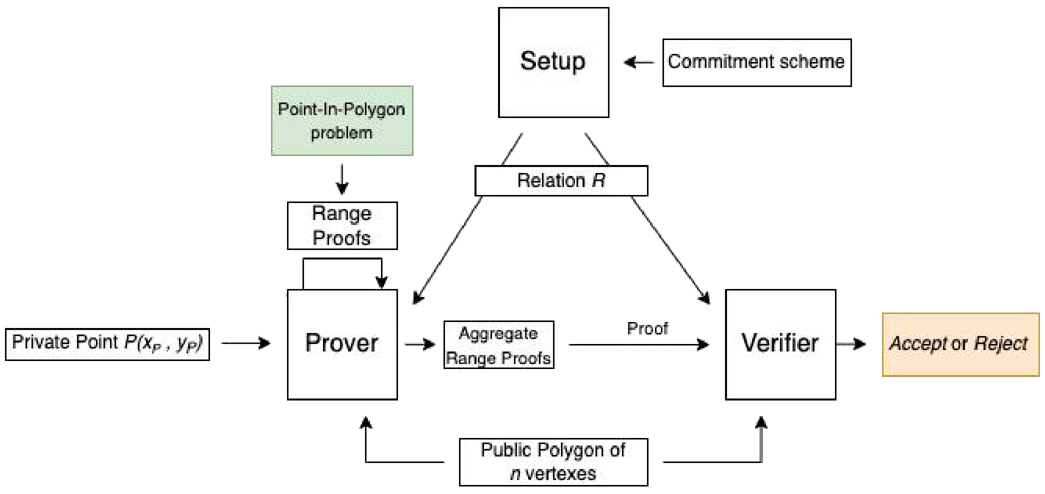

Figure 2.6: Bulletproofs zkPoL architecture with Aggregated Range Proofs

Therefore, it is important to introduce the basic concept that defines a Range Proof.

- **Range Proof**: A Range Proof [10] proves that a secret value, which has been encrypted or committed to, lies in a certain interval. Range proofs do not leak any information about the secret value, other than the fact that they lie in the interval. The mechanism used to generate a Proof will be handled by the concept of *Arbitrary Range Proofs* for intervals of the form [a, b], introduced in a publication by Camenisch et al.[17].

- **Aggregated PIP Range Proof Mechanism**: To solve the PIP problem and generate a valid Proof that suits the Bulletproofs protocol, an aggregated Range Proof has to be computed; this means that multiple Range Proofs must be beforehand generated. The mechanism is described as follows: the extreme edges of the public polygon are used to define two distinct Range Proofs, one for the x-axis and one for the y-axis. Then, the secret point P coordinates are individually tested to be inside the corripstecive axis-Range-Proof. Ideally, the two Range Proofs define a projection of a regular polygon of 4 vertexes. If the coordinates of the secret point correctly lie into their given interval, the two ranges are shrunk and a new couple of Range Proofs are generated. The goal is to reduce the two predefined Range Proofs until the projected 4-edged polygon lies inside the public polygon. Only at this moment, the actual pair of Range Proofs are used to prove that the secret point lies in the polygon. But to be able to generate a valid Proof, the Verifier has to receive all the Range Proofs created correctly; in order not to clog the communication channel, the Bulletproofs protocol provides the mechanism of Multi-Range Proofs, which aggregates multiple Range Proofs into a single compact Proof [17].

Algorithm 2 outlines with more detail and order, the steps introduced needed to implement such a mechanism. Figure 2.7 shows a visual example of how this mechanism works.

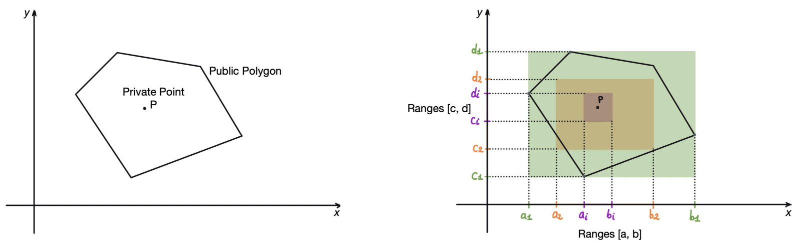

Figure 2.7: Basic parameters and Example for Aggregated PIP Range Proof Mechanism

Algorithm 2: Aggregated PIP Range Proof Mechanism

```
Data: point($x_P, y_P$), polygon[$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$], $n \geq 3$
Result: An Aggregated PIP Range Proof.
valueX1 ← the $x$ coordinate where polygon[$i$].$x$ is the least valuable;
valueX2 ← the $x$ coordinate where polygon[$i$].$x$ is the most valuable;
valueY1 ← the $y$ coordinate where polygon[$i$].$y$ is the least valuable;
valueY2 ← the $y$ coordinate where polygon[$i$].$y$ is the most valuable;
interval1 ←(valueX1, valueX2);
if point.x lies in interval1 and point.y lies in interval2 then
    RangeProof1 ← interval1;
    RangeProof1.commit ; /* commit Range Proof to ceremony */
    RangeProof2 ← interval1;
    RangeProof2.commit;
else
    exit;
stop ← False;
while stop ≠ True do
    interval1 ← restriction of interval1;
    interval2 ← restriction of interval2;
    if point.x lies in interval1 and point.y lies in interval2 then
        RangeProof1 ← interval1;
        RangeProof1.commit;
        RangeProof2 ← interval1;
        RangeProof2.commit;
    else
        exit;
    if the projection of the intervals lies inside polygon then
        stop ← True;
Proof ← Multi-RangeProof.generateProof(commitments of
  RangeProofs);
```

Chapter 3

Implementation of Zero-Knowledge Proof of Location Strategies

The third Chapter of this Thesis proposes an implementation of the Zero-Knowledge Proof of Location strategies introduced in Chapter 2. Each implementation follows one of the zkPoL architectures, using specific frameworks that better suit the technologies of the relative strategy. This chapter defines the implementation of two different strategies: an Interactive zkPoL and a Non-Interactive zkPoL. The chapter is structured as follows:

- **pyZKP**: An Interactive zkPoL Library written with Python3.
- **zkSNARK DApp**: A Non-Interactive zkPoL Application, based on the zkSNARK zkPoL architecture defined in Section 2.3.1.

3.1 pyZKP

pyZKP is a custom Python library that implements the Interactive zkPoL strategy delineated in Section 2.2. The library locally implements every aspect and technology required by the architecture, simulating the communication between two nodes. Therefore, the entities involved in the strategy, the technologies used to solve the PIP problem, and the I-ZKP Protocol, will all be scripted using the Python programming language.

3.1.1 Framework

The objective of the pyZKP library is to briefly demonstrate the potential of an I-ZKP model applied to the PoL problem, as the interactive approach is not as efficient as the non-interactive one. Due to this goal, the choice of the framework used to implement such a strategy has fallen upon the **Python** programming language, using the later release of Python (version 3.9.0).

Python[1], is a high-level, interpreted, general-purpose programming language. An important characteristic is that Python is dynamically-typed and garbage-collected, which lets the implementation of the strategy be direct and clear. It supports multiple programming paradigms, including procedural/object-oriented and functional programming. This choice makes it possible to guarantee a very simple and immediate implementation of the Ray Casting and the DLP algorithms.

3.1.2 Library Structure

pyZKP is organized with the following library structure.

```
pyZKP
├── raycasting
├── entities
└── zkp
```

Each file has its own task, which are described as follows:

- **raycasting.py**: Implementation of the Ray Casting algorithm described in Section 2.1.3.

[1]Python - www.python.org

- **entities.py**: Definition of classes for the entities and data structures involved in the I-ZKP Protocol.
- **zkp.py**: Implementation of the logic of the Interactive zkPoL strategy described in Section 2.2.

3.1.3 Ray Casting with Python3

The first thing to analyze is the implementation of the Ray Casting algorithm. The structure of the algorithm follows the steps delineated in Algorithm 1. It is implemented in the file **raycasting.py** of the library, which contains a list of functions that are necessary to define the logic of the Ray Casting algorithm. The functions implemented in the file are defined as follows:

1. Function **is_inside_polygon(point, polygon) → bool**: to verify that a given point-in-space lies in a polygon. The implementation is outlined in the Code Listing 3.1.
2. Function **doIntersect(segment1, segment2) → bool**: to test if an intersection happens between two line segments. The implementation is outlined in the Code Listing 3.2.
3. Function **orientation(point1, point2, point3) → int**: to extract the orientation of an ordered triplet of points-in-space. The implementation is outlined in the Code Listing 3.3.
4. Function **onSegment(point1, point2, point3) → bool**: to test if point2 lies on the segment point1-point3. The implementation is outlined in the Code Listing 3.4.

Function 1 is the main function, as it implements the actual Algorithm 1 for the Ray Casting. The other three functions are used to implement certain behaviors needed from the *is_inside_polygon()* function.

```python
def is_inside_polygon(points: list, p: tuple) -> bool:
    n = len(points)
    # Polygon must have at least 3 vertices
    if n < 3:
        return False
    # Create a horizontal ray
    ray = (INFINITE, p[1])
    count = i = 0
    while True:
        next = (i + 1) % n
        # Check if ray intersects with current segment
        if (doIntersect(points[i], points[next], p, ray)):
            # If p is collinear with current segment
            # check if p lies on it.
            if orientation(points[i], p, points[next]) == 0:
                return onSegment(points[i], p, points[next])
            count += 1
        i = next
        if (i == 0):
            break
    # Return 1 if count is odd, 0 otherwise
    return (count % 2 == 1)
```

Code Listing 3.1: Function 1 - checks if a point lies in a polygon

```
def doIntersect(p1, q1, p2, q2) -> bool:
    # Find the four orientations needed for
    # general and special cases
    o1 = orientation(p1, q1, p2)
    ...
    o4 = orientation(p2, q2, q1)
    # General case
    if (o1 != o2) and (o3 != o4):
        return True
    # Special Cases
    if (o1 == 0) and (onSegment(p1, p2, q1)):
        return True
    ... other 3 special cases ...
    return False
```

Code Listing 3.2: Function 2 - checks if an intersection happens between two line segments

```
def orientation(p: tuple, q: tuple, r: tuple) -> int:
    val = (((q[1] - p[1]) * (r[0] - q[0])) - ((q[0] - p[0]) *
    (r[1] - q[1])))
    if val == 0: return 0 # Collinear
    if val > 0:  return 1 # Clock
    else:  return 2 # Counterclock
```

Code Listing 3.3: Function 3 - extract the orientation of an ordered triplet of points-in-space

```
def onSegment(p: tuple, q: tuple, r: tuple) -> bool:
    if((q[0] <= max(p[0], r[0])) &
       (q[0] >= min(p[0], r[0])) &
       (q[1] <= max(p[1], r[1])) &
       (q[1] >= min(p[1], r[1]))):
       return True
    return False
```

Code Listing 3.4: Function 4 - to test if q lies on the segment p-r

3.1.4 Interactive ZKP Protocol with Python3

The Interactive ZKP Protocol used from the zkPoL strategy is the Discrete Logarithm Proving algorithm, which is explained in detail in Section 2.2.2. To implement the protocol two files are used.

entities.py : Contains the declaration of the classes that define the entities of the protocol, the Prover, and the Verifier. It also defines a class that implements a simple Polygon structure. The entities will implement the mechanisms required from the DLP algorithm.

The Polygon class defines an ordered list of n points. It is constructed from an empty list which can be then filled with custom points until the list reaches n vertices. Code Listing 3.5 outlines the implementation of the Polygon class.

The Entity class defines a basic entity that is involved in the protocol. Code Listing 3.6 shows the implementation of the Entity class.

The Prover and Verifier classes are subclass of the Entity class. That is because they hereditate its parameters, getters, and setters, but have to implement personal parameters and methods that are required to participate in the I-ZKP Protocol. Code Listing 3.7 and 3.8 show the implementation of the Prover and Verifier classes.

```python
class Polygon:
    def __init__(self, nVertices):
        self.nVertices = nVertces
        self._points = []
    ... __str__ ...
    ... Getters ...
    ... Setters ...
    # --- Methods ---
    def addPoint(self, point: tuple):
        if(len(self._points) == self.nVertices): return
        self._points.append(point)
```

Code Listing 3.5: Polygon class.

```python
class Entity:
    def __init__(self, a, n, p: int, g, poly: Polygon, point:
    tuple):
        self._address = a
        self._name = n
        self._prime = p
        self._generator = g
        self._polygon = poly
        self._point = point
        self._secret = None
    ... __str__ ...
    ... Getters ...
    ... Setters ...
```

Code Listing 3.6: Entity class.

```
class Prover(Entity):
    def __init__(self, a, n, p: int, g, poly: Polygon, point: tuple):
        super().__init__(a, n, p, g, poly, point)
        self._r = None
    ... __str__ ...
    # Compute Secret X
    def computeX(self):
        sumD = 0
        perimeter = 0
        vertexes = self._polygon.getListOfPoints()
        lenght = len(vertexes)
        # 1: Sum of distances
        # 2: Perimeter
        for i in range(0, lenght):
            ... sumD logic & perimeter logic ...
        # 3: prime
        # 4: generator
        self.setSecret(int(sumD + perimeter + self._prime + self._generator))

    # Execution of Ray Casting
    def pointInsidePolygon(self) -> bool:
        return is_inside_polygon(self._polygon.getListOfPoints(), self._point)

    # Compute Proof
    def proof(self):
        return (pow(self._generator, self._secret) % self._prime) # y = g^x modp

    # Compute a random value in a given interval
    def chooseR(self):
        self._r = randint(0, self._prime - 2)

```

```python
    # Compute personal statement
    def cStatement(self):
        return (pow(self._generator, self._r) % self._prime) # c = g^r modp

    #Compute statement from challenge
    def challengeStatement(self, challenge):
        if challenge == 0:
            return ((self._secret + self._r) % (self._prime - 1)) # (x + r) modp
        return self._r
```

Code Listing 3.7: Prover class.

```python
class Verifier(Entity):
    def __init__(self, a, n, p: int, g, poly: Polygon, point: tuple):
        super().__init__(a, n, p, g, poly, point)
        self._challenge = None
        self._received = []
    ... __str__ ...
    ... Getters ...
    ... Setters ...
    # Save a received statement
    def setReceivedValue(self, value):
        self._received.append(value)
    # Choose challenge
    def challenge(self):
        choice = randint(0, 1)
        self.setChallenge(choice)
    # Verify Proof
    def verify(self):
        value1 = 0
        value2 = 0
        if self._challenge == 0:
              # Verify for challenge 0
            value1 = (self._received[1] * self._received[0]) % self._prime
            value2 = pow(self._generator, self._received[2])
    % self._prime
            if value1 == value2: return True
            return False
        # Verify for challenge 1
        value1 = self._received[1]
        value2 = pow(self._generator, self._received[2]) %
    self._prime
        # Accept or Reject
        if value1 == value2: return True
        return False
```

Code Listing 3.8: Verifier class.

The logic for the DLP algorithm is implemented in the third file of the pyZKP library, *zkp.py*.

zkp.py : Contains the logic to implement the Interactive zkPoL strategy described in Section 2.2. It implements three functions that are required to execute the DLP algorithm, and generate parameters to set it up.

1. Function **aPrime(start_value, end_value)** $\rightarrow$ **int**: Computes a random prime number in the interval [start_value, end_value].
2. Function **randomZp(prime)** $\rightarrow$ **int**: Generates an integer of the field $\mathbb{Z}^p$, where p is a predefined prime.
3. Function **zkp(prover, verifier)** $\rightarrow$ **bool**: Executes the I-ZKP Protocol, defined by the DLP algorithm.

```
def aPrime(initial: int, final: int) -> int:
    return randprime(initial, final)

def randomZp(p: int) -> int:
    return randint(1, p-1)

def zkpPIP(prover: Prover, verifier: Verifier) -> bool:
    # Witness Phase:
        # Prover computes Y, sends it to Verifier
    y = prover.proof()
    verifier.setReceivedValue(y) # y: verifier.received[0]

    prover.chooseR()
    c = prover.cStatement()
    verifier.setReceivedValue(c) # c: verifier.received[1]

    # Challenge Phase:
        # Verifier randomly chooses between 0 and 1, sends it to Prover
    verifier.challenge()
        # Prover computes a statement based on the challenge
    challengeSt = prover.challengeStatement(verifier.getChallenge())
    verifier.setReceivedValue(challengeSt) # challengeSt: verifier.received[2]

    # Verifier accepts or rejects Proof
    return verifier.verify()
```

Code Listing 3.9: zkp.py functions.

3.2 zkSNARK DApp

The second zkPoL strategy to receive an implementation is the zkSNARK zkPoL approach described in Section 2.3.1. This applicative is based on the Non-Interactive ZKP Protocol zkSNARK and is a much more reliable and efficient implementation of a zkPoL strategy. The application is called *zkSNARK DApp* because it is, in fact, a decentralized application, as it involves the Ethereum blockchain as a decentralized always-on Verifier.

3.2.1 Frameworks

The application features three distinct technologies required to implement all the aspects of the zkSNARK N-I zkPoL architecture, described in Section 2.3.1. Therefore, three frameworks are used, and each one has its own scope and goals to achieve.

- **Circom**: Circom is a compiler built on Rust for compiling arithmetic circuits, written in the Circom programming language.
- **SnarkJS**: It is a JavaScript and Pure Web Assembly implementation of zkSNARK and PLONK schemes. It uses the Groth16 Protocol as a ZKP construction scheme.
- **Solidity**: Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs that govern the behavior of accounts within the Ethereum state.

3.2.2 Application Structure

The zkSNARK DApp is organized with the following directory structure.

```
zkSNARK DApp
├── Circuits
│   ├── Point-in-Polygon Circuit
│   ├── Inputs files
│   └── Circom generated files
├── PoT Ceremony
│   ├── Ceremony files
│   └── Prover, Verifier, Setup files
└── Contracts
    └── Verifier Smart Contract
```

The applicative strictly follows the architecture of the zkSNARK zkPoL strategy. However, it is important to define the logic flow of the application, which describes how the steps required from the architecture are linked to the frameworks involved. Figure 3.1 outlines a zkSNARK DApp flow chart.

1. Implementation of an arithmetic circuit that solves the PIP problem (in this case, the Ray Casting algorithm) written with the Circom language.

2. Through the Circom compiler, obtain a low-level representation of the arithmetic circuit (as the R1CS file could be).

3. Implementation with SnarkJS of the structural basis of the strategy.

   (A) (Optional) Start a Powers of Tau Ceremony.

   (B) Compute the witness (secret) from the low-level representation of the circuit.

   (C) Generate a trusted Setup.

   (D) Generate the zkSNARK Proof.

   (E) Generate a Solidity Smart-Contract Verifier or verify the Proof locally.

4. Deployment of the Smart-Contract Verifier on Ethereum and verify the Proof on-chain.

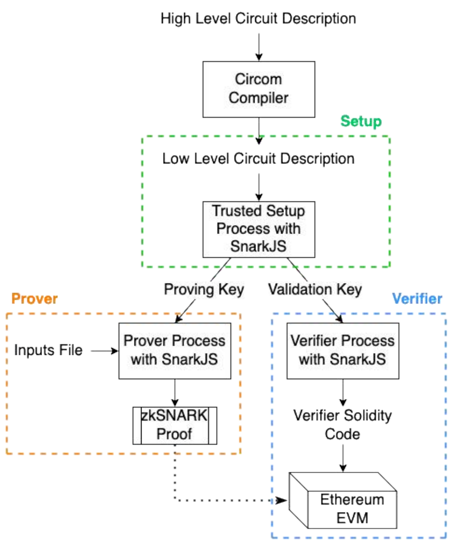
High Level Circuit Description
Circom Compiler
Setup
Low Level Circuit Description
Trusted Setup Process with SnarkJS
Proving Key
Validation Key
Prover
Verifier
Inputs File
Prover Process with SnarkJS
zkSNARK Proof
Verifier Process with SnarkJS
Verifier Solidity Code
Ethereum EVM

Figure 3.1: zkSNARK DApp flow chart

3.2.3 Circom

Circom[2] is a circuit programming language and a compiler that allows programmers to design and create their own arithmetic circuits for ZKPs. Circom highly simplifies the construction process of arithmetic circuits in a reliable and developer-friendly way. The latest version (2.0.4) of the Circom compiler is built upon Rust and is the version used to develop a Point-In-Polygon circuit for the zkSNARK DApp.

Properties

The Circom language addresses peculiar characteristics that have to be described before showing the actual implementation of a circuit that solves the PIP Problem.

- **Arithmetic circuits**: The goal of the Circom language is to let a user script an arithmetic circuit to be used for the generation of ZKPs; in reality, the principal ZKP Protocol that requires the use of such circuits is zkSNARK. zkSNARK allows to prove computational statements, but cannot be applied to the computational problem directly: the statement first needs to be converted into the right form, which in this case is an $\mathbb{F}_p$ arithmetic circuit. Those circuits consist of a set of wires that carry values from the field $\mathbb{F}^p$ and connect them to addition and multiplication gates (mod p). The prime p is set by default to the value $p = 21888242871839275222246405745257275088548364400416$ $034343698204186575808495617$.
- **Signals of a circuit**: An arithmetic circuit takes some **input signals** that are values that belong to the finite field $\mathbb{F}^p$, and performs additions and multiplications (mod p) between them, as a form of gates. The output of every gate is considered an **intermediate signal**, except for the last gate of the circuit, which is the **output signal** of the circuit.

[2]Circom - `www.iden3.io/circom`

To work with the zkSNARK Protocol it is required to describe the relations between signals as a system of equations that relate variables with gates. These are called **constraints**, conditions that the signals of the circuit must satisfy.

- **Rank-1 Constraint System (R1CS)**: Having an arithmetic circuit with signals $s_1, ..., s_n$, where n is the number of gates of the circuit, then it must be defined an equation of the following form

  $$(a_1 \cdot s_1 + ... + a_n \cdot s_n) \cdot (b_1 \cdot s_1 + ... + b_n \cdot s_n) \cdot (c_1 \cdot s_1 + ... + c_n \cdot s_n) = 0$$

  which describes a constraint, and must be a quadratic, linear or constant equation. The values a_i, b_i, c_i are coefficients defined by the gates of the circuit. In general, circuits have several constraints, tipically one per multiplication gate, and the set of the constraints describing the circuit is called Rank-1 Constraint System (R1CS).

- **Witness (secret)**: A set of signals, known as an *assignment*, that satisfies the circuit is called a Witness, which is the secret information that zkSNARK uses to compute a Proof.

The following terminal command defines how the compilation of the circuit occurs and what output files can be obtained.

```
circom circuit.circom --r1cs --wasm --sym --c
```

There are four kinds of possible output formats.

- **–r1cs**: Contains the R1CS representation of the circuit in binary format.

- **–wasm**: Generates the directory *./circuit_js* that constains code in the WebAssembly (WASM) format and other files needed to compute a Witness.

- **–sym**: Generates a .sym file, which is a special symbols format used to debug the circuit.
- **–c**: Generates the directory *./circuit_cpp* that constains files needed to compile the circuit in C code.

Circom Language

It is important to take a look at the basics of the Circom Language. The files are saved with the extension *.circom* and the first line of the file must declare which release of the Circom compiler the code has to follow, and is declared with a *pragma* instruction. To create generic circuits, Circom defines the *template* mechanism. These templates are normally parametric on some values that must be instantiated when the template is used. The instantiation of a template is a new circuit object, which can be used to compose other circuits, so as part of larger circuits. Since templates define circuits by instantiation, they have their own input, intermediate, and output signals.

A generic circuit *template* can be declared as follows.

```
pragma circom 2.0.4;
...
template circuit(param_1, ... , param_n) {
... input signals ...
... intermediate signals ...
... gates ...
... output signal ...
}
```

Signals can be named with an identifier or can be stored in arrays and declared using the keyword *signal*. Signals can be defined as input or output, otherwise are considered intermediate signals. The value assigned to the signal can only be of the field $\mathbb{F}_p$.

```
signal input in;
signal output out[n];
signal intermediate;
```

A constraint is imposed with the operator ===, which creates the simplified form of the given equality constraint. Adding such constraint also implies adding an assert statement in the witness code generation. Constraint generation can be combined with a signal assignment with the operator <==, where the signal is assigned on the left-hand side of the operator.

```
out <== 1 - a*b;  // Where "a" and "b" are input signals.
```

A generic value can be assigned to a signal using <- - but is considered dangerous and should, in general, be combined with adding constraints with ===, which describes by means of constraints what the assigned values are. A template can be called inside another circuit template as a **component**.

```
component c = template_circuit(parameters, ...);
```

Ray Casting with Circom

The problem assessed by the application is the Point-In-Polygon problem. As the protocol requires, an implementation of its solution has to be carried on in the form of an arithmetic circuit. Therefore, the Ray Casting algorithm described in Section 2.1.3 will be implemented using the Circom language. The implementation follows rigorously the steps required from the Algorithm 1, and it implements 5 templates:

- template **PointInPolygon(n)**: implements the Ray Casting logic as a circuit. The parameter n is the number of edges of the Polygon.
- template **Intersects()**: implements the logic to test if an intersection happens between two edges, as a circuit.
- template **Orientation()**: implements the logic to obtain the orientation of a triplet of points, as a circuit.
- template **OnSegment()**: implements the logic to check if a point lies on an edge, as a circuit.

- template **OnVertex()**: implements the logic to check if a point lies on a vertex, as a circuit.

The main circuit is described from the PointInPolygon(n) template, which takes two signals as inputs: one private signal to store the point-in-space and a public signal that stores the polygon. The output of the circuit is an output signal that defines the result of the algorithm.

```
1  template PointInPolygon(n){
2      assert(n >= 3);
3  
4      // Main signals of the circuit
5      signal input point[2];
6      signal input polygon[n][2];
7      signal output inside;
8  
9      // Variable that stores the results of the cases of
     intersection of the PIP
10     var sum = 0;
11 
12     ... Ray Casting Logic ...
13 
14     // If sum is odd, proceed to compute a Witness.
15     inside <-- sum % 2;
16     inside === 1;
17 }
18 component main{public[polygon]} = PointInPolygon(4);
```

Code Listing 3.10: PIP main template

As it can be seen in Line 18 of the Code Listing 3.10, the Circom compiler wants to know, at compile time, the actual value of the parameters the main template needs to implement: in this case, the compiler will generate output files that are implementable only for polygons of 4 vertices. This is because the Circom library lets the developer implement an arithmetic circuit of signals, wires, and gates as if it was a real physical circuit. Therefore, when using Circom, it has to be accounted for that we are working with the concept of the **unknown**. In Circom, constant values and template parameters are always considered *known*, while signals are always considered *unknown*. Expressions depending only on knowns are considered knowns, while those depending on unknowns are considered unknowns. At compile time, the signals are always empty, and the inputs will be filled at a later stage when the generation of a Witness is required. The concept of the unknowns directly involves the *control flow* of the logic of the circuit: conditional statements, for-loops, and while-loops are supported by the language but cannot depend on signals. To run conditional statements with signals, custom templates must be implemented. For example, To check if two input signals are equals, two templates must be implemented: the *IsEqual()* and *IsZero()* templates. These templates define the result of the control flow as a signal output, which is 0 if the statement is false, and 1 otherwise.

```
template IsZero() {
    signal input in;
    signal output out;
    signal inv;
    inv <-- in!=0 ? 1/in : 0;
    out <== -in*inv +1;
    in*out === 0;
}

template IsEqual() {
    signal input in[2];
    signal output out;
    component isz = IsZero();
    in[1] - in[0] ==> isz.in;
    isz.out ==> out;
}
```

Code Listing 3.11: Templates that checks if two input signals are equal or if a signal is zero.

There are other control flow templates to obtain other conditional statements: *LessThan(n)* as $<$, *LessEqThan(n)* as $\leq$, where the value n is the number of bits used to define the signal inputs; and so on with other conditional statements. These templates are widely used in the implementation of the Ray Casting logic because the data used to solve the PIP problem is handled as signals. One example of signal control flow in the Ray Casting circuit can be found in the *OnSegment()* template.

```
template OnSegment(){
    signal input p[2];
    signal input q[2];
    signal input r[2];
    signal output out;
    ...
    var max1 = maxN(a, b);
    ...
    component firstCheck = LessEqThan(bits());
    firstCheck.in[0] <== q[0];
    firstCheck.in[1] <-- max1;
    ...
    // Load into a variable the output value
    // of the LessEqThan() check
    var checks = firstCheck.out + (...other checks...) ;
    ...
    out <-- result;
```

Code Listing 3.12: Example of signal control flow in the Ray Casting circuit.

3.2.4 SnarkJS

SnarkJS[3] is an npm package that contains code to generate and validate ZKPs from the artifacts produced by the Circom compiler. This library includes all the tools required to perform trusted setup multi-party ceremonies: including the universal Powers of Tau ceremony, and the Phase-2 circuit-specific ceremonies. For the purpose of the zkSNARK DApp, this library is used to implement a complete zkSNARK Protocol, which includes both Phase 1 (PoT Ceremony processes) and Phase 2 (Circuit-specific processes). All the commands defined to use this library are terminal commands.

[3]SnarkJS - `www.github.com/iden3/snarkjs`

Phase 1: PoT Ceremony Processes

Phase 1 creates a new Powers of Tau ceremony. All of the files will be stored in the *zkSNARK-DAPP/PoT-Ceremony* directory. The first thing to do is start a new PoT Ceremony, which will be defined with the following command:

```
snarkjs powersoftau new bn128 12 pot12_0000.ptau -v
```

The *new* command is used to start a PoT ceremony. The first parameter after *new* refers to the type of curve we wish to use, in this case, the *bn128* group. The second parameter, which is the value 12, describes the exponent n in the 2^n relation. This represents the maximum number of constraints in a circuit that the ceremony can accept, which in this case is $2^{12} = 4096$. The maximum value supported here is 28, which means the library can securely generate zkSNARK parameters for circuits with up to $2^{28}(\approx 268 million)$ constraints.
The second step is to contribute multiple times to the ceremony.

```
snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.
 ptau --name="First contribution" -v
... second contribution ...
... third contribution by a third party software ...
```

The *contribute* command creates a ptau file with a new contribution. It takes as input the transcript of the protocol so far, in this case, *pot12_0000.ptau*, and outputs a new transcript, *pot12_0001.ptau*, which includes the computation carried out by the new contributors.

To finalize Phase 1 and prepare for Phase 2, it is necessary to apply to the ceremony a *random beacon*.

```
snarkjs powersoftau beacon pot12_0003.ptau pot12_beacon.
 ptau 0239...0939 -n="Final Beacon"
```

The *beacon* command creates a file.ptau with a contribution applied in the form of a random beacon. The PoT Ceremony can now be prepared to be used in Phase 2.

```
 snarkjs powersoftau prepare phase2 pot12_beacon.ptau
  pot12_final.ptau -v
```

Phase 2: Circuit Specific Processes

Phase 2 takes the R1CS and WASM representation of the arithmetic circuit and uses it to execute the Setup, Prover, and Verifier processes through the SnarkJS library. The first thing to do is to generate a Witness from the circuit with the following terminal command, which outputs the result in a *witness.wtns* file.

```
circuit_js$ node generate_witness.js circuit.wasm ../input.
   json ../witness.wtns
```

Then, the Trusted Setup process begins. Having established a universal PoT Ceremony in Phase 1, the trusted Setup will be based on the Groth16 protocol.

```
snarkjs groth16 setup circuit.r1cs pot12_final.ptau
   circuit_0000.zkey
```

This command generates a reference file containing a *zkey*. This key is equivalent to the concept of Proving Key (PK). The first zkey generated is just a model key, as it does not require any Phase 2 contribution to be generated, and is considered to be dangerous. Therefore, to generate a final *circuit_zkey*, it is necessary to first apply at least one contribution to the PoT ceremony file. After having contributed multiple times, a safe zkey can be generated.

```
snarkjs zkey contribute circuit_0000.zkey circuit_0001.zkey
   --name="1st Contributor Name" -v
snarkjs zkey contribute circuit_0001.zkey circuit_0002.zkey
   --name="Second contribution Name"
...other N contributions...
snarkjs zkey beacon circuit_000N.zkey circuit_final.zkey (
   beacon) -n="Final Beacon phase2"
```

Being in possession of a Proving Key, the Setup has to generate a Verification Key (VK). The VK will be exported to a JSON file.

```
snarkjs zkey export verificationkey circuit_final.zkey
   verification_key.json
```

With this command, the Setup process terminates. It is now time for the Prover to compute a zkSNARK Proof.

```
snarkjs groth16 prove circuit_final.zkey witness.wtns proof.
   json public.json
```

This command generates the files *proof.json* and *public.json*: the latter contains the actual proof, whereas the first contains the values of the public inputs and output.

Being in possession of a valid zkSNARK Proof, the Prover process terminates. To verify the Proof, there are two possible ways: the first is to locally verify the Proof with a SnarkJS Verifier; the second option is to generate Solidity code from the circuit implemented, that can be published on the Ethereum blockchain as a Smart Contract that acts as a decentralized always-on Verifier.

```
snarkjs groth16 verify verification_key.json public.json
   proof.json
```

Code Listing 3.13: Local SnarkJS Verifier

We use this command to verify the proof, passing in the *verification_key* we exported earlier. If all is well, it is possible to observe “OK” in the output of the console. This means that the zkSNARK Proof is valid.

3.2.5 Smart Contract Verifier

It is possible to turn the Verifier into a smart contract, written with Solidity. Smart contracts are computer programs that are executed inside a peer-to-peer network, like the Ethereum blockchain. Solidity is one of the most popular languages for writing smart contracts to be deployed on the EVM (Ethereum Virtual Machine). To generate a contract related to the arithmetic circuit used for the zkSNARK Proof, the following command must be executed.

```
snarkjs zkey export solidityverifier circuit_final.zkey ../
    Contracts/verifier.sol
```

Code Listing 3.14: Decentralized Solidity Verifier

The command *generateverifier* takes a verification key as input, in this case, verification_key.json, and generates a Solidity version of the Verifier in a file of our choice, *verifier.sol*, stored in the zkSNARK-DApp/Contracts directory. The *verifier.sol* contract generated contains two contracts: a Pairings contract and a Verifier contract. For our purposes, we just need to deploy the Verifier contract.

To verify the Proof on-chain, we can upload *verifier.sol* directly into the Remix IDE. Remix is an open-source tool that makes it possible to write and deploy Solidity contracts straight from the browser. To locally run and test the Verifier Smart Contract, the Remix IDE makes available a **JavaScript VM** option. The JavaScript VM option runs a JavaScript Virtual Machine blockchain within the browser, which allows us to deploy and send transactions to a blockchain within the RemixIDE directly in the browser. This is particularly useful for prototyping, especially since no dependencies are required to be installed locally.

Once the Verifier contract is published, the last thing to do is to verify the zkSNARK Proof generated. The Verifier contract deployed in the last step has a function called *verifyProof*. The function takes as input four parameters and returns **True** if the Proof and the inputs are valid. To generate these parameters, run the command:

```
snarkjs generatecall --proof proof.json --public public.json
```

Code Listing 3.15: Decentralized Solidity Verifier

The *generatecall* command takes two inputs: the ZKP we want to use, in this case *proof.json*, and the public inputs/outputs contained in *public.json*. Next, we will need to copy the output of this command into the parameters fields of the *verifyProof* method in Remix and click on the **call** button. If everything works fine, this method should return **True**; otherwise, the result will be verifiably **False**.

Chapter 4

Performance Testing and Evaluation

This chapter presents a performance evaluation of the implementations of the zkPoL strategies presented in Chapter 3. It is important to outline a behavioral pattern of the applications to analyze their performance and evaluate their strengths and weaknesses. The results are obtained through a process of testing that involves the main characteristics and phases of each system, and they mainly consist of temporal measurements of the various computational processes. The evaluation consists in performing a series of predetermined tests on each system involved and collecting the relative results. For each type of test, a certain number of executions has been made, in order to exclude the presence of unreliable results given by unpredictable states of the computer on which they are performed. Due to this reason, the presented data are the average of the different executions and are measured in milliseconds $[ms]$. The final purpose of this evaluation is to provide qualitative and approximated information about the performance and constraints of the approaches presented in the previous chapters, outlining an indicative view of the potential applications that can be built based on these technologies.

4.1 Tests Structure

The tests are performed using the implementations of pyZKP and zkSNARK DApp, respectively presented in Section 3.1 and Section 3.2. Each application is based on a specific zkPoL strategy, thus they share the same basic concepts and parameters. The focus of the tests is the evaluation of the performance of the processes of the systems based on the variation of an essential parameter and on the variation of the public polygon. Despite the methodologies used being quite similar, it is important to highlight the differences between the two implementations before making a direct comparison of the results. As previously mentioned, pyZKP is a personal implementation of an I-ZKP for purely informational purposes, therefore to be used as a demonstrator of the protocol. On the other hand, zkSNARK DApp is a much more complete set of open source libraries that implement the zkSNARK protocol for arithmetic circuits, carrying out various security procedures in order to guarantee data integrity. As a consequence of this, pyZKP worst-case scenario performance will result much better than on the DApp side. Therefore, from these tests, it is possible to deduce a series of quantitative information on the individual progress of the implementations, but in such terms, a direct comparison between the two is difficult to apply. However, it is possible to outline a comparison by looking at the performance trend of the two tests varying the same parameters, that is analyzing in qualitative terms how the variation of the usual parameter affects the performance of the different techniques.

The tests are performed by processing the system's PoL data based on a simple cartesian coordinate system.

- **Private Location**: Tested as a *point* $P(x_P, y_P)$ of cartesian coordinates.
- **Public Perimeter**: Tested as a **polygon**, which is a list of n edges.

These parameters are the basis of the tests and, in order to obtain data that describes the behavior of the technologies involved in the systems, a variation of such parameters is performed. As the zkPoL strategy imposes, each implementation has scripted a version of the Ray Casting algorithm, which helps the system to have the knowledge if the point P lies inside the polygon. In order to test the algorithms in their entirety, the points P tested are chosen always to be inside, so that both systems are able to execute all the steps; therefore, in this way is possible to perform a *worst case* scenario testing, assuming to obtain the maximum effort from any process involved in the application tested. The cartesian coordinates tested are processed as digits of the order of magnitude up to 10^6, which can be described with 32-bit digits and let the tests easily reach values of the order of 1 million. The tests, as mentioned earlier, are performed through the variation of these parameters, mainly involving the characteristics of the polygon. There are three types of variations used to evaluate the performance of the systems:

- Variation of the number of points of the polygon.
- Scaling of the order of magnitude of the cartesian coordinates of n-edged polygons.
- Expansion of the area of n-edged polygons on their plane with a scaling of the order of magnitude of the cartesian coordinates.

The testing phase is carried out with the use of a Macbook Pro® personal computer. Its hardware specifics are presented in Table 4.1.

Macbook Pro® (2017)	
CPU	2,3 GHz Intel® Core i5 dual-core
RAM	8 GB 2133 MHz LPDDR3
OS	macOS® Monterey 12.4 (21F79)

Table 4.1: Macbook Pro® Characteristics

To test both systems, different software tools are used.

- pyZKP Tools: The pyZKP system is tested through the use of the function *time_ns()*, which is implemented from the *time* module made available by the Python Standard Library. The function is used to catch the computation time, measured in nanoseconds, that each process tested spends to perform its task.

- zkSNARK DApp Tools: The zkSNARK DApp system processes are mainly invoked through terminal commands; therefore, to test them it is necessary to use the *time* command for UNIX and UNIX-like systems, which determines the duration of execution of a particular command.

4.2 pyZKP Evaluation

The pyZKP library implements an I-zkPoL. The processes that need to be tested to observe their performance are the *secret generation*, the *proving process*, the *verification process* and the *interactive protocol execution.*

The first testing process is the testing of the performance of the algorithms based on the growth of the number of edges of a polygon. As it can be seen from the plots presented in Figure 4.1, the Prover and Verifier processes remain time-stable through the growth of the number of edges of the polygons. An interesting result is otherwise obtained from the analysis of the Secret Generation and Protocol Executions processes; both appear to behave with a linear growth as the number of edges increases. That is mainly caused by the structure of the Secret Generation process, which includes the *perimeter* parameter, and with the growth of the polygon is also expected the consequent growth of the size of the secret, which leads to higher computation times with linear growth. Figure 4.1 presents the plots that confirm these claims.

The second testing phase delineates the performance of the processes in relation to the translation of the polygons in the plane, with a $\log_{10}$ increment, working with cartesian coordinates up to 10^6 values. The results do not differ from those outlined in Figure 4.1 and remain stable to those values. Therefore the pyZKP library seems to maintain performance even when working with coordinates of a higher order of magnitude. Figure 4.2 presents

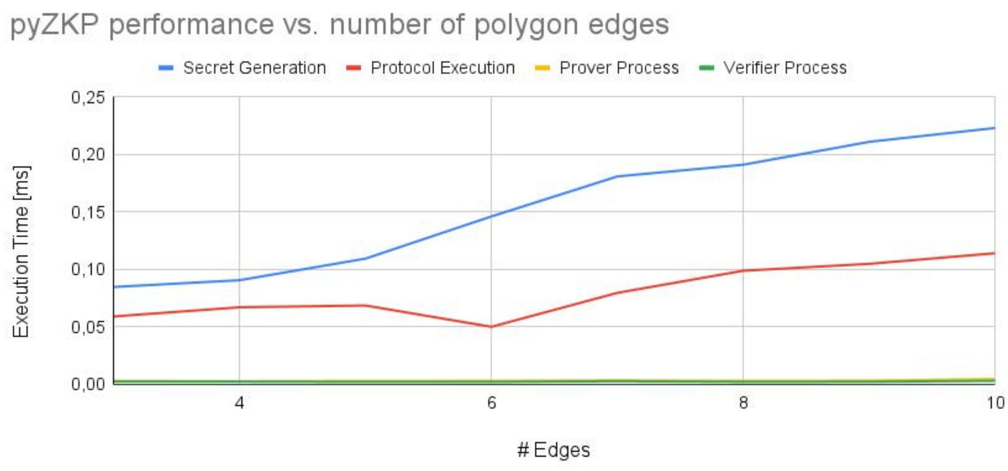

Figure 4.1: pyZKP's performance (execution time) with respect to the edges of the polygon

the plots of the performance of a 3-edged polygon based on its translation in the plane.

The third testing phase involves the expansions of the area of the polygons following a $\log_{10}$ growth. As for the second testing phase, the results remain stable for the *secret generation* and *verification* processes in reflection of the data presented in Figure 4.1. But an interesting result seems to outline an important behavior in relation to the *proving* and *protocol execution* processes: the more the area is expanded, the more the execution time of these two processes grows with linear trend. As was mentioned in the result of the first testing phase, this performance behavior is due to the structure of the generation of the secret, which involves the sum of various parameters, with the inclusion of the perimeter of the polygon and the sum of distances from P to the edges. These two values in particular grow in relation to the growth of the size of the polygon. Then, the secret is passed to the Prover, which computes a Proof π using that secret value as a discrete logarithm of π. When working with polygons that have coordinate values in the order of magnitude of 10^6, the computation time of the secret generation and the protocol execution grows linearly. Figure 4.3 presents the plots that show the results of this testing phase.

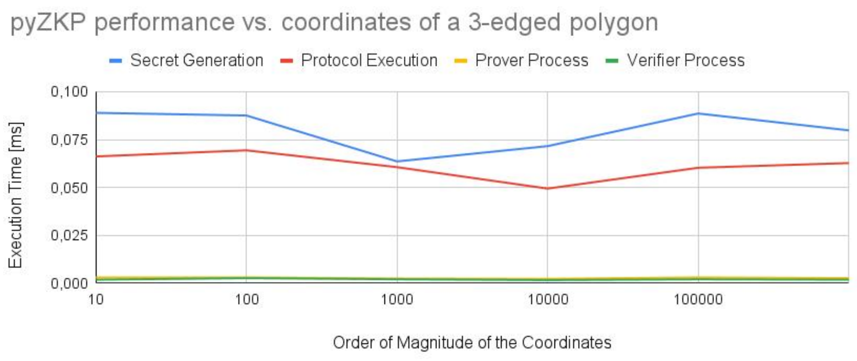

Figure 4.2: pyZKP's performance (execution time) with respect to the order of magnitude of the coordinates of a 3-edged polygon on its plane

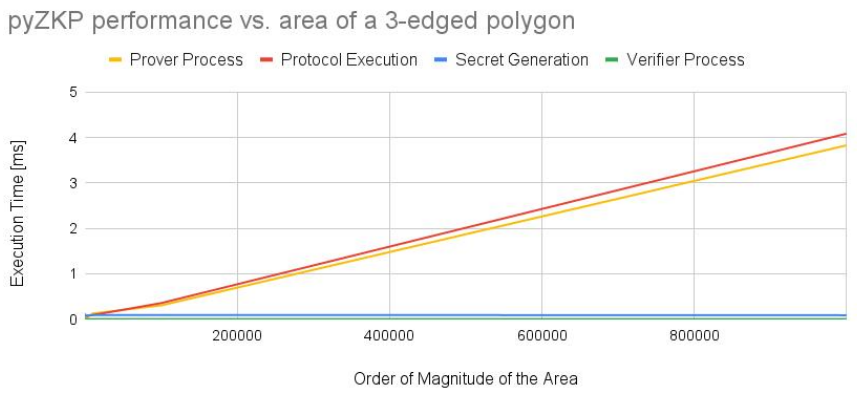

Figure 4.3: pyZKP's performance (execution time) with respect to the order of magnitude of the area of a 3-edged polygon

4.3 zkSNARK DApp Evaluation

The zkSNARK DApp implements a NI-zkPoL. The processes that have to be tested are more numerous with respect to the pyZKP library; these processes are the *circom process*, the *secret generation*, the *Setup process*, the *proving process* and the *verifying process*. There are also two parameters that need to be accounted for, which are the number of constraints used by the arithmetic circuit and the consequent exponent k used to generate a Powers of Tau Ceremony (the constraints accepted from the PoT are defined by 2^k).

The first testing phase is focused on observing the performance of the application processes with respect to the growth of the edges of the polygons tested. The results can be seen in the plots of Figure 4.4. By evaluating these data, it can be seen that the performance of the Setup and Verifier processes remains stable through the growth of the number of edges. What seem to have a different behavior are the Circom, Secret generation, and Prover processes. Each process has a linear growth in relation to the increasing number of edges used for the polygons; the Circom process seems to have a more stable linear growth, with an increment of $0,1s$ per edge added to the polygon. Otherwise, the other two processes have a more steep linear growth. Another important observation that can be made from these results is the number of constraints generated from the arithmetic circuit that implements the Ray Casting algorithm: each edge of the polygon is described by about 2000 constraints. Based on the number of constraints generated by the circuit, a specific PoT Ceremony, that can handle this parameter, has to be used. Figure 4.4 shows the plots that affirm these results.

The second testing phase involves the evaluation of the performance of the application in relation to the translation of the polygon in their plane with a $\log_{10}$ increment, working with cartesian coordinates up to 10^6 values. The tests were conducted on polygons that require different PoT ceremonies setups, and either result does not show any performance boost or leak as greater the value of the coordinates could reach. In fact, the performance of all processes remains stable to the data collected and shown in Figure 4.4.

Figures 4.5 and 4.6 show the results of this testing phase.

The third, and last, testing phase of the performance of this application involves the expansion of the area of the polygons that follows an increment of $\log_{10}$ growth. The approach taken in the second testing phase is now again presented for the third testing phase; the polygons tested use different PoT ceremonies setups to obtain data that could potentially show a peculiar performance behavior. The results, which can be seen in Figure 4.7 and 4.8, remain stable in relation to the data in Figure 4.4.

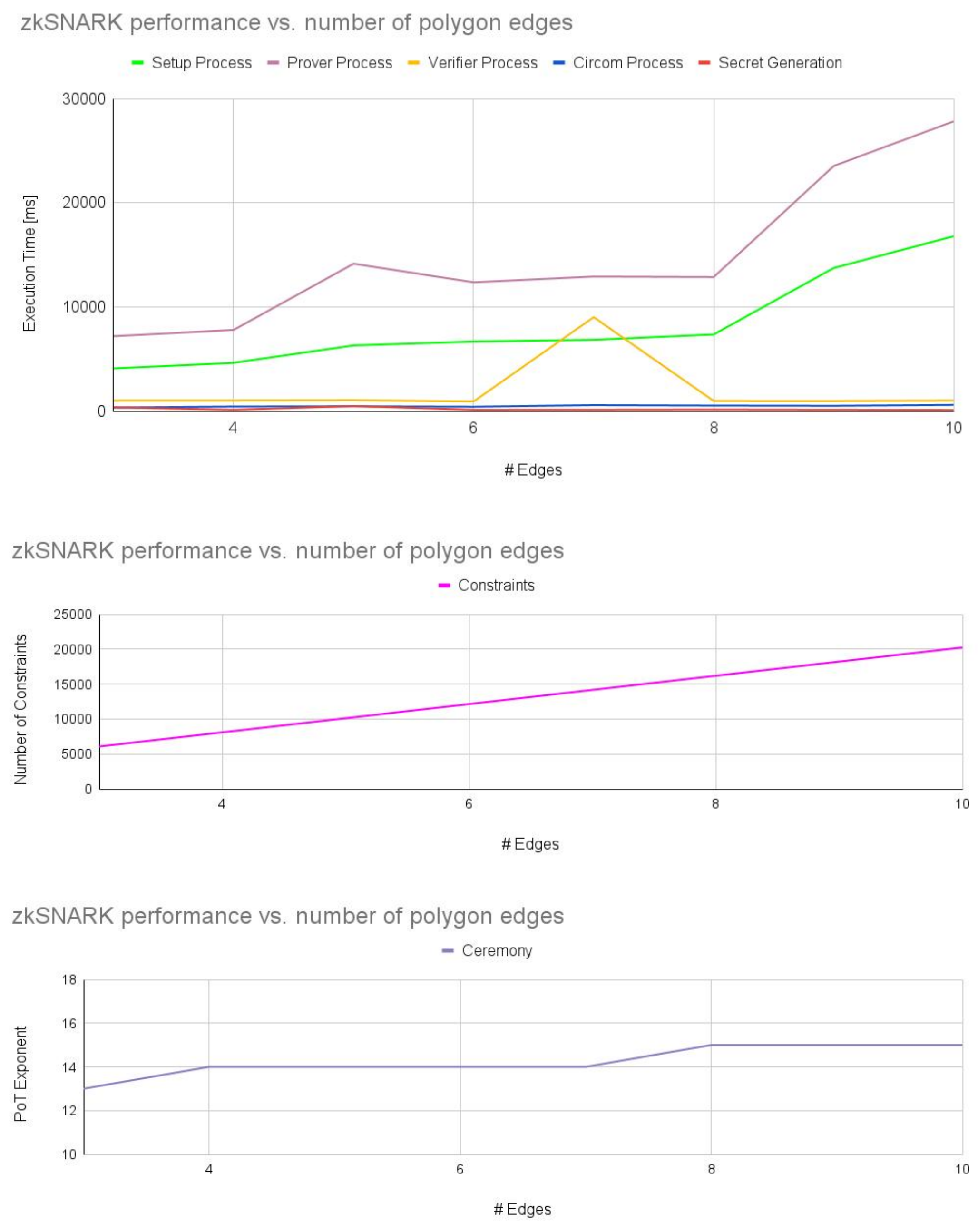

Figure 4.4: zkSNARK's performance with respect to the number of edges of the polygon

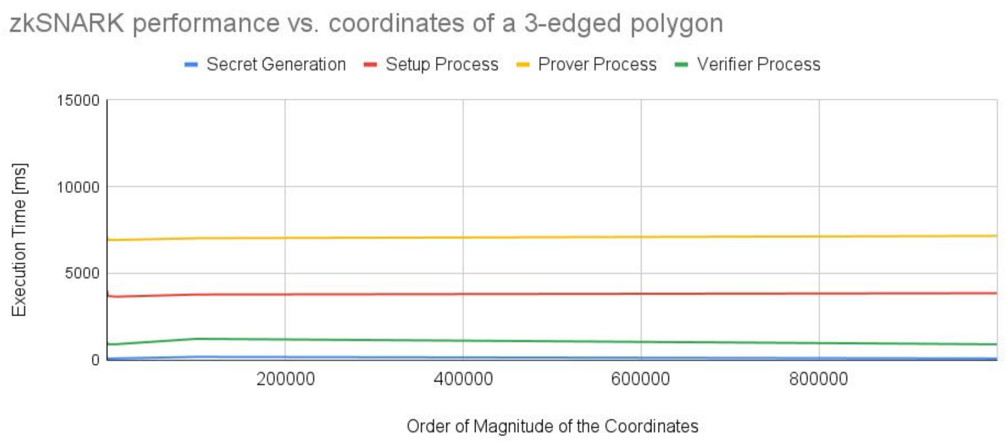

Figure 4.5: zkSNARK's performance (execution time) with respect to the order of magnitude of the coordinates of a 3-edged polygon on its plane

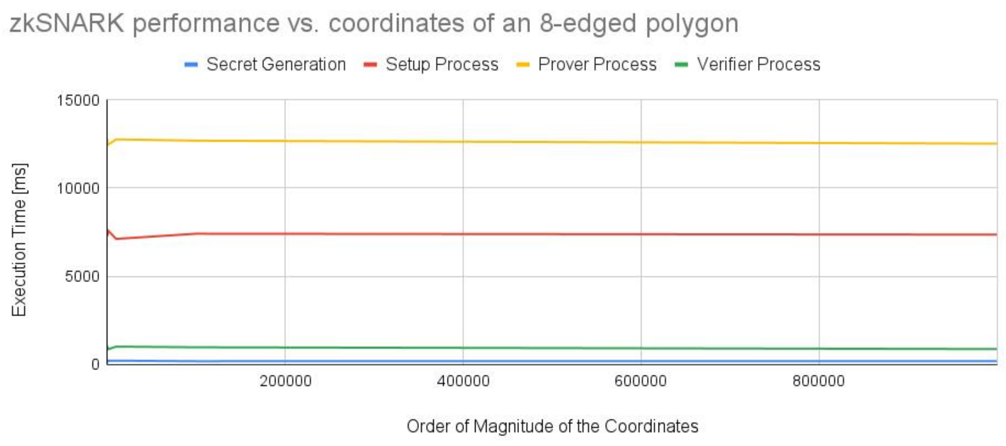

Figure 4.6: zkSNARKS's performance (execution time) with resepect the order of magnitude of the coordinates of an 8-edged polygon on its plane

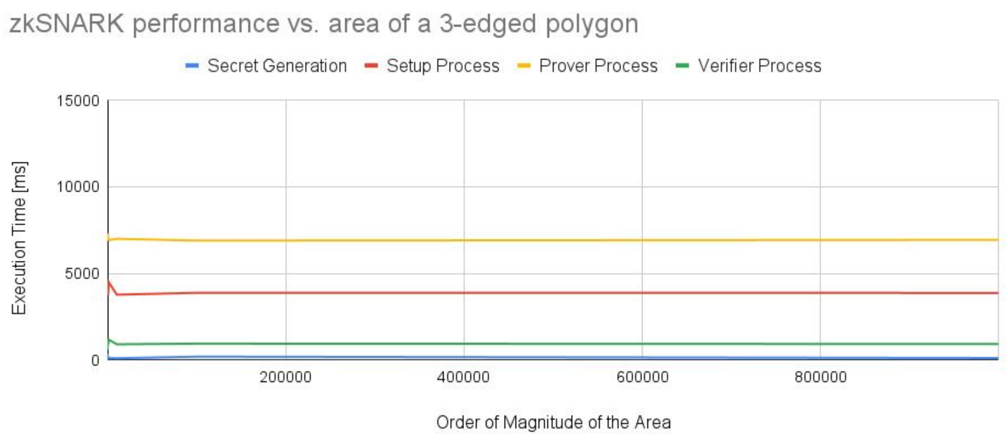

Figure 4.7: zkSNARKS's performance (execution time) with respect to the order of magnitude of the area of a 3-edged polygon

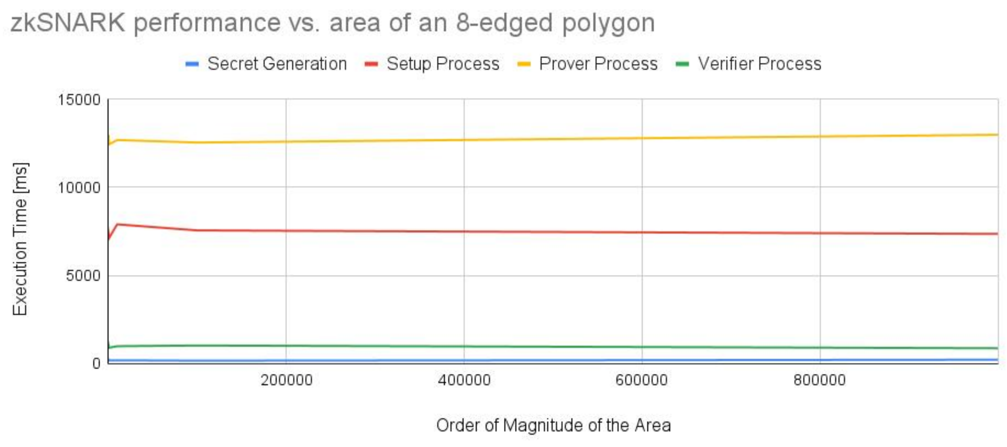

Figure 4.8: zkSNARK's performance (execution time) with respect to the order of magnitude of the area of an 8-edged polygon

4.4 Results

The direct comparison of the performance evaluation of the pyZKP and zk-SNARK systems shows very different execution times of the involved processes. What has to be counted into the picture is the fact that a single interaction has been used to test the library with the goal of always obtaining the worst-case scenario by focusing on the single processes of the system. The tests have shown that pyZKP is much more efficient than the zk-SNARK DApp in terms of execution time. However, from the performance testing of the pyZKP library, an interesting behavioral pattern is outlined: the greater the order of magnitude of the area of the polygon tested, the greater the execution time for the secret-generation process and the protocol execution, with a linear trend. On the other hand, the same tests have been imposed for the zkSNARK DApp and, even though its execution times are higher with respect to pyZKP, the performance remains stable with almost constant execution time with respect to expanding polygons. In conclusion, pyZKP is a very promising solution, but still requires some fine-tuning to achieve performance stability with respect to the problem instance - in particular, the size of the considered area.

Conclusions

The purpose of this Thesis project was to study and implement advanced strategies for the PoL problem based on ZKP Protocols, defined as Zero-Knowledge Proof of Location strategies. What has been accomplished is the development of two different zkPoL systems, namely the pyZKP library and the zkSNARK DApp. To define these systems, and make conclusions for this Thesis, all the concepts and technologies involved have been gradually presented.

Before any type of structural planning and design for zkPoL strategies, an in-depth analysis of the PoL problem, the ZKP cryptographic method, and the ZKP Protocols relevant for this Thesis have been presented in Chapter 1. Only when the theory behind these technologies was better understood, it was possible to define a generic zkPoL strategy and approach it with different structural parameters based on the ZKP Protocols used, which can be found in Chapter 2. Multiple strategies have been outlined based on Interactive and Non-Interactive-ZKPs (such as the Discrete Logarithm Proving algorithm, zkSNARK, and Bulletproofs protocols). The implementations of pyZKP and zkSNARK DApp have been described in Chapter 3. Lastly, it was important to test the performance of these two systems and confront them with each other in the case interesting results were found: as described in Section 4.4, pyZKP can be seen as a very promising solution, but still requires some fine-tuning to achieve performance stability with respect to the problem instance - in particular, the size of the considered area. On the other hand, the zkSNARK DApp, even though it performs at a higher execution time, represents a more stable system in relation to the goals of the zkPoL. From these results, it appears that the execution times, with respect to both systems, are reasonably low such that it is possible to find real fields of application for these techniques, like sports race-tracking, satellite-based navigation software, and self-reported positioning for Augmented Reality or Virtual Reality software games could be.

This Thesis lays down the foundations for the development of Proof of Location strategies based on ZKP Protocols, but the work shown in these chapters could be perfected and implemented with other technologies. This means that, with respect to the zkSNARK DApp, a more concrete and in-depth linking with blockchains as decentralized always-on Verifiers, other than Ethereum, could be tackled. Another aspect that this Thesis introduced but could not present an implementation for, is the Bulletproofs ZKP Protocol. This protocol is, overall, the most efficient ZKP Protocol that has been theorized, but lacks actual libraries that implement it in its entirety. Therefore, Section 2.3.2 presented various approaches that could be used to implement a zkPoL strategy with Bulletproofs, but could not be implemented due to the lack of libraries that implement Bulletproofs with the properties needed for the zkPoL strategy. Further development could be, once the right Bulletproofs technologies are available, the implementation of a Bulletproofs-based zkPoL strategy, and confront it with the pyZKP and zkSNARK DApp systems.

Bibliography

[1] Michele Amoretti, Giacomo Brambilla, Francesco Medioli, and Francesco Zanichelli. Blockchain-based proof of location. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 146–153. IEEE, 2018.

[2] Xiaoqiang Sun, F. Richard Yu, Peng Zhang, Zhiwei Sun, Weixin Xie, and Xiang Peng. A survey on zero-knowledge proof in blockchain. *IEEE Network*, 35(4):198–205, 2021.

[3] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[4] Wikipedia contributors. Zero-knowledge proof — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Zero-knowledge_proof&oldid=1087608345`, 2022.

[5] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, Boston, MA, third edition, 2013.

[6] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 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, page 326?349, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Xavier Salleras and Vanesa Daza. Zpie: Zero-knowledge proofs in embedded systems. *Mathematics*, 9(20):2569, 2021.

[8] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct {Non-Interactive} zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014.

[9] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[10] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

[11] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012.

[12] Dmitry Khovratovich. Bulletproofs. In *Bulletproofs **, 2019.

[13] Eduardo Morais, Cees van Wijk, and Tommy Koens. Zero knowledge set membership. *none*, 2018.

[14] Desmond Schmidt, Kenneth Radke, Seyit Camtepe, Ernest Foo, and Michał Ren. A survey and analysis of the gnss spoofing threat and countermeasures. *ACM Comput. Surv.*, 48(4), may 2016.

[15] Wikipedia contributors. Point in polygon — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Point_in_polygon&oldid=1083655916`, 2022.

[16] Christian Reitwiessner. zksnarks in a nutshell. *Ethereum blog*, 6:1–15, 2016.

[17] Jan Camenisch, Rafik Chaabouni, et al. Efficient protocols for set membership and range proofs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 234–252. Springer, 2008.