POLITECNICO DI TORINO

Master degree course in Computer engineering

Master Degree Thesis

VerComp: A Framework for Verifiable Computation in Cloud Environments Using zk-STARK Proofs

Supervisor
prof. Antonio Lioy

Candidate
Riccardo Salvatelli

Anno accademico 2023-2024

Alla mia famiglia, che mi ha permesso di raggiungere questo traguardo

Summary

Zero-Knowledge Proofs (ZKPs) have recently emerged as cutting-edge technologies capable of verifying the truthfulness of a statement without revealing any underlying information but only proving its correctness.

This thesis takes a closer look at the study of ZKPs and in particular investigates the zk-STARK (Zero-Knowledge Scalable Transparent Arguments of Knowledge) protocol. It also studies the property of verifiable computation, analysing protocols in which this is combined with ZKP technologies.

The main objective of this thesis is to realise and discuss the implementation of a software called VerComp, which simplifies and verifies the execution of software in server infrastructures. It allows even machines with reduced computational capacity to delegate the execution of applications to external servers and then obtain back ZKP proofs attesting to their correct execution.

This study examines the state of the art with regard to ZKPs and verifiable computation, with the aim of providing an overview of current progress. The thesis then presents a detailed description of VerComp talking about its design, implementation and presents some possible uses.

Key findings of the thesis highlight the possibility of integrating and using ZKPs in new environments outside the blockchain environment, thus enabling their global use for all kinds of applications. In addition, it shapes new use cases and scenarios in which mutually unknown (untrusted) entities can still communicate and share proofs to prove possession of information or the execution of an operation.

Acknowledgements

Vorrei ringraziare il Prof. Lioy per avermi dato la possibilità di lavorare sulle tecnologie trattate in questa tesi. Un immenso grazie anche alla Dr.ssa Silvia Sisinni e il Dr. Enrico Bravi che mi hanno dato un grande sostegno tecnico e mi hanno affiancato in questo lungo percorso.

Ringraziamenti ancora più sentiti per i miei familiari che mi hanno sostenuto in tutti i modi possibili in questo duro e lungo percorso, mi hanno rialzato nei momenti difficili, mi hanno spronato a non mollare e far emergere la migliore versione di me. Un grazie soprattutto a Michele, senza il quale oggi non avrei raggiunto questo traguardo.

Ad Alessandro, compagno di università ma soprattutto di vita, che tanto mi ha insegnato e con cui ho avuto la fortuna di condividere tutto in questi anni. Al Dr. Truffaut, carismatico e frenetico compagno d'avventure che con la sua spontaneità riesce sempre a strapparmi una risata. Un sentito ringraziamento anche a Emanuele, sempre col sorriso e la battuta pronta, il miglior complice di distrazioni e svago in università, in grado di alleggerire anche le infinite giornate al Politecnico. Un grande grazie a Riccardo, amico di una vita che nonostante la lontananza c'è sempre stato per me. Un pensiero anche a Zanca che non c'è più ma vive nei miei ricordi.

Contents

Chapter 1

Introduction

Until a few years ago, microservices architectures and the potential of virtualisation were niche technologies known only to a few, but today they have spread like wildfire in the IT environment, finding use in both large and small to medium-sized companies. In the former case, microservices are used to ensure greater isolation from application to application, greater flexibility (it is possible to create a virtual machine without having to buy a physical machine specifically), better scalability and, above all, consolidation: it allows the consumption of energy and hardware resources to be optimised by trying to saturate the capacity of each individual machine as much as possible. To take advantage of these technologies very often companies, especially small and medium-sized ones, turn to other companies that specifically provide their hardware and/or software resources, known as Cloud Service Providers (CSP). Small companies turn to these all the more so since, rather than making a large initial investment to buy machines and manage and maintain their IT infrastructure themselves, they decide to rent the infrastructure provided by the CSP at a significantly lower price.

However, this model presents critical issues from a security point of view. Inevitably, the company's data is managed within the CSP's machines, and as a result, the information leaks out of the customer's so-called trusted domain, thus becoming risky information. Security solutions can be implemented to try to curb this problem (e.g. firewalls), but it remains more difficult to protect the data against possible internal attacks, since control of the hardware and low-level software must still remain in the hands of the CSP.

There then arises a need on the part of the customer to receive greater guarantees concerning the operations performed on the machines provided by the CSP. In technical terms, the customer wants to be guaranteed verifiable computation, i.e. the property that ensures that the operation delegated to the CSP's machine is carried out correctly, maintaining computational integrity. The most popular solution today to guarantee that the computer's behaviour is as expected is trusted computing. To be able to make use of trusted computing systems, a suitable hardware component is required, providing Trusted Platform Module (TPM) or extensions such as Intel SGX for enclave technologies. Furthermore, this is based on EK (Endorsement Key) certificates created during the manufacture of the TPM, in which the asymmetric key generated in the TPM is not exportable. So given the context there must be an inherent trust in the hardware manufacturers.

Given the inherent architecture of TPM-based models and thus trust in hardware manufacturers, a trusted authority is strictly necessary to guarantee system integrity, thus not allowing for a decentralised architecture.

This work deals with a technology that makes it possible to generate a cryptographic "result" of the operations performed on a machine. Such a result can be generated independently of the presence of hardware with support for trusted computing. The final result of the thesis was made possible by developments in research in recent years in the field of Zero Knowledge Proof (ZKP) technologies and related protocols. Today, ZKPs are mostly used in the context of blockchain, where they allow the privacy of transactions to be guaranteed, making it possible to prove their validity without revealing the details. Another use concerns the aggregation of several transitions into a single proof, so that they can be verified in less time.

Specifically, the zk-STARK (Zero-Knowledge Scalable Transparent Argument of Knowledge) protocol is used to guarantee verifiable computation, confidentiality of the data processed during the app's execution, and the possibility for anyone in possession of the app to verify that it was executed correctly. More precisely, it is possible to check whether the app in question was actually executed, whether it was tampered with during execution and whether the output associated with it was tampered with.

VerComp, the framework developed in this thesis, addresses a critical gap in cloud computing security. While cloud services offer convenience and scalability, they present an inherent trust issue: clients must rely on CSPs to handle their applications and data accurately. Traditional security measures, including virtual machines and containers, fall short of providing guarantees against potential misuse by CSP operators. VerComp faces this challenge by enabling verifiable computation and ZKP in cloud environments. Its approach allows clients to confirm the integrity of their cloud-based operations without needing direct access to the underlying infrastructure. Although this system increases the computational load on the service provider's end, it significantly simplifies the verification process for clients. This trade-off is particularly advantageous, as it eliminates the need for clients to duplicate entire computations to ensure accuracy, thereby saving considerable time and resources.

Chapter 2

Zero Knowledge Proofs

Zero Knowledge Proof (ZKP) represents a cryptographic method between two parties: the Prover and the Verifier. The Prover aims to convince the Verifier of the validity of a specific statement without revealing any additional information. These proofs find several applications in blockchain technology, digital signatures, and identification protocols, with a strong focus on privacy guarantees and information confidentiality.

2.1 Historical background

It was 1985 when reference was first made to this technology by Goldwasser, Micali and Rackoff [1]. In the first part of the paper, they introduced a new method for communicating proofs, emphasising the efficiency. In the second part of the paper, they addressed the critical question of how much knowledge must be communicated to convincingly prove a theorem. It is here that a system, in which only the information strictly necessary to verify the truthfulness of the theorem is disclosed, is first referred to as a zero-knowledge proof system.

In 1987, Fiat and Shamir [2] defined a signature and identification protocol based on ZKP. The revolutionary aspect, that made this paper fundamental in the field of ZKPs, is the adoption of a cryptographic hash function, or more generally a random oracle, to simulate the random challenges in an interactive protocol, eliminating the need for direct communication between the prover and verifier. This approach, by reducing the data exchanged between prover and verifier, improved the practicality and scalability of zero-knowledge proofs, making them applicable in real-world cryptographic systems.

A few years later, another paper was published that turned out to be crucial in the field of succinct interactive proofs [3]. The proposed method ensures that each language in the polynomial-time hierarchy has an interactive proof system. As a result of this assumption, it is possible to reduce a claim over the sum of a multilinear polynomial's evaluations to a single evaluation at a randomly chosen point. This makes the verifier process more efficient.

Thanks to these publications, there have been substantial advances in ZKPs to date, especially dictated in recent years by blockchain privacy requirements.

2.2 Preliminary concepts

In this subchapter, the mathematical and other notions necessary for understanding ZKP systems are provided.

2.2.1 NP problems and language

A formal language L is composed of a sequence of words, which in turn are composed of letters. These words are well-formed and conform to a set of rules called formal grammar. If we denote by Σ^* all the possible finite strings that can be created from a finite alphabet Σ, then a language L over Σ is a subset of Σ^*, $L \subseteq \Sigma^*$. Any language can have finite, infinite or null cardinality.

A wide variety of computational problems can be distinguished, but the one dealt with in P and NP problems is that of decision problems. This category contains all those algorithms that, given a certain input, always return an output of boolean type.

P indicates the class of problems that can be solved in polynomial time. In other words, it consists of problems for which there exists an algorithm that can provide a "yes" or "no" answer in time bounded by a polynomial function of the problem's input size.

NP (Non-deterministic Polynomial time) problems are a class of decision problems for which a proposed solution can be verified as correct or incorrect in polynomial time by a deterministic Turing machine. This means that if a solution is given, it can be checked quickly, even if finding the solution may take an indeterminate amount of time. The class NP is crucial in computational theory and includes many important problems like the Travelling Salesman Problem, Knapsack Problem, and Boolean Satisfiability Problem (SAT).

In his foundation work, Stephen Cook [4] established the connection between these two classes by defining a specific type of relation.

Let's consider the binary relation $R \subseteq \Sigma^* \times \Sigma_1^*$ for Σ, Σ_1 all possible finite sequences from finite alphabets Σ^*, Σ_1^*. Let $\mathcal{L}_R$ be a language over $\Sigma \cup \Sigma_1$, defined as follows:

$$\mathcal{L}_R = \{w\#y | R(w, y)\}$$

The relation R is said to be polynomial-time if deciding whether a pair $w\#y$ belongs to $\mathcal{L}_R$ can be done in polynomial time, meaning $\mathcal{L}_R \in P$. $\mathcal{L}$ over Σ is in **NP** iff $\exists k \in \mathbb{N}$ and a polynomial-time checking relation $R : \forall w \in \Sigma^*$,

$$w \in \mathcal{L} \Leftrightarrow \exists y(|y| \leq |w|^k \text{ and } R(w, y))$$

where $|w|$ and $|y|$ represents the lengths of w and y.

2.2.2 Arithmetic circuit

An arithmetic circuit C over the field $\mathbb{F}$ and the set of variables $X = \{x_1, ..., x_n\}$ is a directed acyclic graph [5], allowed to either add or multiply, where:

- The vertices are called gates;
- Every gate that has no input wire can assume a value $k \in \mathbb{F}$ or a variable $x_i \in X$;
- The other gates can be labelled by either $\times$ or $+$ and have 2 input.

The arithmetic circuit is used in several ZKP protocols to represent polynomials. Figure 2.1 is an example of polynomial representation using an arithmetic circuit.

2.2.3 Turing machine

The Turing machine defines a theoretical model of computation, introduced by Alan Turing in 1936. It was created to help investigate what can and cannot be computed by a computational device. The Turing machine is fundamental in computer science, as it provides a precise definition of what it means for a function to be computable. It forms the basis for understanding concepts like algorithms, decidability, and complexity, and is essential for exploring advanced topics such as Zero-Knowledge Proofs (ZKPs).

According to De Mol [6] a Turing machine can assume a finite set of configurations $q_1, \ldots, q_n$. There are different variants of the Turing machine, the one under consideration has 3 tapes available:

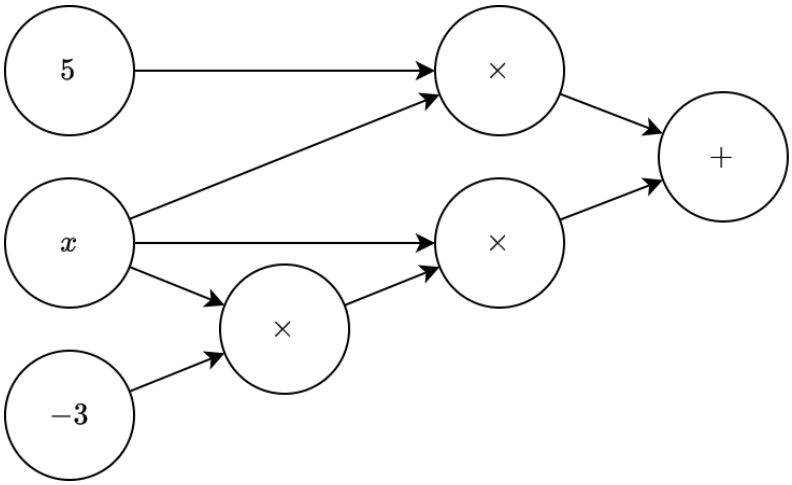

Figure 2.1. Arithmetic circuit of the polynomial $-3x^2 + 5x$.

- Read-only input tape;
- Work tape;
- Random tape.

In each variant, the way the tape is accessed (e.g.: sequential access, one-way etc.) may be different.

These characteristics make the Turing machine an automatic machine, which means that the behaviour of the machine is always determined by the current state and scanned symbol. The Turing machines that will be considered are:

- Polynomial time Turing machine: the amount of read and write operations done is upper bounded by a polynomial $S(n) = \mathcal{O}(n^k), k \in \mathbb{N}$ and n input size;
- Infinite power Turing machine: it has no computational limitations.

2.3 Interactive zero knowledge proof system

The interactive ZKP system includes the set of protocols in which the prover and verifier interact with each other by exchanging messages with the aim of verifying whether the prover's statement is true or false. The *zero knowledge* property is what makes interactive ZKP system different from interactive proof system. Many of these protocols iterate multiple times in the same process to achieve the desired level of security.

In order to explain the system properties, it is necessary to explain another actor: the simulator S. It is a polynomial time Turing machine capable of replicating the prover behaviour, so it can interact with the verifier in the same way the prover would. It also can do the rewind operation: whenever the verifier asks a question that it is not able to reply to, it rewinds the interaction to the previous state and continues from there. Potentially, after many rewards, the simulator can generate a proof that is accepted by the verifier. This means that the verifier cannot distinguish between the prover and a simulator but, more importantly, means that the protocol does not leak anything to the verifier [7].

The prover (P) is represented by an infinite power Turing machine, the verifier (V) is represented by a polynomial time Turing machine. In addition to the tapes mentioned in 2.2.3, (P, V) share two additional tapes:

- Tape PV: the prover reads and the verifier writes;

- Tape VP: the verifier reads and the prover writes.

Let $\mathcal{L} \subseteq \{0,1\}^*$ be a language known by the prover and verifier, ε a negligible value and (x, w) a proof statement composed of the public input and the witness respectively. (P, V) is an interactive ZKP system if it satisfies these properties:

- **Completeness**: the probability that a correct proof produced by an honest prover is considered as such is close to 1;

$$\forall (x,w) \in \mathcal{L}, Probability[\langle P(x,w), V(x)\rangle = 1] \geq 1 - \varepsilon$$

- **Soundness**: the probability that a verifier considers a false proof produced by a cheating prover P' as correct is close to 0;

$$\forall (x,w) \notin \mathcal{L}, Probability[\langle P'(x), V(x)\rangle = 1] \leq \varepsilon$$

- **Zero knowledge**: There is no distinguisher capable of distinguishing protocol execution between simulator and verifier and protocol execution between prover and verifier.

P' represents a cheating prover that only knows x. The distinguisher is a polynomial-time algorithm that compares the interactions of the real protocol (between P and V) and the simulated protocol (between S and V) [8]. As mentioned above, by definition the simulator does not know the witness. The property of zero knowledge states that the distinguisher cannot tell which of the two interactions is the real one, so the real protocol does not disclose any information about the witness.

Another type of soundness should also be mentioned, computational soundness. This gives the same guarantees as soundness but considers a polynomial time Turing machine prover.

2.3.1 Schnorr's protocol

Schnorr's identification protocol is a fundamental example of a zero-knowledge protocol [9]. The purpose of this protocol is to realise a process for making digital signatures so efficient that they can be executed by processors with limited computational power, i.e. smart cards. The protocol is based on the discrete logarithm problem in a finite cycle group.

Let x the secret value that the prover wants to prove it knows. More precisely, the prover proves it knows $h = g^x$, $x \in \mathbb{Z}_q = \{0, 1, 2, \ldots, q-1\}$, q large prime number and $h \in$ cyclic group $\mathbb{G}$ of prime order q. g is a generator of $\mathbb{G}$. The prime number q must be large enough to exploit the discrete logarithm problem.

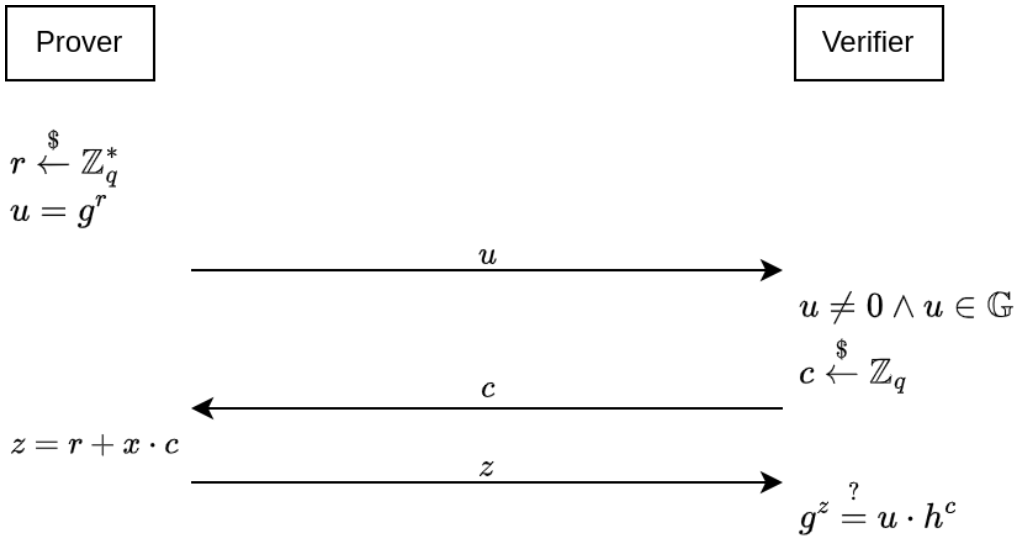

Figure 2.2. Schnorr's interactive identification protocol.

The interactive version of the protocol is shown in figure 2.2. The symbol "$\xleftarrow{\$}$", used to define r and c, indicates that the value is chosen randomly within the given set.

The operations done in this protocol can be split in three parts:

- **Commitment**: the prover, using a randomly generated number, commits to the verifier;
- **Challenge**: the verifier replies with a challenge, usually based on a randomly generated number;
- **Response**: the final computation based on the use of the values exchanged and calculated so far.

This common pattern describes a large category of protocols for proving statements and is called Σ-protocols [10].

2.4 Non-interactive zero knowledge proof system

The first ZKP protocols implemented were of the interactive type: it was necessary for the verifier to be present in order to carry out the interaction with the prover. The absence of the verifier means that the proof cannot be generated. However, this mode of proof generation is not compatible with the various scenarios in which it is used, including the blockchain environment or in general in contexts where the simultaneous presence of both entities cannot be guaranteed and again, when the number of interactions must be minimised in order to make it more efficient and practicable.

This need led to the development of techniques to transform interactive systems into non-interactive systems.

2.4.1 Fiat-Shamir heuristic

The Fiat-Shamir transformation is the most efficient construction of non-interactive zero-knowledge proofs. It transforms the 3-round protocol to a non-interactive ZKP protocol that only requires one round from the prover to the verifier. The only necessary assumption is that both the prover and verifier have access to a random oracle (RO), a pseudorandom function that takes an argument as input and returns a pre-specified length output. The random oracle function is of the form $RO : \{0,1\}^n \times \{0,1\}^a \rightarrow \{0,1\}^b$. The hash function is often used as RO. So the random oracle is a one-way function whereby, given distinct x_i, uniquely uniformly random outputs y_i are returned.

Let $c_0 = Hash(g, q, h, u)$ be the first computed challenge in a protocol that performs several iterations in order to achieve a sufficient level of security. For each iteration (after the initial one) the challenge value will be

$$c_i = Hash(c_{i-1})$$

That sequence makes it more difficult for a dishonest prover to forge the value of $c_0, ..., c_n$.

The figure 2.3 shows how Schnorr's protocol changes in the non-intearctive version, designated using the Fiat-Shamir transformation. The big change is therefore the computation of c. The more elements are concatenated in the hash calculation, the more unpredictable and unique the value of c becomes. This is why it is recommended to include all the public information necessary to compute the proof in c. Furthermore, the hash function, or more generally the random oracle, has the characteristic of not revealing any information.

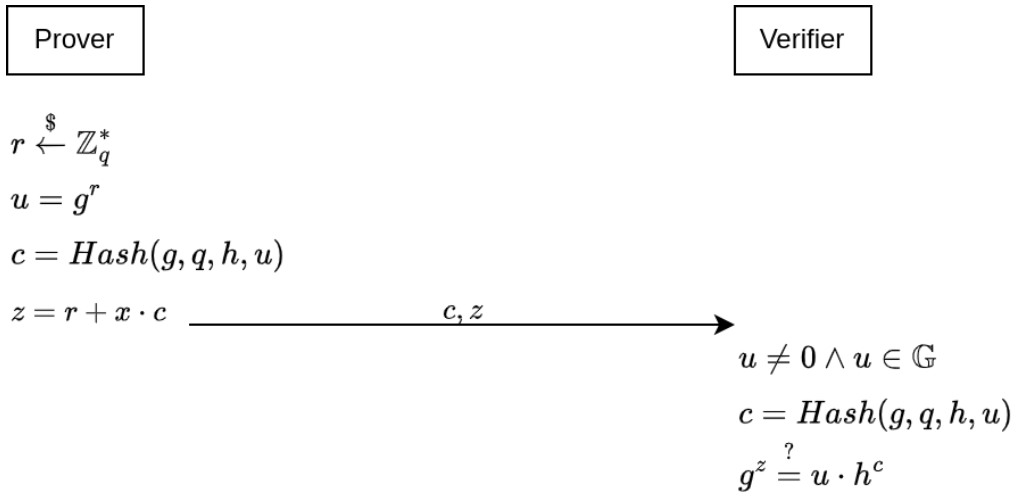

Figure 2.3. Schnorr's non-interactive identification protocol.

2.4.2 Common Reference String (CRS)

The CRS component serves as a pre-established, shared string of data used by both the prover and verifier to efficiently facilitate the secure non-interactive proof generation and verification. The string consists of homomorphic encodings [1] of specific multiples of $z \in \mathbf{F}_p$, where p is a large prime number. The aim of the prover is to demonstrate to the verifier the knowledge of the polynomial representing the problem on these values unknown to him, by exploiting the linear combination of the group elements of the CRS. Furthermore, the creation of these values prevents the verifier from having to generate them again each time to avoid replay attacks [11]. Finally, they make the protocol non-interactive.

2.5 Zero knowledge types

The simulator has been previously defined as a polynomial time Turing machine capable of communicating with the verifier without the verifier noticing. However, the indistinguishable nature of the simulator can have various shades that lead to the distinction of various types of zero knowledge.

In the *perfect zero knowledge* proof the simulated protocol is exactly equal to the actual protocol, so no information is leaked.

The *statistical zero knowledge* proof only discloses negligible information. More precisely, simulated and actual distributions are statistically close.

The weaker type is the *computational zero knowledge* proof, where the actual and simulated protocol are indistinguishable only in polynomial time.

2.6 zk-SNARK

zk-SNARK (*Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge*), like ZKP definition says, allows proving that something is true without revealing any other information by exchanging messages. It potentially has infinite possibilities for applications:

[1] An injective homomorphism $E : \mathbf{F}_p \to G$ such that it is hard to find x given $E(x)$.

- Prove statement on sensitive data: declare a financial situation or make anti-fraud checks while preserving privacy of the audited;
- Authorisation: acquire permissions to perform operations by means of privacy-aware authentication (no personal information disclosure);
- Outsourcing computational tasks: ask external entities to do complex computation and validate it rapidly, without redoing the computation.

In today's landscape, zk-SNARK is integral to enhancing privacy and scalability across various applications. It is massively used in blockchain technologies to create privacy-preserving cryptocurrencies, one among all Zcash [12] that permits the anonymous exchange of cryptocurrencies. Still in the area of cryptocurrencies, zk-SNARK is also present in Ethereum, giving the possibility of creating distributed application (smart contracts) with a focus on confidentiality.

It also allows good performance in terms of scalability, given the speed with which the verification process is performed. This speed is enhanced when it comes to complex computation where verification remains a fast operation, and also in contexts such as blockchains where multiple nodes need to verify the outcome of the same operation.

This is the meaning of the *SNARK* acronym [13]:

- Succinct: because the sizes of the messages are tiny compared to the length of the computation;
- Non-interactive: there is no or minimal interaction, only a setup phase and then just a message from the prover to the verifier;
- ARguments: the security is guaranteed against provers with limited computational power (computational soundness);
- of Knowledge: the only way the prover can generate a valid proof is by knowing the witness (the private data).

zk-SNARK can be used for any problem belonging to the NP class. However, instead of adapting the various NP problems to the zk-SNARK protocol, there is a tendency to make use of NP-complete problems. In addition to having the characteristics of NP-problems, these work as a "bridge" to other NP problems: an input to any problem in NP can be transformed to an equivalent input for a NP-complete problem.

Let $C \in \mathcal{NP}$-complete:

$$\forall p \in \mathcal{NP} \ \exists \ f \ \ s.t. \ \ p(x) = C(f(x))$$

where f is called *reduction function* and can be computed in polynomial time. The $\mathcal{NP}$-complete considered problem is Quadratic Arithmetic Programs (QAP).

According to the work of Gennaro, Gentry, Parno and Raykova [14], Q is a Quadratic Arithmetic Program over the field F, where $Q = \{V, W, Y, D\}$:

- $V = \{v_k(x) : k \in \{0, ..., m\}\}$
- $W = \{w_k(x) : k \in \{0, ..., m\}\}$
- $Y = \{y_k(x) : k \in \{0, ..., m\}\}$
- $D(x)$ divisor polynomial
- $f : F^n \to F^{n'}$

Q is a QAP that computes $f \Rightarrow a_1, ..., a_n, a_{m-n'+1}, ..., a_m \in F^{n+n'}$ is a valid assignment to the input/output variables of $f \Leftrightarrow \exists\ (a_{n+1}, ..., a_{m-n'}) \in F^{m-n-n'}$:

$$D(x) \ \Bigg| \ \left(v_0(x) + \sum_{k=1}^{m} a_k \cdot v_k(x)\right) \cdot \left(w_0(x) + \sum_{k=1}^{m} a_k \cdot w_k(x)\right) - \left(y_0(x) + \sum_{k=1}^{m} a_k \cdot y_k(x)\right)$$

Given a generic NP problem, the relation R_v on that problem is defined as the one that, through a verification algorithm V that takes the witness as input, returns the satisfiability of the NP problem as output.

2.6.1 Setup phase

The setup phase begins with the conversion of the NP relation R into the arithmetic circuit C. As anticipated above, the CRS is the set of elements from the finite cyclic groups in the elliptic curves. These act as homomorphic hidings of the random secret z. The elliptic curve is chosen because it offers the possibility of doing homomorphic encryption. It has to be chosen together with the *pairing function* e:

$$e(g^x, g^y) = e(g, g)^{xy}$$

As already mentioned in the section 2.4.2, these secret values must remain as such to ensure security. In fact, they must not be known to either the prover or the verifier. This is why they are called "toxic waste". Complying with this requirement in the generation of the CRS is fundamental, since the entire security of the protocol is based on it. It is therefore necessary for this phase to be carried out by reliable third parties who, once the string has been generated, destroy the secret values.

2.6.2 Rank-1 constraints system (R1CS)

R1CS defines a format for representing a polynomial. It is composed of four vectors $a, b, c, s \in \mathbb{F}_p^n$ where the equation to be satisfied is:

$$as \cdot bs - cs = 0$$

where s is the solution vector and as is the dot product. Given a vector s, the constraint system is satisfied if the result is 0. Before using this representation, it is necessary to "flatten" the polynomial. It is to be manipulated in order to obtain a system of equations, each with a form of the type:

$$x = y \ \texttt{OPERATOR}\ z$$

This is the form achieved using the arithmetic circuit, where each gate represents a simple equation that can only use the addition or multiplication operator [15]. By combining all the gates together, the R1CS representation becomes

$$As \cdot Bs - Cs = 0$$

where $A, B, C \in \mathbb{F}_p^{n \times m}, m$ number of gates. Once this form is achieved, it is possible to transform into R1CS.

Once obtained the R1CS, a further conversion must be done. QAPs extend the concept of R1CS by transforming the polynomial constraints into a form that is more friendly to zk-SNARK. Using the Lagrange interpolation the R1CS is transformed in QAP with 3 polynomials $L(x), R(x), Q(x)$, respectively the left input wires, the right wires and the output wires of all multiplication gates.

$$Q(x)v = L(x)v \times (R(x)v)\ , \quad v = [1, x^T, w^T]^T$$

where x is the public input and w the witness. The polynomials are the representation of the original circuit, and its constraints hold when the prover can generate $P(x) = L(x)R(x) - Q(x) = 0$ [16].

2.7 zk-STARK

Zero-Knowledge Scalable Transparent Arguments of Knowledge (zk-STARK) represent an advancement in cryptographic proofs, addressing the need for scalable, transparent, and post-quantum secure methods for verifying computational integrity [17]. zk-STARK offer a way to prove the correctness of computations without revealing the underlying data or requiring trust in a third party. Ben-Sasson, one of the zk-STARK authors [18], stated that:

> "Computational integrity means that the output of a certain computation is correct."

As the acronym says, the two properties used to describe this protocol are: *scalable* and *transparent*. The first one is the same as zk-SNARK. The second one refers to the lack of need for a trusted setup phase, which does not make it dependent on a trusted party for its execution.

At the root of it all is a computational problem (C) whose correct execution the prover wishes to prove. The verifier therefore wants to be certain that this computation (and the output associated with it) is carried out correctly, and that the prover has no possibility of falsifying the output. One precarious method the verifier could use to ascertain the correct execution is to run C again and compare the results. The zero knowledge property, on the other hand, takes over in cases where C must be executed on secret data S to which the verifier does not have access. In this, the verifier could not even re-execute C, but even worse, the prover could mask a lack of computational integrity by manipulating false data.

2.7.1 Execution trace and Low Degree Extension (LDE)

As a first step, the prover executes C. During execution, it writes the execution trace, a table where each column describes an algebraic register and each row the state of the computation for each instant of time. To make the explanation simpler from now on, only one register (one column) will be considered. The demonstration is however applicable to the case with multiple registers.

Let $S, S' \subset \mathbb{F} : |S'| > |S|$, $f : S \to \mathbb{F} \Rightarrow$ the LDE of f to S' is $f' : S' \to \mathbb{F}$ such that $\forall x \in S, f(x) = f'(x)$.

First, an interpolation algorithm is applied to the column, using additive Fast Fourier Transform (FTT), and then the LDE. The resulting function is called *trace polynomial*. At this point, the first commitment (c_1) is calculated as the Merkle Tree root, in which all column values are given as input (one commitment per column).

2.7.2 Constraints

The prover and verifier not only agree on the computation but also on the constraints: polynomial conditions which, applied to the execution trace, must be satisfied for the computational integrity. Let $g_1(x), ..., g_m(x), \forall i \in [1, m]$ $g_i : S' \to \mathbb{F}$ the constraints, then $\forall i \in [1, m]$ is defined $p_i(x)$ as rational function of g_i such that $g_i(k) = 0$ $\forall k \in S'$ where that constraint is applied. In other words, $p_i(x)$ is a rational function that only vanishes where the constraint is applied. This constraint manipulation makes it possible to demonstrate that the constraints are satisfied iff the prover knows a trace polynomial such that $p_1, ...p_m$ are polynomials. For convenience, a linear combination of these polynomials is defined as *composition polynomial*

$$CP(x) = \alpha_1 p_1(x) + ... + \alpha_m p_m(x)$$

Again, the constraints are satisfied iff $CP(x)$ is a polynomial and not a rational function. Another Merkle Tree commitment (c_2) on CP is done.

2.7.3 Fast Reed-Solomon Interactive (FRI) protocol

Up to this point, the only check the verifier can make with the information provided by the prover (i.e. the commitments) concerns the satisfiability of the constraints. This check is carried out by means of the commitment and decommitment (they will be explained later in section 2.7.4). At this point, it is necessary to use the FRI protocol to do low degree testing. This test serves to prevent a malicious prover from searching for a higher-degree composition polynomial that still satisfies the constraints for most x.

Let S a multiplicative subgroup of a finite field $\mathbb{F}$. The FRI protocol is used to prove that a given polynomial $CP : S \to \mathbb{F}$ has at most a given degree d. It works in such a way that, for each query, the degree of the polynomial is halved.

The FRI operator is applied at each iteration (round). For each ith-iteration, it takes a random value β_i to be executed. These are the steps:

1. The even and odd parts of $CP(x)$ are separated

$$CP(x) = g_0(x^2) + xh_0(x^2) \tag{2.1}$$

   where g_0 is the even part and h_0 the odd one.

2. β_1 is used to calculate a new function $CP_1 : S^2 \to \mathbb{F}$

$$CP_1(x) = g_0(x) + \beta_1 h_0(x), \quad x \in S^2$$

Finally, the commitment on $CP_1(x)$ is done.

This algorithm is repeated λ times until CP_λ is a constant. The prover sends CP_λ to the verifier. The complexity of the whole algorithm is $\mathcal{O}(logN)$ [19].

2.7.4 The proof

The verifier is currently in possession of the commitments provided by the prover:

- Commitment on the trace polynomial f';
- Commitment on the composition polynomial CP;
- Commitment on CP_1 obtained by applying the FRI operator to CP;
- ...
- Commitment on CP_λ obtained by applying the FRI operator to $CP_{\lambda-1}$.

These commitments are represented by the Merkle Tree root calculated on their evaluation.

On the basis of how the polynomial constraints $p_1, ..., p_m$ are defined, the verifier, in order to ensure that the calculation of the composition polynomial (which in turn is function of $p_1, ..., p_m$) is correct, needs the prover to provide the value of f' at some random points $x_1, ..., x_n$ strictly necessary for the calculation of the constraints. With this information, the verifier is able to calculate the composition polynomial at point $\overline{x}$ ($CP_0(\overline{x})$).

In reality, the prover does not only provide the calculation of f' at certain points. Along with this comes the *authentication path* of the Merkle tree. It consists of the leaf to which the value corresponds and the various hashes needed to allow the verifier to recalculate the Merkle tree and finally compare the autonomously calculated root with the one committed. This procedure is called *decommitment*.

The decommitment is done for $f'(x_1), ..., f'(x_n)$ and for $CP(\overline{x})$. From now on, each value that the prover provides is accompanied by the authentication path.

Based on the definition 2.1

$$CP_i(-x) = g(x^2) - xh(x^2) \Rightarrow \begin{cases} g(x^2) = \frac{CP_i(x)-CP_i(-x)}{2} \\ h(x^2) = \frac{CP_i(x)-CP_i(-x)}{2x} \end{cases} \Rightarrow CP_{i+1}(x^2) = g(x^2) + \beta h(x^2)$$

All these steps so show a result: $CP_{i+1}(x^2)$ can be computed by knowing $CP_i(x)$ and $CP_i(-x)$. Thanks to this the verifier can go on with the verification. It asks for $CP_0(\overline{x})$ and computes by itself $CP_1(\overline{x}^2)$. This is repeated until CP_λ and concludes the proof.

2.8 Use cases

The ZKP technology started to see the light with actually usable and efficient protocols around 2016. It is still an early stage technology. Nevertheless, it has been tested in business areas by corporates such as J.P. Morgan and Ernst & Young.

2.8.1 Privacy-preserving authentication

ZKP-based identification solutions allow users to prove their identity without sharing personal details. An individual can prove they are who they claim to be without sharing anything. The most effective example is that of adulthood: it can be used to verify that a customer is of a legal age without needing to see their birthdate.

Financial institutions can meet regulatory requirements by using ZKP to verify customer identities. This method ensures compliance and safeguards customer privacy by allowing customers to prove they meet the necessary criteria without sharing sensitive documents.

2.8.2 Decentralized finance

Common blockchains like Bitcoin or Ethereum do not implement properties of anonymity but rather pseudonymity. This is because they make public all transactions which, yes, are associated with addresses, but which can still be aggregated and traced back to the same person. To cope with this situation, ZKPs are introduced in many blockchains, firstly, in chronological terms, zCash but also Monero. The latter anonymises user balances and transactions (sender, recipient, amount).

2.8.3 E-voting protocols

There are currently a myriad of proposals promising voting systems based on ZKP technologies and often coupled with blockchain infrastructures. These systems apparently have many advantages. First, the ability to vote remotely. Thanks to the use of decentralised data structures (like blockchains), the risk of manipulation of election result is reduced. In addition, the integrity of the election process is guaranteed, confidentiality with regard to the anonymity of the vote and the possibility of verifying that one's vote has actually been counted.

Chapter 3

Verifiable Computation

With the increasing adoption of cloud computing, a crucial problem that arises is the potential untrustworthiness of data processing results. Users lose full control over their outsourced data stored outside their trust domains, raising concerns about the correctness of data processing and the possibility of their raw data being learned or exploited by a semi-trusted cloud service provider (CSP).

Traditional measures, such as firewalls and virtualization, are employed to safeguard user data security and privacy from external attackers. However, these mechanisms are insufficient against internal attackers within the CSP, as the provider controls the hardware and lower-level software.

Conventional encryption techniques fail to address this concern effectively because performing meaningful computations on encrypted data is challenging. Fully Homomorphic Encryption (FHE) offers a potential solution by allowing computations on encrypted data without decryption, but current schemes are impractical due to their high computational complexity [20].

This context underscores the need for research in verifiable computation to ensure the trustworthiness of cloud data processing. Verifiable computation enables clients to verify the correctness of the results returned by the cloud, thus enhancing the overall trust in cloud services (see Figure 3.1).

Cloud computing offers several advantages, including:

- no need to buy and maintain hardware;
- you pay what you use;
- scalability;
- location independent.

On the other hand, there might be no guarantees about data confidentiality, the correctness of the executed operations and the result might be returned incorrect on purpose. For example, a dishonest CSP could edit the operations to be executed for financial reasons. In addition to the cloud context, there are other areas of use in the real world that require verifiable computation [21].

Volunteer computing is a form of distributed computing where individuals contribute their computational resources, such as storage and processing power, to help compute small tasks for various projects. The process involves breaking down large computations into smaller units, distributing these units to volunteers for processing, and then aggregating the results in a simplified manner. The Great Internet Mersenne Prime Search (GIMPS) was the pioneering project that introduced the concept of volunteer computing. The Berkeley Open Infrastructure for Network Computing (BOINC) is a comprehensive middleware system for volunteer computing, encompassing a client, a graphical user interface for the client, an application runtime system, server software, and software for managing a project website. It supports a wide range of scientific

projects from the fields of astronomy, biomedicine, mathematics, climate change, physics, and chemistry.

The key factors that make a verifiable computation system successful, in order of relevance, are:

1. client's confidential information must not be leaked;
2. the client uses way less resources than that are needed to perform the computation;
3. SP cannot use extra resources than the actually required for performing the computation;

where the client is intended as the CSP client, which requires the execution of some operations.

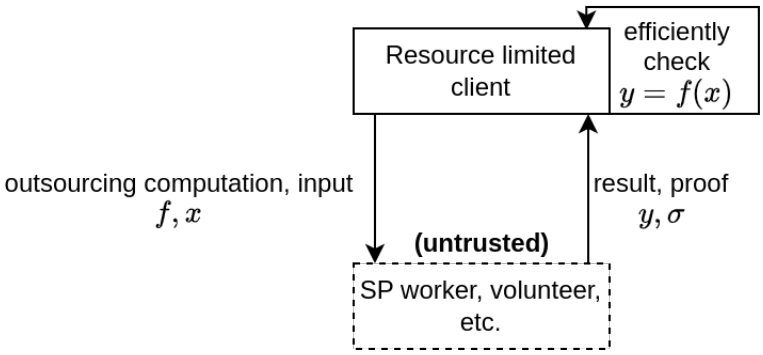

Figure 3.1. Verifiable computation scheme.

3.1 Assumptions based VC

Some of these solutions are focused on specially designed trusted hardware, while others presented audit-based solutions for ensuring security and verifying the correctness of computations. Additionally, other proposals suggested using assumptions of replication or attestation to achieve these goals.

Although secure coprocessors have been proposed for commercial use, these are limited in terms of tamper resistance. Indeed today *trusted platform modules* (TPMs) are the standard for secure crypto-processor. They offer secure storage and cryptographic key generation. Although TPMs are available commercially at reduced costs, they present certain issues related to client privacy and offer limited physical tamper resistance. Furthermore, TPMs do not ensure the integrity and confidentiality of the physical memory space, which are critical concerns in verifiable computation. Trusted hardware based solutions require high design and maintenance cost.

Other solutions make use of *attestation mechanisms*. Pioneer [22] is one of them. It is a software based attestation system that can report failures at run-time using a challenge-response model. As a drawback, it is required to the verifier to know the prover hardware configuration.

Another field of research focuses more on *replication* of computation by several servers. These kinds of solutions put more emphasis on data consistency and availability, in fact they guarantee the execution of operations even in case of failures (up to a threshold). However, this method completely disregards the privacy aspect of the client.

3.2 Proofs based approaches

In this context, proofs are used, based on cryptographic and mathematical protocols, which allow efficient verification of the correctness of the computation. In this way, the verifier is convinced

by the proof and without the need to perform the computation again. As noted by Ahmad et al. [21], there are different types of proof based approaches.

Interactive proof system (IP) takes into account an untrusted exponential time prover and polynomial time verifier. It has been formalised for the first time by Goldwasser [1]. The properties of these kinds of systems are:

- Completeness: the verifier accepts the correct statements returned by the prover.
- Soundness: the verifier can be convinced with a very small probability about an incorrect statement returned by the prover.

Probabilistic checkable proof system takes into account a verifier that runs a polynomial-time randomised algorithm with restrictions $r(n)$ on the amount of randomness it can use and on the maximum number of bits $q(n)$ it can read from the proof. Given an input x and a membership σ, the verifier must accept correct proofs and reject incorrect ones with very high probability. The properties are:

- Completeness: the verifier accepts every random string of correct proof with probability 1.
- Soundness: the verifier rejects all the random strings of incorrect proof with probability more than 1/2.

Computationally sound proof system aims to efficient verifiability, provability and recursive universality. Differently from other systems, in computationally sound proof systems, it is straightforward to compute a proof for a true statement, but difficult or nearly impossible to compute a proof for a false one.

Zero knowledge proof system takes into account a prover that tries to convince a verifier about a statement without disclosing any related information.

3.3 Formalisation

Verifiable computation enables a computational weak client to outsource its operations to any untrusted party. The outsourced operation is represented by f that is computed on some input $x_1, ..., x_n$ dynamically provided to the powerful but untrusted worker. Once it finishes the computation, it returns back the result y and the proof σ (as shown in Figure 3.1).

Gennaro et al. formalised the verifiable computation scheme in 2010 [23].

The first phase is *preprocessing*: the client computes public and private auxiliary information associated with f. It can take as long as computing f itself, but since it is done only once, its cost, in terms of time, is amortised. Then the *input preparation* takes place: it consists in computing auxiliary information about the input x. Again, information is in the public and private domain. Public information is sent to the worker. Once the worker finished computing f, it returns π_x to the client. π_x is a string that encodes the result and the necessary information later used by the client to verify the correctness.

The verifiable computation scheme can be summarised by 4 algorithms [23]: $VC = (KeyGen, ProbGen, Compute, Verify)$.

1. $KeyGen(f, \lambda) \rightarrow (P_k, S_k)$: according to the security parameter λ the key generation algorithm returns a keypair where the public one is used by the worker to compute F. S_k is kept private by the client.
2. $ProbGen_{S_k}(x) \rightarrow (\sigma_x, \tau_x)$: it returns the above mentioned auxiliary information about the input x. σ_x is encoded using the S_k and is meant as public value. τ_x is kept private by the client.

3. $Compute_{P_k}(\sigma_x) \rightarrow \sigma_y$: using the encoded input the worker computes the encoded output $\sigma_y \leftarrow y = f(x)$.
4. $Verify_{S_k}(\tau_x, \sigma_y) \rightarrow (y, \perp)$: this algorithm uses the private input auxiliary information and the encoded output to calculate y, otherwise returns $\perp$ when σ_y does not represent a valid output of $f(x)$.

3.4 VC frameworks

This section describes three protocols based on zero knowledge proof that allow verifiable computation.

3.4.1 ZEKRA

ZEKRA, *Zero Knowledge Control Flow Graph*, is a novel approach to control flow graph (CFG) attestation [24]. ZEKRA is the first privacy-preserving control flow attestation protocol and involves 3 actors (see figure 3.2. The *prover* is considered untrusted and underpowered. It must only be capable of tracing the program execution and authenticating it. The *verifier* is another untrusted device that only requires the execution of a program. It does not necessarily have the appropriate hardware characteristics (i.e.: IoT device). For this reason, a semi-untrusted *worker* is also employed. The worker is in charge of generating the proof on behalf of the prover of the computation requested by the verifier. It uses zk-SNARK thanks to which it is possible to hide inputs, such as attested execution path and program details, from an untrusted verifier. The *execution path* is the ordered set of memory addresses that are taken during the execution of the program. ZEKRA contributes to transforming the verification of the attested execution path into a verifiable computational task that can be delegated to an untrusted worker.

The control flow attestation is mainly efficient against runtime attacks. Among possible runtime attacks, ZEKRA counteracts those of the control and data types. The *control-based* attacks aim to manipulate the control flow of a program during its execution. These include code injection and code reuse. The first one consists of diverting the execution path to external malicious code. The second one "recycles" benign pieces of code (often called gadgets) that end with return or indirect jump instructions. The *non-control-data* attacks aim to leak sensitive memory area that are not directly used in control transfer instructions. Attacks have been discovered that are capable of escalating program privileges to root through the only modification of a variable [25].

As mentioned before the nodes taken during the execution are stored as memory addresses. However, representing the CFG using the memory addresses is not an efficient solution in terms of lookups. This is why a mapping has been adopted. The mapping is used to translate the list of possible memory addresses to indexes. These labels are stored in a *IndexedBitArrayEdges*, structure suggested when consecutivity between neighbours of vertex is present [26]. In this specific case it exploits the concentration of edges in specific area of the adjacency matrix ($2 \times N \times N$ matrix where $M(n_i)(n_j) = 1$ means the existence of an edge from n_i to n_j).

The proof generated by the worker states that it successfully verified that the execution path taken by the prover belongs to a legal control flow graph referred to by its image $h = hash(CFG)$. The verifier sends the execution request by the program reference and a nonce (for freshness reason), respectively, $@P$ and r_1. The prover executes the program P and the trusted traces contemporary. It elaborates the execution path εP that is hashed with the verifier nonce and signed with the prover secret key tsk. At the end the worker converts the execution path representation in labels and then generates the proof. The worker sends the proof, public inputs y and prover signed message to the verifier. y is composed of the digest of CFG, the mapping, the execution path, the entry and exit nodes and r_1.

3.4.2 Circom

Circom is a framework that allows developers to design arithmetic circuits at a constraint level. The language offers a clear and intuitive syntax, making it more accessible to define complex

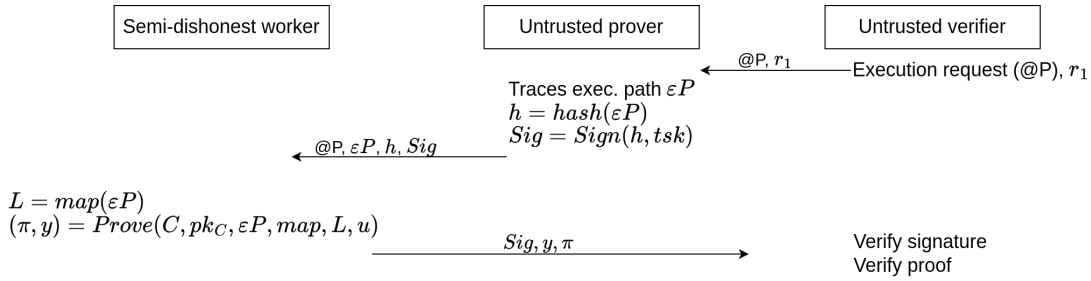

Figure 3.2. Simplified representation of the ZEKRA protocol.

circuits. The primary output of Circom is a file that describes the circuit in the Rank-1 Constraint System (R1CS) format (see Section 2.6.2). Circom defines a language with syntax specific to circuit construction and provides a compiler. The generated circuit can be then used with zk-SNARK to generate and validate zero knowledge proofs associated to the circuit.

The *template* in Circom is the structure corresponding to the circuits. They are characterised by the input and output signals and describe the relationship between them. Templates allow the realisation of more or less complex circuits thanks to their modularity: it is possible to define small circuits and compose them together to form a more complex one. The Circom compiler gives the possibility to compile the program in both C++ and WebAssembly. Additionally the compiler offers the possibility to generate the binary representation of the R1CS. This second option is given so that the ZKPs are brought to the web. Regardless of which two programs is used, it, by providing the input values, calculates the remaining circuit values. In case the input, intermediate and output values are valid, they are called *witness*. A scheme of Circom's operation is shown in figure 3.3. Circom being a mature ecosystem, it also offers the possibility of debugging through the use of a logging function that can be of great help in debugging circuits. Circom also

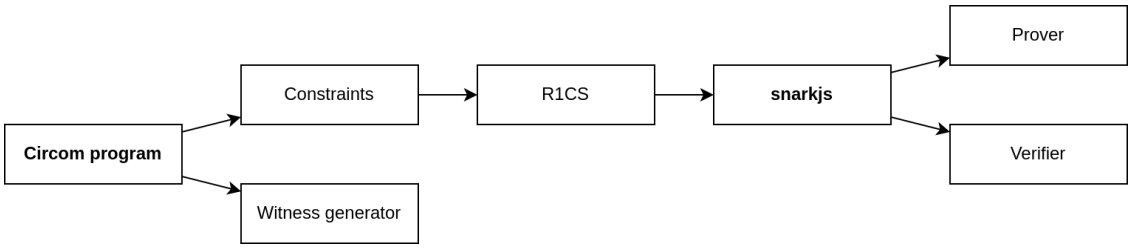

Figure 3.3. Circom compilation scheme.

provides a wide variety of ready to use templates of all kinds called *circomlib*. As said before, they can be used to build larger circuits. The available templates are for simple operations like logic gates, comparators and multiplexers but also for more complex computations like digital signatures and hash functions.

Among the possible applications of Circom there is hashing. Thanks to all the available hashing functions in the circomlib, it possible to construct a template that generates a proof that the prover knows a given value from which the public output hash has been computed. The drawback of this implementation relates to the fact that common hash functions such as `sha256` are computationally complex, and this one in particular is described in Circom by means of 29450 constraints. For this reason, there are templates in circomlib for other hash functions that are more suitable for arithmetic circuits (e.g., Poseidon and Pedersen hash).

It is also easy to define a template for the generation of a public key generated on an elliptic curve. Once the two parties agree on the generator (G), through a suitably structured short template, it is possible to generate the secret key (S_k) as a random scalar and the public key as $S_k \times G$.

3.4.3 RISC Zero

RISC Zero provides an open source virtual machine with a zero knowledge proof system that, used together, gives the possibility of guaranteeing computational integrity (Section 2.7) of a binary. The ZKP system generates a zk-STARK proof which, together with other information, is called receipt that is paired with the program output. Any binary compiled for RISC-V can be executed within the zkVM (*zero knowledge Virtual Machine*), so even untrusted third parties can verify if the computational integrity is respected.

This is the framework chosen for the implementation of the orchestrator, and is therefore explored in more detail in the chapter 4.

Chapter 4

RISC Zero

The zkVM is the core component that makes RISC Zero a remarkable technology. It enables developers to take advantage of an environment where it is no longer needed to code a circuit but an app. This brings with it the possibility of exploiting the Rust ecosystem of libraries, without worrying about rewriting them.

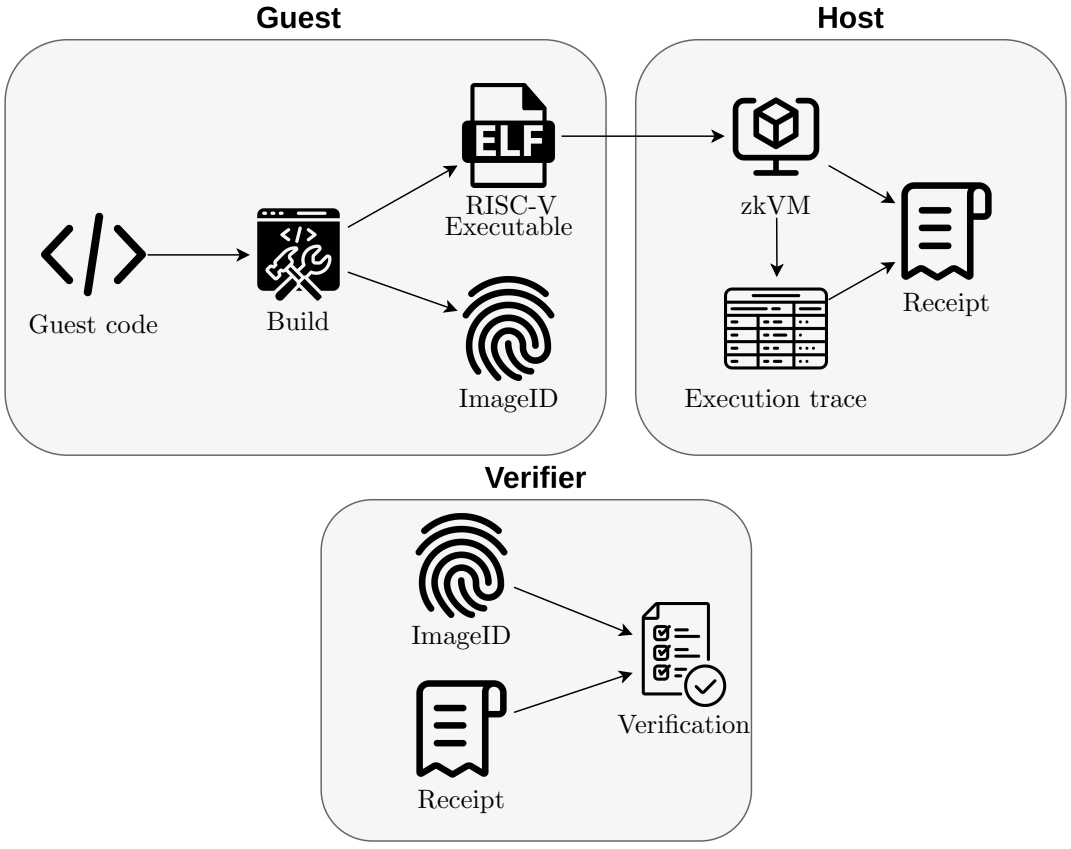

Figure 4.1. RISC Zero architecture.

4.1 RISC-V

Before proceeding with the zkVM, it is necessary to mention the RISC-V Instruction Set Architecture (ISA). First of all, the name is an acronym standing for "Reduced Instruction Set

Computer". The "V" indicates that this ISA is the product of the fifth edition of UC Berkeley's RISC architecture. Like its predecessors, this architecture is open, so anyone can inspect and modify it as they wish. This key feature ties into the primary goal of the architecture: making the ISA usable across as many devices as possible. It must therefore be implemented in such a way that it does not reference or rely on pre-existing microarchitectural patterns [27].

Being an open and free standard, consumers are more incentivised to adopt this ISA rather than a paid one, whose actual implementation is not fully known. Finally, it is ideal for those who, distrusting closed-standard implementations and fearing industrial or governmental espionage, choose to rely on transparent and modifiable standards.

As for the innovations in this latest version, they include:

- Support for 32-bit and 64-bit address spaces.
- Support for variable-length instruction set extensions.
- Hardware support for C11 and C++11.
- Separation between the ISA and optional extensions, aimed at streamlining the base ISA while simultaneously improving performance.

The RISC-V version used in RISC Zero is `rv32im`. As mentioned earlier, in RISC-V, there is a distinction between the base version of the ISA and the extensions. In this case, the base ISA is `rv32i` and `M` is the extension.

`rv32i` is a base integer instruction set and, as such, is designed to support operations on 32-bit integer numbers. It provides 40 unique instructions that enable arithmetic, logical, control flow operations, and data movement between memory and registers. The "M" extension implements multiplication and division operations between two integers in registers [28].

4.2 zkVM

The zkVM is a software emulator that implements `rv32im` and enables the generation of a zk-STARK proof for the program executed within it.

Both in the zkVM and in the physical CPU, the clock cycle is used as the smallest unit of time. It is marked by the tick of the CPU's internal clock and, in this architecture, represents the time required to perform a basic operation. Indeed, as in many architectures, not all operations have the same cost. Logically, an add instruction requires fewer cycles compared to a div instruction. According to the RISC Zero documentation [29], the peculiar aspect of the zkVM is that div takes twice as long as add. In physical CPUs, however, the div instruction takes between 15 to 40 times [30] as long as the add instruction.

Regarding the actual execution of the program, it is single-threaded and does not differentiate between privileged and user modes. Instructions are always executed in order and are never re-ordered by the zkVM.

4.3 Concepts

4.3.1 ImageID

The verifier is able to check whether the computation has been altered in some way by only having the receipt (see 4.3.2) and the app identifier, that is known as *ImageID*. This identifier has been defined to face the need to uniquely refer to an app without the need of having it. Doing so, the verifier can check without having the app stored in local but the ImageID, which is the root hash given by the Merkle tree of the memory right after the binary is loaded. Thanks to the way it is defined, the ImageID ensures that the identifier changes together with the guest code, even for minor changes. SHA-256 is used as hash function.

4.3.2 Receipt

Once the execution is completed in the zkVM, it is not produced a common zk-STARK proof but a *receipt*. This structure is RISC Zero specific (Figure 4.2), and it is useful for a variety of reason. The *claim* contains the public data. The most relevant information is the journal: the

Figure 4.2. Receipt structure.

developer of the guest code can decide whether and which variables to make public during the execution. They all are then appended to the journal. The claim also includes information about the program exit (in order to understand if it terminated correctly or not), the memory state at the end of the execution and the ImageID too.

The *seal* is the actual zk-STARK proof. Thanks to the receipt, it not only binds to the app execution but also to the claim information. In fact, if the claim data is changed, it can be found out. At the same time, if the zk-STARK proof is changed or substituted with another proof, it is detected because of the information in claim section (i.e.: ImageID).

4.4 Roles

In the RISC Zero architecture there are three roles:

- Guest
- Host
- Verifier

The guest knows the source code (also called guest code) and thus also the input and output of the app. Only those in possession of the guest code (i.e. the guest) are able to generate the ImageID. The executable produced from the guest code is executed on a machine called the host, which handles the input values given to the zkVM and the output values returned from it. The host machine, in the original RISC Zero architecture, is considered untrusted. Even if it has malicious intentions, the host can only read and write the zkVM I/O, but it can't tamper the ZK app execution inside the zkVM without being detected. In case the zkVM runs in a trusted domain, the generated proof, even if computed using private data, does not disclose any of it.

In the end the verifier (considered untrusted as the host) is anyone who knows the ImageID and has a receipt. In RISC Zero the verification in some contexts can be very convenient as multiple proofs can be aggregated into a single proof. Thus the verification of a single proof guarantees the computational integrity of multiple ZK apps.

4.5 Workflow

In this section the architecture of the framework, its components, and how they interact with each other are introduced. Figure 4.1 provides a summary diagram of the components and how they are interconnected.

First the guest code is written and, along with it, the inputs and outputs received, their types and, if desired, the variable to commit are defined. Outside these operations the application can be written with Rust, by also relying on the wide offer of available libraries (as today more than 70% are natively compatible with RISC Zero). The guest code is now ready to be compiled. As the framework suggests, the binary is compiled in order to be executed later in the zkVM that indeed adopts an ISA inspired to RISC-V rv32im specification. This architecture also allows developers to write code in other RISC-V compatible languages, such as Go and C.

At this point, the host relieves. For it to be able to perform its tasks, it must have at its disposal:

- ELF file;
- input (also the type of it).

The host instantiates the *Executor*, the component in charge of generating the execution trace (mentioned on Section 2.7.1), set the maximum number of cycles, set environment variables, arguments and writing to the standard input to communicate to the guest app. The executor collects the data representing a snapshot of the full state of the machine at each clock cycle. The transition from execution trace to receipt is carried out in a manner similar to that of zk-STARK. The peculiarity is that in this case the execution trace consists of:

- data columns comprising the processor status, ISA registers, the program counter, ALU registers and other microarchitecture details;
- control columns are responsible for managing system initialisation and termination processes, establishing the page table with the program's memory image, and orchestrating various control signals that operate independently of the program's execution;
- accumulator columns for emulating RISC-V memory.

With this data at hand, the procedure is the same as for zk-STARK: padding of the execution trace is done, trace polynomials, constraints polynomials and FRI protocol are generated (see Section 2.7).

The next zkVM component is the *Prover* that transforms the execution trace into a receipt.

Once the receipt has been created and distributed, anyone can check its validity by just knowing the ImageID. The verification process is the same as that of zk-STARK which is explained in the 2.7.4 section. The only differences are that in this case there is no continuous communication but the protocol is non-interactive: all the information the verifier needs is contained in the receipt. Furthermore, the proof not only attests the correct execution of the program, but also links to the public output, the journal. The proof is false if the journal is tampered with.

4.6 Compliant application: password checker

An example of a possible application of RISC Zero is the password checker. The operation of the program is as follows: given as input a string and an array of bytes, password and salt respectively, the validity of the password is checked (see Figure 4.3). The criteria by which this password is considered valid or invalid are specified in a constant called `PasswordPolicy`, in which parameters such as `min_length`, `max_length`, `min_lowercase` and others are specified. In this specific case, the policies to be observed are those indicated in the snipped 4.1, which can of course be completely changed and managed as required. If the parameters are not met, the application panics, otherwise it proceeds with the generation of the hash (using the salt provided) and finally commits the password hash to the journal, so that the proof is bound to the public data (hash).

```
const POLICY: PasswordPolicy = PasswordPolicy {
  min_length: 7,
  max_length: 64,
  min_numeric: 2,
  min_uppercase: 2,
  min_lowercase: 2,
  min_special_chars: 1,
};
```

Listing 4.1. Possible implementation of password policy.

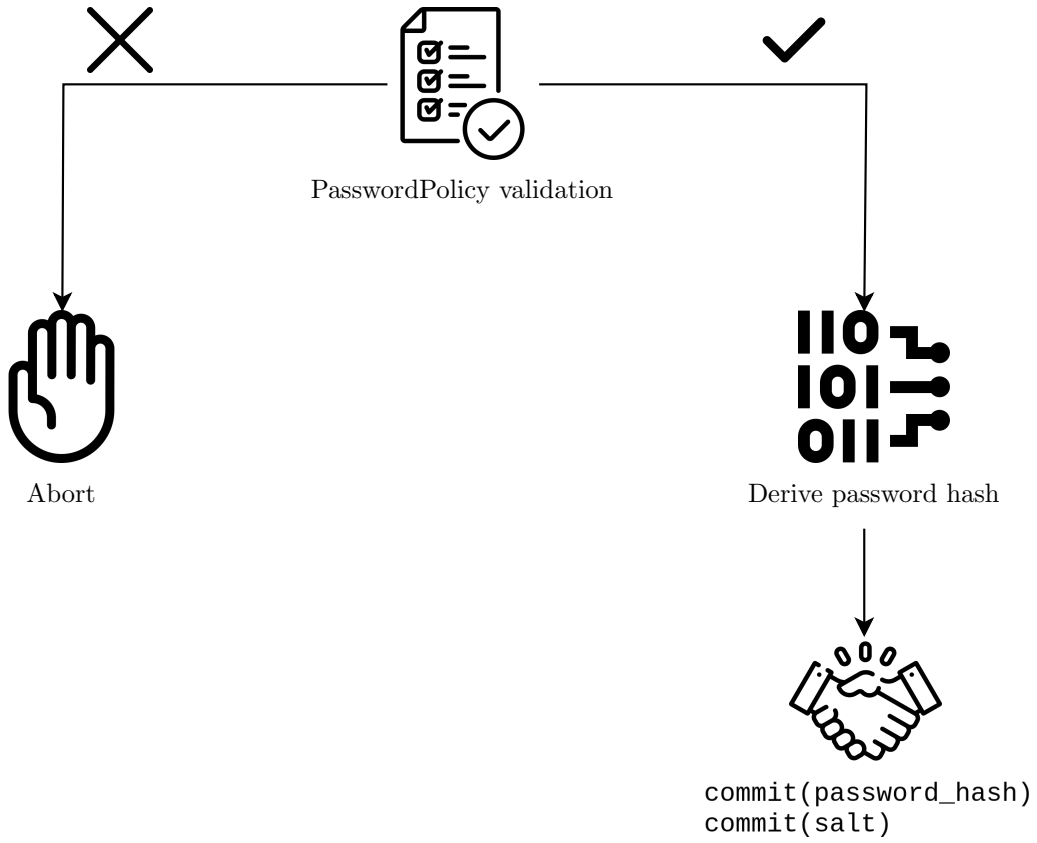

Figure 4.3. Password checker algorithm.

This specific application is useful because it allows plain password checks to be carried out locally, thus avoiding having to do them on a web page. It could be said that doing so reduces the *possible shared information* and thus reduces the surface attack.

4.7 Cryptographic security

In the RISC Zero documentation, cryptographic security is also discussed. More specifically, two potential threats are considered: the difficulty for a malicious user to generate a valid proof without knowing a valid execution trace, and the difficulty for a malicious user to extract information from a proof. Assuming the presence of a Random Oracle Model, the RISC-V Prover achieves 98-bit security while maintaining quantum-safe property.

Further studies have also been conducted on the STARK Prover (that is used in the RISC Zero prover). The problem analysed is that of collision brute force against the STARK protocol. It was assumed that an attacker has access to 1 million high-performance GPUs (RTX 4090). According to the benchmark published on GitHub [31], this graphics card has a hash rate of less

than 25 billion hashes per second. Based on the following calculation, it would take 400,000 years to brute force a collision:

$$\frac{2^{98}}{25{,}000{,}000{,}000\,\text{Hashes/s}\ \cdot 1{,}000{,}000\,\text{GPUs}} = 1.27 \cdot 10^{13}\,\text{seconds} = 401{,}969\,\text{years} \tag{4.1}$$

Chapter 5

VerComp Design

The designed application aims to provide an environment for the execution of software with verifiable computation and ZKP included. The environment under consideration consists of a client and a server. One or more clients can share an app with the server and request its execution. The server will respond with a receipt demonstrating the actual execution of the software. In this way, after the execution, the client is assured that its app has been executed correctly, in its entirety.

5.1 Purpose and motivation

The reasons why VerComp was created are many and will be discussed in this section. When a customer decides to take advantage of cloud services such as the use of instances in remote servers, he must also be aware that, despite all the possible guarantees given by the CSP (Cloud Service Provider), the customer can never be certain that what he asks to be executed, will be executed without a hitch (as explained in chapter 3). VerComp provides more guarantees in this context.

This tool allows customers to delegate computationally intensive tasks to untrusted cloud service providers, while guaranteeing the correctness of the results. It ensures that computations are performed correctly, and that results can be verified by the customer, increasing security and trust in the cloud environment. In addition, the decision to use VerComp in the cloud could be advantageous for small and medium-sized companies in terms of cost. Delegating computationally intensive tasks to a CSP can reduce the costs of maintaining and scaling their infrastructure. So far, computationally intensive operations have been mentioned because the involvement of ZKP and verifiable computation techniques (hence the use of RISC Zero) is expensive in terms of required resources.

This solution guarantees verifiable computation with the addition of ZKP. This additional feature ensures that the proof does not disclose any information relating to the computation (except intentionally, as explained in the section 4.3.2). In this way, the receipt may be inspected first and foremost by the client, but also by third parties, without them learning any private information. Moreover, in the event of unauthorised disclosure of this data, due to the characteristics just mentioned, the attacker would not be able to steal information.

If on the generation side, proof represents a great effort in terms of computation, on the other hand, at the verification side, it can benefit from the possibility of performing verification in a very short time. It thus becomes fast and convenient to verify the proof and thus the correct execution of applications that on the client side could not even be executed without the appropriate hardware. In conclusion, as this is an innovative area and still under development, VerComp could solve existing problems or create new opportunities. Since the exchange of receipts between untrusted third-parties is possible, there is the possibility of interaction even in the absence of trust (*trustless* interaction), creating new scenarios and use cases in which verifiable computation and ZKP can come to the rescue, especially in cases where very sensitive data such as health, finance and digital governance are involved.

5.2 Actors and protocol

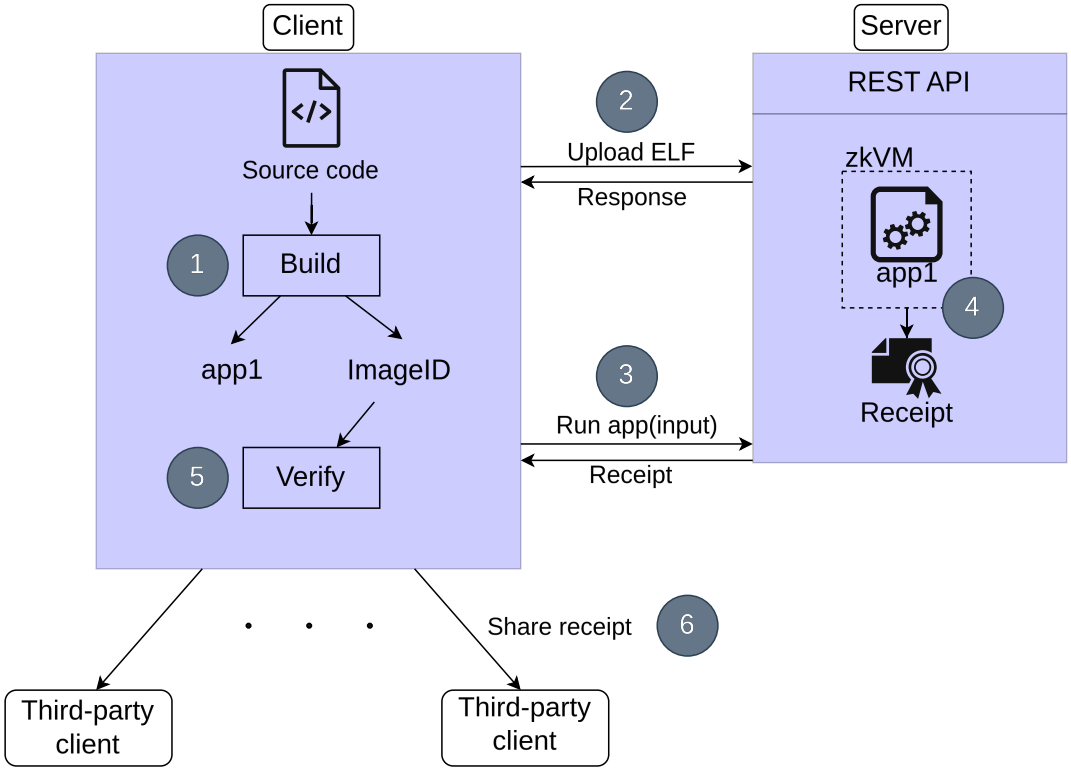

Figure 5.1. Receipt generation process actors.

As depicted in Figure 5.1, there are three actors:

- **Client**: it owns the source code of the application to be executed in the zkVM and to be proved. It is possible to generate the ImageID from the source code, so only he is able to generate it.
- **Server**: the machine that handles the execution requests and instantiates the VMs on which the applications are executed. It uses HTTP REST API requests to communicate with the client.
- **Third-party client**: this is the only *optional actor*. Based on the kind of application, they can be part of the process. They might be interested in making sure the application has been correctly executed. Again, based on the scenario, they might already be in possession of the ImageID or they might ask it to client.

Proceeding in order, the data involved in this process are as follows:

- Source code: This is written by the client using a programming language compatible with the RISC-V architecture (in VerComp is used Rust). It is also possible to distinguish within the code the parts that require verification from those that do not. Additionally, the application must be developed using the RISC Zero library, which allows integration with the zkVM, as it provides methods for reading inputs (which are passed by the host at runtime) and appending public outputs to the journal.
- The application (as ELF file), and the ImageID (discussed in section 4.3.1 are generated as a result of the build process.
- Input values: referring to Figure 5.1, at point (3), input values are passed to the server. These are written in JSON format, for a reason that will be specified later (Section 6.1).

- Receipt (discussed in section 4.3.2), serialised and forwarded to the client, who writes it into a file. For simplicity, although there are no constraints on this, the receipt is usually saved in a file, with the extension `.risc0`.

The Figure 5.1 also describes the interactions between the actors and shows when the above-mentioned data are used. Initially, the client, in possession of the source code, (1) performs the build process, resulting in the ELF binary file (app1) and the ImageID, obtained from the Merkle tree root calculated on the state of the virtual machine after the executable has been loaded into memory (represented by a SHA-256). Subsequently, using the APIs provided by the server, (2) the client requests the server to upload the executable so that it can be executed at a later time. If successful, (3) the server will respond with an identifier that the client can use later to reference the uploaded executable. The client then makes a second request to the server and, if necessary, also attaches the input values. At this point, (4) the server instantiates a zkVM, loads the requested binary file, and executes it using the input provided by the client. Upon completion of the execution, a receipt will be generated and returned to the client. Finally, (5) the client can verify that the application was indeed executed by the server and, if needed, examine the additional public information (journal). Optionally, as a final step, the client may (6) share this receipt with third parties interested in the outcome of the execution. These parties will only need the ImageID and the receipt to verify the result.

5.3 Additional components

Overall, two additional components are utilised to complete the process. The first is employed during the build phase. As explained in Section 4.3.1, the ImageID is represented by an SHA-256 hash, which depends on the binary file loaded into memory. Since compilation with Cargo is typically non-deterministic, the developers of RISC Zero have defined a method to make the build phase reproducible by using Docker.

The other additional component is used during the verification phase. It consists of a CLI interface that includes a command which, when provided with the ImageID and receipt, swiftly verifies the correctness of the execution. It was also necessary to develop this software in order to allow the verifier to control the receipt easily with a simple terminal command.

5.4 Software design

The server provides routes that can be reached via HTTP requests. Recalling what was said in section 4.4, the host has the task of creating and managing the zkVM. Again, the VerComp host does the same job as the RISC Zero host in addition to handling the requests received. So once it receives a request to execute a certain program, using the RISC Zero library, it creates the zkVM, passes it the input and starts execution. On the other hand, when it is asked to save a program so that it can be executed later, a simple data copy operation is performed from the HTTP request body to the server's file system.

Figure 5.2 shows the sequence diagram that describes the entire VerComp process.

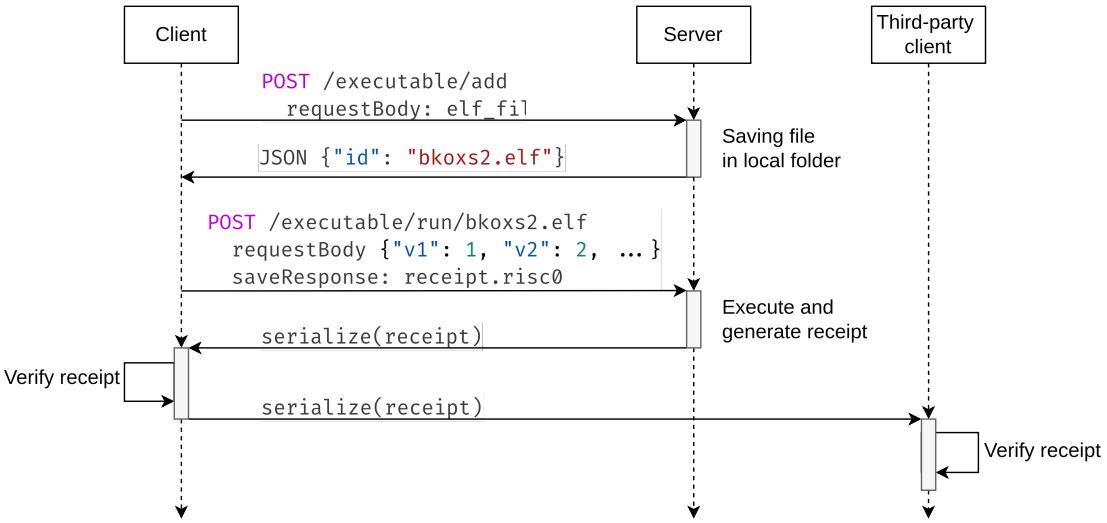

Figure 5.2. VerComp sequence diagram.

The client instead only needs to send HTTP requests with a more or less automated tool of his liking.

5.5 Security reflections

Before proceeding further, it is necessary to provide a specification regarding the server. It is firstly assumed that the server is accessible upon an authentication layer (e.g.: via site-to-site VPN).

Furthermore, due to the nature of the protocol, the server must receive input from the client in order to forward it to the application running within the zkVM. This implies that the server has the ability to read the data in clear text. For this reason, to maintain the integrity and confidentiality of this sensitive information, Homomorphic Encryption techniques should be used. However, due to the early stage of Homomorphic Encryption, it is necessary to wait for the advancement of study and research in this area in order to implement it in real scenarios.

5.5.1 Possible attack vectors

A malicious attacker could try to act and tamper with the system in various ways. Let us assume that the attacker has taken control of the host machine. In this way, he could intercept and observe all executables at his disposal (saved locally) and intercept all requests received from the client. In this case, the attacker could only observe the execution of the program, always considering that it is executed within a virtual machine. He could also decide to divert the normal execution flow of the program, which would then be a control-based attack, also mentioned in section 3.4.1. Should the attacker do this successfully, verifiable computation, the property that VerComp aims to guarantee, remains unchanged. In fact, once the execution is complete and the receipt is generated, it will be incorrect, and thus the client will be aware that something has gone wrong. The attacker could also try to modify a receipt, with the intention of faking it. For instance, he might try to modify the output of the program, the journal information. Again, the client would become aware of this during verification. Finally, as far as replay attack is concerned, this is not directly counteracted by VerComp. To prevent an attacker from modifying the execution flow by sending the client a receipt relating to a correct but previous execution, a nonce must be added as an additional input to the application and committed. In this way, when the client later goes to check the receipt, it would also check the nonce in the Journal to ensure that it is not the victim of a replay attack.

Chapter 6

VerComp Implementation

In this section, practical aspects of the implementation are explored in greater detail. First, the handling of data formats will be explained, followed by a description of the routes implemented by the server to manage requests.

6.1 Data serialization

The ELF file running within the virtual machine reads input data when necessary using a method provided by the RISC Zero library. This method allows for the reading of data of any type, including classes defined by the guest itself. This approach gives the developer complete freedom to handle input data as she pleases. However, in VerComp, this flexibility introduces a compatibility issue, as it is not possible to know in advance the type of variable that a program will take as input.
To resolve this issue while still allowing the use of all data types, the input is conventionally serialized as JSON. This approach ensures that the user retains full flexibility in handling data, but it also requires them to independently manage the serialization and deserialization of the data.
The format of the Receipt has also been modified. The original format, as defined by RISC Zero, is not directly usable. For this reason, each time before returning the receipt to the client, the server serializes it using *bincode*, a well-known Rust crate for serializing and deserializing structs[1] into bytes [32].
The last and only object whose management cannot be facilitated and improved is the Journal. This is defined as a vector of bytes and as such, in order to allow for better understanding and interpretation of the information contained within it, it would be necessary to know how to interpret the bytes at the time of reading. The process of verifying the receipt, and thus the reading of the Journal, takes place via a CLI which, in order to keep the command simple, cannot also handle this aspect. This is why the Journal array is printed as a sequence of bytes during verification.

6.2 Server

6.2.1 APIs

Two APIs are made available by the server, which manage the two possible client requests. The endpoint `POST executable/add` requires the binary ELF file to be inserted within the POST

[1] In Rust the struct is the equivalent of class.

body. The server subsequently saves this file in the local directory `executables` with a randomly generated 6-character identifier. In the event that the file is successfully transmitted and copied, the server returns a `201 status code` and a JSON object containing only the ID value to communicate the executable's identifier.

The other endpoint is `POST executable/run/<elf_id>`, where the path parameter includes the file extension (e.g., `bkos2x.elf`). The `POST` method was chosen because the input to be subsequently fed to the application is passed in JSON format within the body. It will then be the Executor's responsibility, if the input is present, to communicate it to the application within the zkVM. In the event that the application's execution terminates without errors, the Receipt is returned serialized using `bincode`. Otherwise, text containing the error that interrupted the execution is returned.

6.2.2 Technical implementation

On the server side, as on the client side, the software was implemented in Rust. This choice was guided by the characteristics of this programming language, including the resolution of dangling pointers, protection against buffer overflows, critical runs, arithmetic overflows and much more. The checks on these criticalities and many others, unlike many other languages, are done at compile time, thus avoiding unexpected behaviour and devastating consequences. Wanting to make a comparison with Java, Rust does not need a garbage collector, precisely because of this long series of checks that are done at compile time and not at runtime. Besides the great potential of the language, the choice of the latter is also due to the fact that the RISC Zero library is written in Rust, so it was also easier to integrate RISC Zero into a program written directly in Rust. This library, like Rust on the other hand, is based on strong typing, which helps to avoid errors and especially to reveal them at compile time. Another useful feature is JSON deserialisation, which is automatically handled by Rocket. In terms of security, it facilitates the implementation of HTTPS, data limits to prevent DoS attacks and much more. Finally, it also provides tools for carrying out unit and integration tests.

The CLI tool made available to the client is also written in Rust. The decision to realise this tool is driven by the objective of increasing the usability of the software, more precisely the verification phase. This phase can be encapsulated by the execution of a verification function, made available by RISC Zero, which, receiving the imageID and receipt as input, determines the correctness of the latter. However, performing this operation without the help of this tool would mean having to manually launch a script each time to which these arguments are passed. Moreover, the ImageID is managed internally by RISC Zero with the type `Digest`, which is obviously not directly compatible with the base64 string returned as a result of the build (see section 5.3). It is therefore also necessary to perform a decode operation. All these operations are handled by this tool, defined using the clap [33] library, which stands for Command Line Argument Parser. It was very helpful because it was sufficient to define an enum variable with the command and the various options with their characteristics (data type, optional, etc.). Providing the essential information then takes care of everything else, including the generation of error messages, help messages and even giving the shell hints on how to complete an argument (as in the case of the receipt where the argument is accepted as long as it points to an existing file).

Chapter 7

Test and Validation

7.1 Testbed

In order to assess the proper functioning of VerComp and its security, a single machine acting as both client and server was used. Below are the technical specifications:

- CPU: Intel® Core™ i5-4440 × 4.
- RAM: 16,0 GiB.
- OS: Fedora Linux 40 (Workstation Edition).
- Kernel version: Linux 6.10.11-200.fc40.x86_64.
- rust: 1.77.2.
- cargo: 1.77.2.
- RISC Zero toolchain: 0.19.1.

7.2 Functional Test

PasswordChecker is used to prove that the receipts are strictly related to the ImageID, hence to the source code. In this test, a receipt is generated after execution of the PasswordChecker app. The application code is then modified, omitting the policy check. These lines of code shown in 7.2 are then commented out. By doing so, the chosen password is not checked, but the application can

```
if !POLICY.is_valid(&request.password) {
  panic!("Password invalid. Please try again.");
}
```

still be executed successfully. What is important is that the ImageID changes from the previous one. A new receipt is generated for the app with the policy check removed, with the following command:

```
curl $endpoint/executable/run/du3hsv.elf  \
--data '{"password": "password","salt":
  [12, 198, 45, 128, 250, 67, 14, 89, \
  255, 3, 172, 201, 90, 37, 112, 56, \
  144, 220, 7, 33, 244, 188, 67, 0, \
  154, 200, 47, 69, 120, 213, 88, 132] \
}'
```

At the end of execution, a valid receipt is generated. Through the verify command, it is possible to distinguish and recognise that a receipt is related to the first version of the password checker or related to the version that does not check the policy. This distinction is possible thanks to the imageID, which, regardless of how large the changes to the code are, changes. Consequently, when verifying the same receipt (the last one generated) by referring to the imageID of the original PasswordChecker (which correctly checks the policy), the verification ends with a negative result:

```
The receipt is not valid.
image_id mismatch
```

If, on the other hand, the verificatio is carried out with a correct receipt and the right ImageID, it is printed:

```
The receipt is valid. This is the journal:
Journal {...}
```

7.3 Performance Tests

Considering the hardware limitations of the machine used, performance tests were carried out in the following situations:

- The server runs the application on bare metal and in the zkVM without generating the receipt.
- The server runs only one application at a time.
- The server runs two applications at a time.
- The server runs three applications at a time.

In each of these four cases, measurements of the same application were repeated 10 times, in fact, 10 different trends over time can be distinguished in the following graphs.

7.3.1 Execution on the machine and without generating the receipt

With the aim of having a basis for comparison, further measurements were first carried out with regard to the execution of the PasswordChecker application without the aid of RISC Zero. Only the code actually executed by the guest, i.e. the generation of the password hash if the policy is complied with. Given the speed of this operation, the only approximate metric that could be grasped from this application was the execution time. Even in this case, the measurement was taken 10 times and the result is 0,031s per execution.

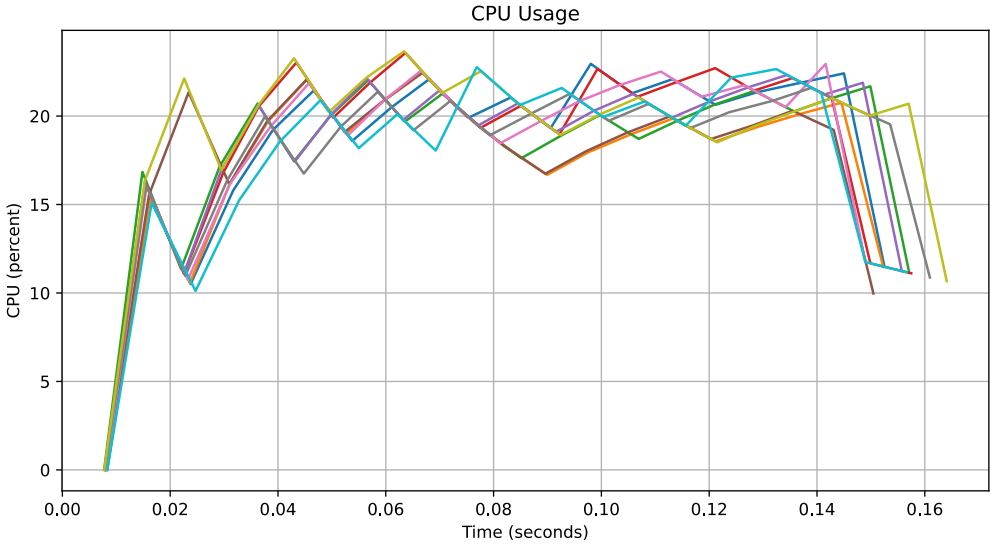

Figure 7.1. Measurement of the CPU consumption of the PasswordChecker application within the zkVM (without generating the receipt).

The second measurement was made on the application that makes use of RISC Zero, thus also of zkVM, but does not generate the receipt. To do this, it was sufficient to place the environment variable `RISC0_DEV_MODE=1` before the executable to be launched. Obviously, the execution time and the expenditure of resources is greater than if the application was executed directly on the machine.The figure 7.1 shows the CPU consumption over time and the figure 7.2 the memory consumption. These graphs show that using the zkVM increase the execution time by an order of magnitude compared to execution on the machine. However the running time remains below 0.156 seconds and on average 0.148.

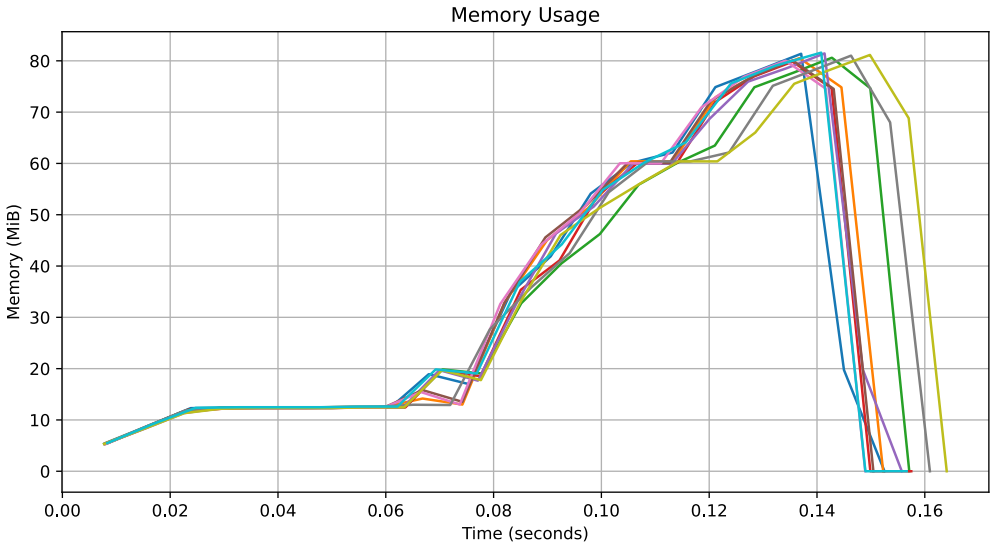

Figure 7.2. Measurement of the memory consumption of the PasswordChecker application within the zkVM (without generating the receipt).

7.3.2 One or more parallel executions

Regardless of the number of applications launched in parallel, the method for measuring performance was as follows. Using a script written in Python, the command to start the Vercomp server is launched. Of this, the PID and all its child processes are considered. At an interval of one second, measurements of CPU and RAM consumption are taken and written to a CSV file. Once the measurement was finished, using another script also written in Python, two different graphs were generated representing the consumption of each of these two resources.

All cases in which the receipt is generated are enclosed here. Since the generation of the receipt is the most onerous task of Vercomp, the time will be incredibly long compared to the previous ones. Table 7.1 summarises the results of these tests.

Considering the previous measurements taken while not generating the receipt, it can be seen that the execution itself has a minimal, practically negligible time compared to the generation of the receipt. In fact in figure 7.3 can be seen that the average duration is 406 seconds, reaching a maximum of 452 seconds (approximately 7 minutes). Figure 7.4 shows memory consumption.

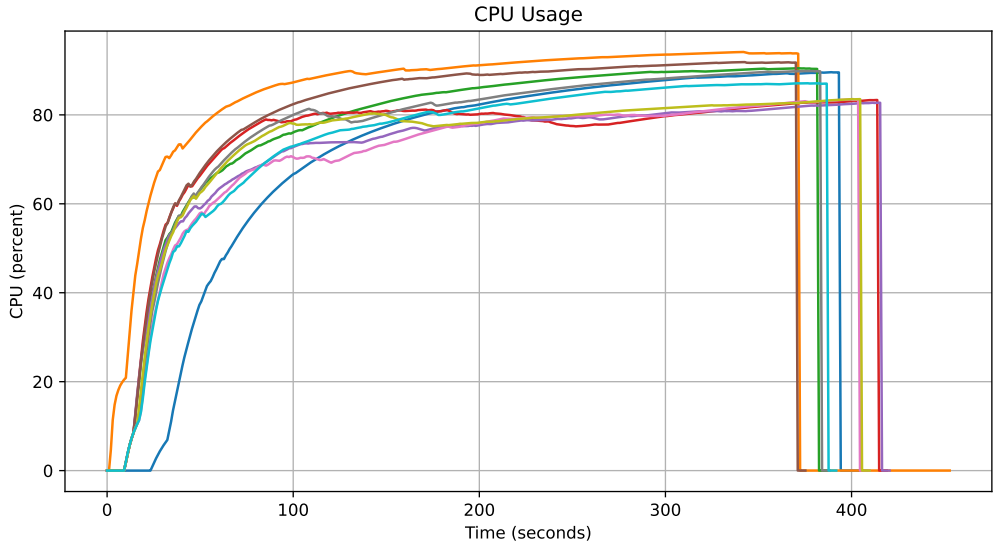

Figure 7.3. Measurement of the CPU consumption of one PasswordChecker application.

Compared to the memory consumption shown in figure 7.2, in this case it can be seen that in all executions, at around the second 170, there is a brief plunge and shortly afterwards a small rise, to then remain practically constant until the end. When running two applications in parallel, the peak CPU consumption is around 95% 7.5, as in the case of a single application. However, the average CPU consumption increased from 72% to 87%. In fact, the curve in the first phase is steeper than in the previous case. Memory had a predictable trend as average and peak consumption doubled to 7846 MiB and 10002 MiB, respectively (7.6). The valley previously described continues to be seen to occur proportionally at the same spot. With one more application than in the previous case, the time increased by 1.7 times, almost doubling.

In the last case under consideration (Figures 7.7 and 7.8), the memory consumption growth remained constant and the average CPU consumption increased by 4 percentage points to 91%.

As had been the case when switching from one to two applications, the average execution time increase by about 300 seconds (5 minutes) to an average time of 1058 seconds and a maximum time of 1066 seconds (more than 7 minutes).

Finally, the table 7.1 is shown, in which all the results discussed so far are merged to give a general overview of how the increase in applications running in parallel increased the resources used.

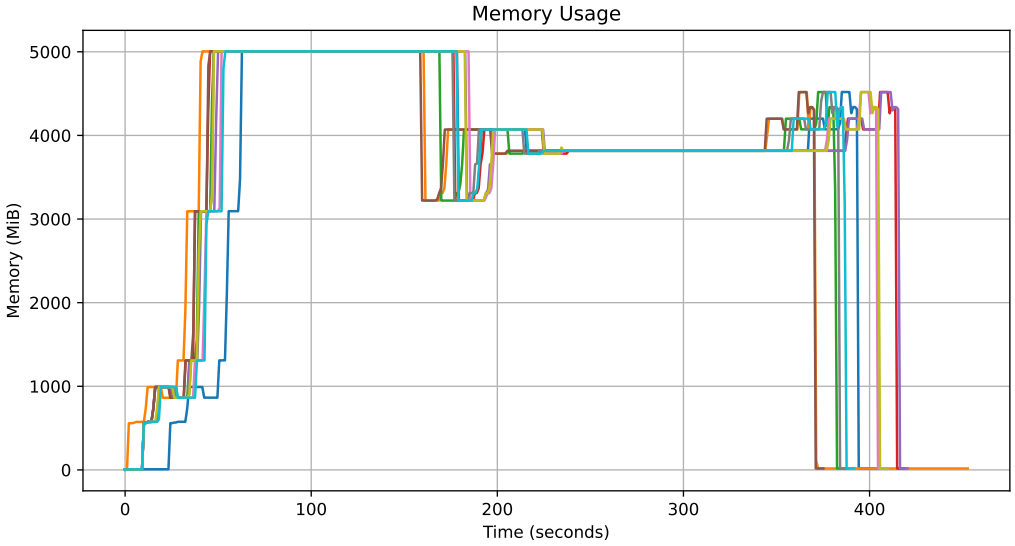

Figure 7.4. Measurement of the memory consumption of one PasswordChecker application.

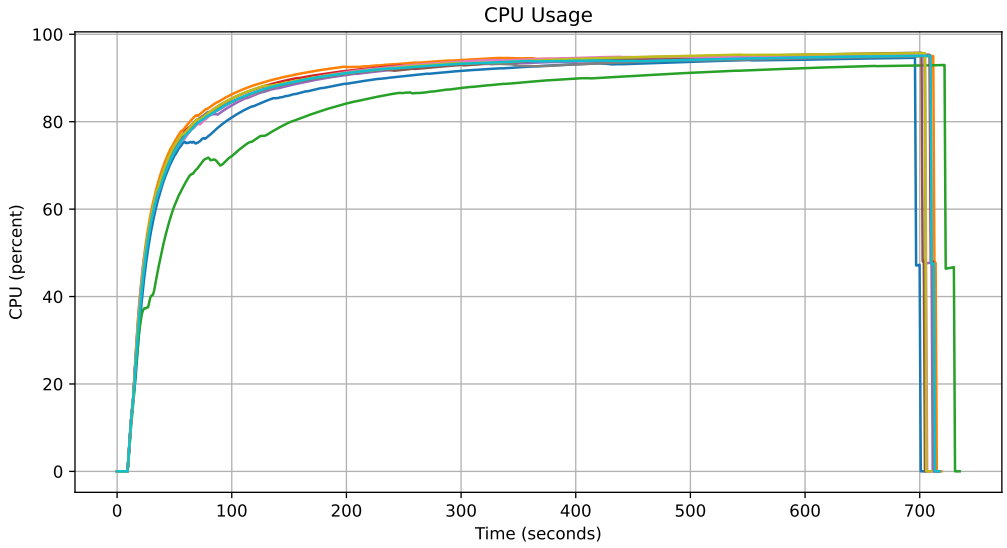

Figure 7.5. Measurement of the CPU consumption of two PasswordChecker applications.

7.3.3 Verifier

Lastly, a mention of the tests carried out on the verifier. In order to understand its performance, the execution of the verify command was considered, which determines whether a receipt relating to an ImageID is correct or not. The duration of the execution was then measured 10 times. The time over the various measurements remained fairly constant, with an average duration of 0.12 seconds. The command launched is as follows:

```
risc0_receipt_verifier verify \
  --image-id a08818cc157091e7b36f0f68754e75f5ce23bee478b83e157ae93243574da40d \
  --receipt receipt.risc0
```

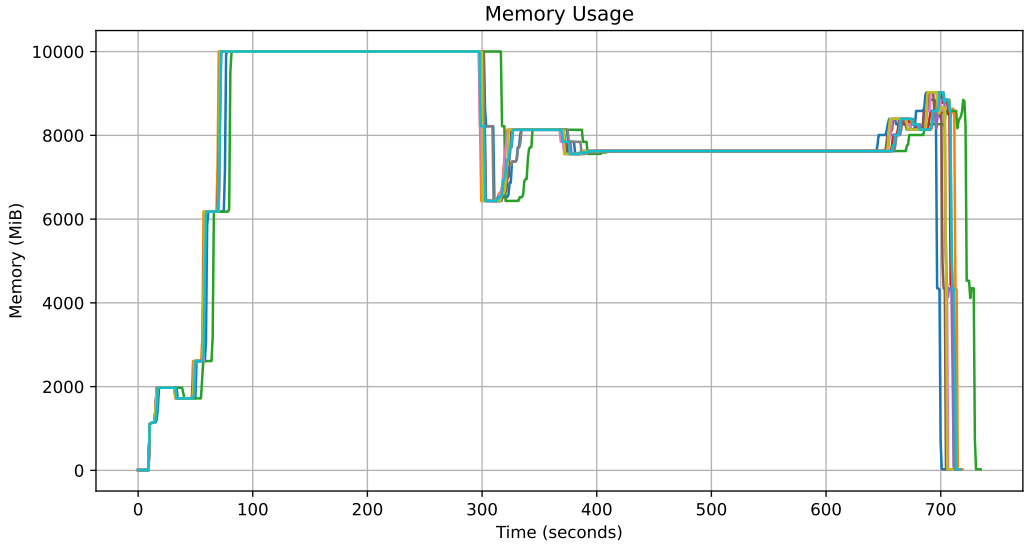

Figure 7.6. Measurement of the memory consumption of two PasswordChecker applications.

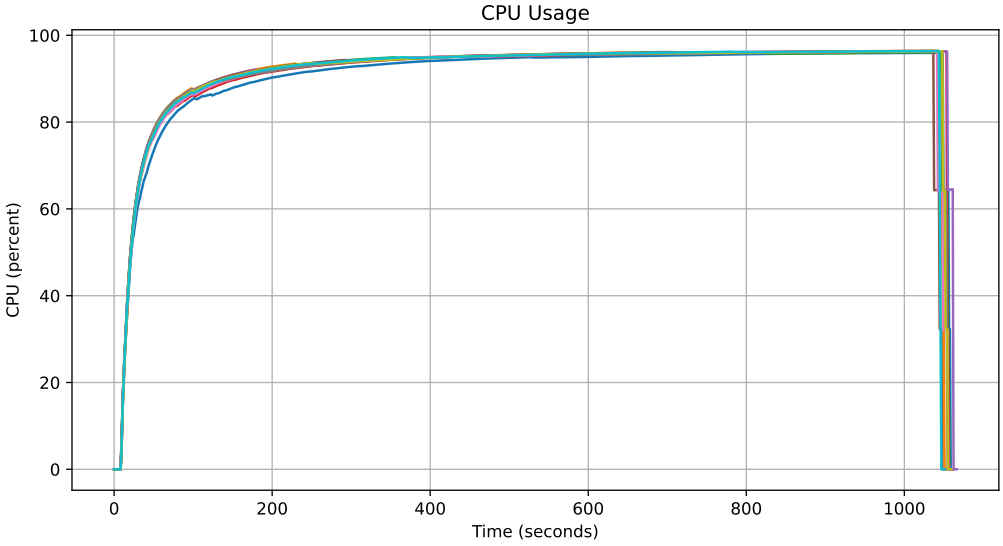

Figure 7.7. Measurement of the CPU consumption of three PasswordChecker applications.

	Time (seconds)			**CPU** (%)			**Memory** (MiB)		
	Min	AVG	Max	Min	AVG	Max	Min	AVG	Max
Without receipt	0.14	0.14	0.15	0	18.17	23.65	0	33.08	81.59
One application	375.17	406.13	452.74	0	72.80	94.14	0	3756.96	5006.02
Two applications	704.97	715.37	734.88	0	87.08	95.76	0	7846.42	10002.77
Three applications	1050.89	1058.57	1066.28	0	91.12	96.44	0	11430.52	14872.66

Table 7.1. Overview of time and resources in the various performance tests carried out.

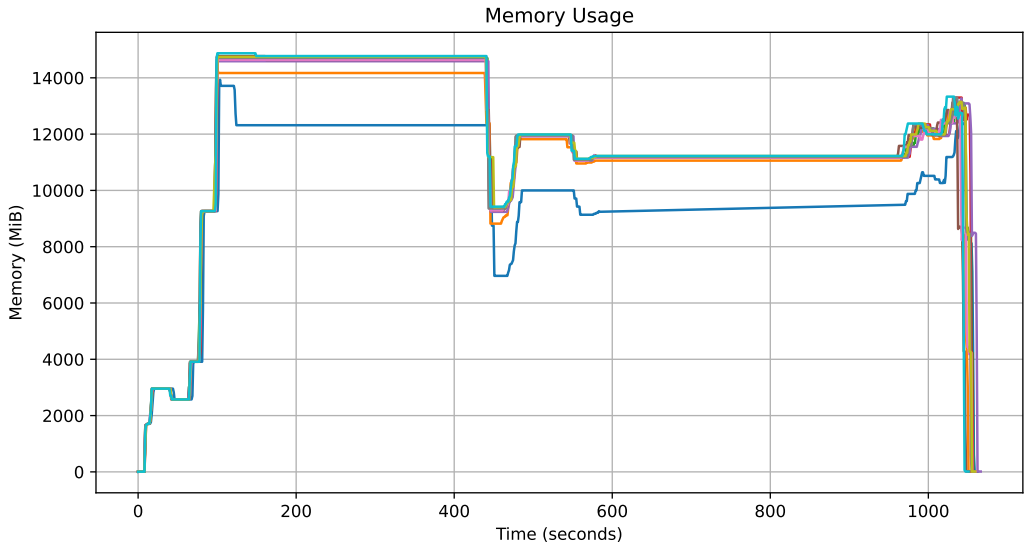

Figure 7.8. Measurement of the memory consumption of three PasswordChecker applications.

Chapter 8

Conclusion

The main objective of this thesis was to investigate the area relating to secure remote execution, i.e. verifiable computation. To this was added the study and use of ZKP technologies. In fact, before developing the final solution, several solutions were explored and evaluated to implement the verifiable computation mechanism with the addition of ZKP. In the end, the choice fell back on the zk-STARK protocol. Numerous reasons guided this choice, among which, the non-compulsory trusted setup phase and above all the fact that it is post-quantum resistant since, unlike zk-SNARK, it is based on technologies that, for example, do not rely on the discrete logarithm problem.

Once a receipt for an application has been generated, VerComp allows anyone in possession of that application, or its identifier, to ensure that the execution was correct, thus guaranteeing verifiable computation without disclosing private information. In addition to being certain that the application was executed correctly, the output of the program (also protected from tampering) is also attached. It is also possible to integrate most of the available external libraries, distributed by the Rust package manager, into the application. This possibility makes the development of such applications easier and faster for developers.

On the other hand, VerComp, as demonstrated by the conducted tests, exhibits execution times that are several orders of magnitude higher due to the generation of the receipt, compared to execution without receipt generation. This significant slowdown is primarily attributable to the zk-STARK protocol, which, in order to generate the proof, must first meticulously save all execution data (in this case, the virtual machine data) on a clock cycle by clock cycle basis, and then process this vast amount of information to extrapolate the proof. This intricate process, while ensuring a high degree of security and verifiability, introduces a substantial computational overhead.

The complexity and computational intensity of the zk-STARK protocol directly impact the system's performance. Each operation must not only be executed but also recorded and subsequently processed for proof generation, effectively multiplying the workload for every single operation. This approach, while offering robust zero-knowledge proofs, comes at the cost of increased time and resource consumption.

Consequently, at present, to adapt such a solution to real-life scenarios, it would be necessary to provide adequate hardware support. This might involve high-performance computing systems, possibly including specialised hardware accelerators designed to optimise zk-STARK computations.

Furthermore, an additional consideration arises from the architecture of the library that has been utilised. When running an application that receives data as input, the server (or host) receives this data in an unencrypted form. This architectural characteristic introduces a potential threat for confidentiality. For this reason, to maintain the integrity and confidentiality of this sensitive information, this machine must be considered trusted and must be physically located within a secure, trusted domain.

As far as possible developments and improvements of VerComp are concerned, the APIs made available could be extended. Security mechanisms could be added, including the authorisation whereby, when uploading, the user specifies whether the uploaded application can be executed by anyone, only by himself or by a specific group of users. Obviously, such an underlying mechanism would need an authentication mechanism to identify the various users. Finally, as a matter of readability, the display of the Journal (public output) could be improved. At present, this is printed out as a sequence of bytes, since the server is not aware of the type of object being transported in it. Furthermore, this object could be of a custom class, specially created by the client, which the server does not even know exists. It could be considered to bind the journal output type to Rust's standard types, or to make a more complex modification that allows the server to also be provided with the class that defines the object transported in the journal.

Bibliography

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems", Proceedings of the seventeenth annual ACM symposium on Theory of computing - STOC '85, 1985, pp. 291–304, DOI 10.1145/22145.22178

[2] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems", Advances in Cryptology — CRYPTO' 86 (A. Odlyzko, ed.), Berlin, Heidelberg, 1987, pp. 186–194, DOI 10.1007/3-540-47721-7_12

[3] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems", Journal of the ACM (JACM), vol. 39, no. 4, 1992, pp. 859–868, DOI 10.1145/146585.146605

[4] S. Cook, "The p versus np problem", Clay Mathematics Institute, vol. 2, 2000, p. 6

[5] A. Shpilka and A. Yehudayoff, "Arithmetic circuits: A survey of recent results and open questions", Foundations and Trends® in Theoretical Computer Science, vol. 5, no. 3–4, 2010, pp. 207–388, DOI 10.1561/0400000039

[6] L. D. Mol, "Turing Machines", The Stanford Encyclopedia of Philosophy (E. Zalta, ed.), Metaphysics Research Lab, Stanford University, Winter 2021 ed., 2021

[7] M. A. Barbara, "Proof of all: Verifiable computation in a nutshell", arXiv preprint arXiv:1908.02327, 2019

[8] U. Feige and A. Shamir, "Zero knowledge proofs of knowledge in two rounds", Advances in Cryptology — CRYPTO' 89 Proceedings (G. Brassard, ed.), New York, NY, 1990, pp. 526–544, DOI 10.1007/0-387-34805-0_46

[9] C. Schnorr, "Efficient signature generation by smart cards", Journal of cryptology, vol. 4, 1991, pp. 161–174, DOI 10.1007/BF00196725

[10] I. Damgård, "On σ-protocols", Lecture Notes, University of Aarhus, Department for Computer Science, vol. 84, 2002

[11] T. Chen, H. Lu, T. Kunpittaya, and A. Luo, "A review of zk-snarks", 2023, DOI https://doi.org/10.48550/arXiv.2202.06877

[12] Zcash: Privacy-protecting digital currency, https://z.cash/

[13] C. Reitwiessner, "zksnarks in a nutshell", Ethereum blog, vol. 6, 2016, pp. 1–15

[14] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps", Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32, Berlin, Heidelberg, 2013, pp. 626–645, DOI 10.1007/978-3-642-38348-9_37

[15] V. Buterin, "Quadratic arithmetic programs: from zero to hero", https://vitalik.eth.limo/general/2016/12/10/qap.html

[16] J. Yang, W. Zhang, Z. Guo, and Z. Gao, "Trustdfl: A blockchain-based verifiable and trusty decentralized federated learning framework", Electronics, vol. 13, no. 1, 2024, DOI 10.3390/electronics13010086

[17] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity", Cryptology ePrint Archive, Paper 2018/046, 2018, https://eprint.iacr.org/2018/046

[18] E. Ben-Sasson, "Computational integrity definition", https://softwareengineeringdaily.com/2019/03/04/starkware-transparent-computational-integrity-with-eli-ben-sasson/

[19] A. Berentsen, J. Lenzi, and R. Nyffenegger, "A walk-through of a simple zk-stark proof", Available at SSRN 4308637, 2022, DOI https://dx.doi.org/10.2139/ssrn.4308637

[20] X. Yu, Z. Yan, and A. V. Vasilakos, "A survey of verifiable computation", Mobile Networks and Applications, vol. 22, no. 3, 2017, pp. 438–453, DOI 10.1007/s11036-017-0872-3

[21] H. Ahmad, L. Wang, H. Hong, J. Li, H. Dawood, M. Ahmed, and Y. Yang, "Primitives towards verifiable computation: a survey", Frontiers of Computer Science, vol. 12, no. 3, 2018, pp. 451–478, DOI 10.1007/s11704-016-6148-4

[22] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems", Proceedings of the twentieth ACM symposium on Operating systems principles, 2005, pp. 1–16, DOI 10.1145/1095810.1095812

[23] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers", Advances in Cryptology – CRYPTO 2010 (T. Rabin, ed.), Berlin, Heidelberg, 2010, pp. 465–482, DOI 10.1007/978-3-642-14623-7_25

[24] H. B. Debes, E. Dushku, T. Giannetsos, and A. Marandi, "Zekra: Zero-knowledge control-flow attestation", Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, New York, NY, USA, 2023, p. 357–371, DOI 10.1145/3579856.3582833

[25] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks", 2016 IEEE Symposium on Security and Privacy (SP), 2016, pp. 969–986, DOI 10.1109/SP.2016.62

[26] P. Liakos, K. Papakonstantinopoulou, and A. Delis, "Realizing memory-optimized distributed graph processing", IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 4, 2018, pp. 743–756, DOI 10.1109/TKDE.2017.2779797

[27] A. Waterman, "Design of the risc-v instruction set architecture". PhD thesis, EECS Department, University of California, Berkeley, Jan 2016

[28] RISC-V Foundation, "The risc-v instruction set manual, volume i: User-level isa, document version 20191213", Dec 2019. Please cite as: "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213"

[29] "Risc zero zkvm optimization", 2024, https://dev.risczero.com/api/zkvm/optimization#most-risc-v-operations-take-exactly-one-cycle

[30] "Operation costs in cpu clock cycles", 2024, http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

[31] "Hashcat v6.2.6 benchmark on the nvidia rtx 4090", 2022, https://gist.github.com/Chick3nman/32e662a5bb63bc4f51b847bb422222fd

[32] "Bincode: A serialization library", 2024, https://github.com/bincode-org/bincode

[33] K. B. Knapp and The Clap Community, "clap", September 2024, https://github.com/clap-rs/clap

[34] G. Hoare, "Rust programming language empowering everyone to build reliable and efficient software", https://www.rust-lang.org

[35] "RISC Zero GitHub repository", https://github.com/risc0/risc0/

[36] D. Stenberg, "curl, command line tool and library for transferring data with urls", 2024, https://curl.se/

[37] S. Klabnik and C. Nichols, "The Rust Programming Language", The Rust Project Developers, 2024. https://doc.rust-lang.org/book/

Appendix A

User's manual

This chapter explains the process for installing and using the VerComp framework. The instructions that the server machine must follow to install and run VerComp are first introduced, and then those of the client. With regard to the executed application, the Password Checker (mentioned in section 4.6) will be taken as an example.

A.1 Server installation

Since the server application is written in Rust, version 1.77.2 of Rust and Cargo must first be installed, according to the official website [34]. Following installation, it will not only be possible to build and execute projects, but also to manage and import dependencies, as installing Rust also automatically installs Cargo, the package manager.

At this point, the RISC Zero toolchain must be installed. The RISC Zero documentation mentions commands to be sent from the terminal to install the toolchain directly via Cargo. However, this cannot be done, because the latest version would be installed, while VerComp makes use of the `0.19.1` version. Therefore, the best solution is to retrieve it directly from the [35] repository on GitHub and download the already compiled binary. It will then be necessary to move the contents of the archive into the `/.cargo/bin/` folder, where the globally installed executables are located. Once the binaries are moved, the commands `cargo-risczero risczero install` and `cargo-risczero risczero build-toolchain` must be run. In this way, it is possible to immediately check whether everything is working by running the command `cargo-risczero -V` to check whether the binary file has been saved in a folder in the `$PATH` variable or not. If the result is positive, the next step is to proceed with the actual installation of the server application.

Assuming that the VerComp source code is available, move into the folder and compile the project with the command `cargo build` or `cargo build --release` if it is desired to generate the optimised executable. The result of the build will be found inside the `target` folder and then on the corresponding subfolder, depending on which type of build was made. Finally, the executable is launched, and the server is ready to receive and execute the client's requests. From the moment the application runs, the ELF executables received from the client will be stored in the `executables` folder. In addition, a web server will be created that will respond to requests received at port `8000`.

A.2 User installation

In the same way as the server, the client also needs to install Rust and Cargo first. For the client, they are even more necessary because it must develop the application and integrate it with the libraries it needs. Therefore, it must follow the same procedure as mentioned above (including moving the executables to the correct folder). Once the installation of Rust and

Cargo is finished, RISC Zero will also be installed in the same way as the server. Next, it is necessary to ensure that Docker is installed on the machine. The latter is essential because, as mentioned in the section 5.3, the deterministic generation of the ImageID following the build is guaranteed thanks to a reproducible build method implemented using the Docker container. The last installation required concerns the software with which to send HTTP requests. During development, cURL [36] was used, so follow the instructions on the official site for installation.

```
cargo risczero new my_project --guest-name guest_code_for_zk_proof
```

The client only has to deal with the module of the *guest*, and thus with its corresponding folder. This is because, the executable that is generated after the build, and which is supplied to the server, is the one related to the guest and not to the host, which is instead completely implemented and managed server-side. It is important to remember that the data before being exchanged must be serialised according to the logic explained in the 6.1 section, so it must be considered that the input received from the host is encoded as JSON, so parsing must be done before it can be used.

To simplify the explanation and give a practical demonstration, assume that along with the VerComp source code, the code for the *PasswordChecker* is supplied. The application is then prepared and ready for the build process to be executed. Start the *Docker* service and then run this application build command from inside the project folder:

```
cargo risczero build --manifest-path methods/guest/Cargo.toml
```

It is important that Docker is running because it is being used at the moment. In fact, by reading the terminal logs, it can be clearly seen that a container is being launched. At the end of the operation, the ImageID for the application and the path to the application compiled with the corresponding identifier are printed out on the terminal.

At this point, the first request to the server can be made: uploading the app. Assume that the IP address or URL of the server is stored in the system variable `$server`, then the command must be run:

```
curl $server/executable/add --data-binary "@<elf_path>"
```

Where `elf_path` is the path to the executable given along with the ImageID. The server will return the application ID (`id_app`). The second request that is sent by the client is to request the execution of the application that has just been loaded:

```
curl $server/executable/run/<id_app> --data {"password":"<plain_password>",
    "salt":  [161,252,...]} --output receipt.risc0
```

At the end of the request, the client will save the receipt transmitted by the server under the name `receipt.risc0`, as specified in the command, in the current path.

The client may decide whether to verify the receipt itself and/or to communicate it to third party clients. Regardless of this, anyone wishing to verify its correctness must fill out the verifier CLI. This is also made available with the VerComp source code and must be installed by pointing to the *verifier* folder with the command `cargo install --path .` . The executable `risc0_receipt_verifier` will be installed into the `.cargo/bin` folder. At this point, simply run this command:

```
risc0_receipt_verifier verify --image-id <image_id> --receipt <receipt_path>
```

Appendix B

Developer's manual

This chapter will describe the APIs made available by the server to manage the addition and execution of applications.

B.1 VerComp Server APIs

Add executable

POST /executable/add

It uploads the executable to the server.

Request Object

- elf_file (File): the file to be uploaded.

Response JSON Object

- id (`string`): randomly generated by the server. 6 random alphanumeric characters with an elf extension.

Example response

```
{
  "id": "fwo2j7.elf"
}
```

Run executable

POST /executable/run/<elf_id>

It requires the execution of the app identified by `<elf_id>`.

Request JSON Object

The JSON representation of the input data that the executable needs to run.

Request JSON example

```
{
  "password": "S3cretPla1nPWD!",
  "salt": [
    12, 198, 45, 128, 250, 67, 14, 89,
    255, 3, 172, 201, 90, 37, 112, 56,
    144, 220, 7, 33, 244, 188, 67, 0,
    154, 200, 47, 69, 120, 213, 88, 132
  ]
}
```

This example takes up the PasswordChecker program discussed in the section 4.6.

Response Object

Regardless of the outcome of the computation, if the executable is correctly developed and thus handles unrecoverable errors appropriately, the receipt is returned. The unrecoverable errors, as specified in Rust [37], are the cases where the program *panics*, e.g. tries to access memory locations beyond the size of the array.

B.2 Verifier

```
#[derive(Subcommand)]
pub enum Commands {
  /// Verifies a receipt based on its path and base64 encoded imageID
  Verify {
    /// base64 encoded ImageID
    #[arg(short, long)]
    image_id: String,
    /// Path of the receipt
    #[arg(short, long, value_hint= clap::ValueHint::FilePath)]
    receipt: String,
  },
}
```

This is the enum variable which, thanks to clap, can define one or more commands with their options attached. As can be seen, this library allows a simple and clear definition of the CLI. The syntax of clap is so compact that even comments marked "\\\" are used as a description of the related option. In the above case, the command is only one, as already mentioned, and checks the receipt via the two options `image_id` and `receipt`.

The result from the terminal is shown in the Figures B.1, B.2. The ImageID is decoded via the trait `From` that converts from a base64 string to an object of type `Digest` (see code B.2. In Rust, the trait defines a functionality that a given type has, which it also shares with other types. For example, a function may also be defined in which the required parameter is not of a specific type but implements a specific trait.

The receipt, as already mentioned in section 6.1, once produced, is serialised so that it can be communicated via HTTP and deserialised once received. The command shown in the figure B.2 is used.

Figure B.1. Risc ZERO receipt verifier description.

Figure B.2. Risc ZERO receipt verifier, verify command help.

```
let image_id_digest = Digest::from_hex(image_id)?;
```

Figure B.3. Decode base64 ImageID.

```
let receipt_serialized: Result<Vec<u8>> = bincode::serialize(&receipt);
```

Figure B.4. Receipt serialize.