OST
Eastern Switzerland
University of Applied Sciences

IBM **Research**

MASTER THESIS

Assessing RISC Zero using ZKit: An Extensible Testing and Benchmarking Suite for ZKP Frameworks

Author:
Roman BÖGLI

Supervisors:
Dr. Angelo DE CARO
Dr. Kaoutar EL KHIYAOUI

Academic Supervisor:
Dr. Alexandru CARACAS

Examiner:
Dr. Thorsten KRAMP

A work submitted in fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Science (MSE CS)

at the

Institute for Network and Security (INS),
Departement of Computer Science,
Eastern Switzerland University of Applied Sciences (OST)

in collaboration with the

Security Department at IBM Research Zurich.

January 23, 2024

Abstract

Zero-Knowledge Proofs (ZKPs) are cryptographic protocols designed to verify a statement without disclosing any information beyond its boolean verification outcome. A prevalent use case for such protocols resides in the realm of digital cash, where payers whish to prove the validity of a token without disclosing any specification about the token in particular to uphold privacy. Numerous ZKP software libraries or frameworks have emerged to enhance accessibility for developers and encouraging the widespread adoption of this technology in practical applications.

The thesis introduces the readership to ZKPs, elucidating their fundamental attributes and delving into two key implementation families — namely, Succinct Non-Interactive Argument of Knowledge (SNARK) and Scalable Transparent Argument of Knowledge (STARK) systems. Our emphasis is on the latter, given its perceived post-quantum security. Furthermore, we provide an overview of promising STARK-based ZKP frameworks and discuss their distinguishing features.

One of such frameworks is RISC Zero, which facilitates verifiable general-purpose computations in zero-knowledge through its virtual machine. Essentially, it proves statements indirectly by proving the integrity of all chronologically recorded register states throughout a computational process. We elucidate its internal mechanisms and assess its efficiency by examining two different disciplines. The first involves proving a preimage to a hash value. The second entails proving the membership of a data leaf within a Merkle tree, also known as inclusion proofs (IP).

To streamline systematic analysis, we present the concept of `ZKit`, an extensible test and benchmark suite designed to accommodate diverse ZKP frameworks. `ZKit` enables the orchestration of activities through a command line interface and incorporates a suggested information exchange format for IPs. Additionally, it demonstrates the process of porting ZKP functionality defined in the Rust programming language to Go.

We utilize `ZKit` to benchmark RISC Zero across various settings and workloads. Our results reveal that generating a single STARK proof for a batch of IPs can be up to 3.9 times more efficient than proving each IP individually. Verifying such batch proofs can even offer a performance improvement of up to 14 times. The thesis concludes by discussing key insights gained during the research process and summarizes the implications of the findings.

Keywords: Zero-Knowledge Proof · Inclusion Proof · STARK · RISC Zero · Benchmark

Declaration of Authorship

I, Roman BÖGLI, declare that all material presented in this paper is my own work or fully and specifically acknowledged wherever adapted from other sources. I understand that if at any time it is shown that I have significantly misrepresented material presented here, any degree or credits awarded to me on the basis of that material may be revoked. I declare that all statements and information contained herein are true, correct and accurate to the best of my knowledge.

Foreword

This thesis constitutes the final work of my Master of Science in Engineering, focusing on Computer Science, at the Eastern Switzerland University of Applied Sciences (OST). In the pages that follow, the term «we» is employed to convey a collaborative and inclusive tone, symbolizing the shared journey between the author, his supervisors, and the research presented in this thesis.

I deeply value the diverse experiences and knowledge gained throughout this study program, reflecting on this period as a truly enriching journey. Thus, I would like to take the opportunity to express my gratitude and extend heartfelt thanks to all those who have contributed to and supported me during this journey.

First and foremost, I would like to thank Dr. Alexandru Caracas, Dr. Angelo De Caro, and Dr. Kaoutar El Khiyaoui for their invaluable supervision throughout this thesis. Collaborating with you has been a pleasure, and I have truly appreciated the constructive discussions we engaged in. Also, I thank Dr. Thorsten Kramp for his willingness to serve as an examiner for this thesis.

Additionally, my gratitude extends to Dr. Nathalie Weiler for her prized support throughout the main course of my studies and for proposing the interesting opportunity to collaborate with IBM Research Zurich.

Many thanks go to my former teacher, Stefan Rey, for introducing me to the world of programming years ago, thereby playing a key role in shaping my academic path. I also want to express my appreciation for the invaluable support from the Swiss Study Foundation throughout this journey and the Werner Siemens Foundation for their generous fellowship.

Lastly, I want to extend my sincere thanks to my beloved partner for graciously accepting and supporting the numerous hours I spend immersed in my passion for computers.

Contents

1 Introduction

In the realm of digital money, a common use case involves proving the membership of a coin or token within a publicly accessible set of valid coins. The authority responsible for issuing or minting these valid coins establishes this set, for example, using a Merkle tree data structure [1]. To demonstrate the inclusion of a specific coin in this set, a prover $\mathcal{P}$ provides an *authentication path* or *Inclusion Proof (IP)*, which is a list of hash values or digests from the tree's leaf to its root. The verification process involves verifier $\mathcal{V}$ hashing the leaf data, appending the next digest to the result, and repeating this step until processing all elements in the authentication path. $\mathcal{V}$ is convinced of the coin's inclusion in the set when the final digest matches the publicly known root hash of the Merkle tree. Bitcoin [2], for instance, incorporates this mechanism into its *Simple Payment Verification (SPV)* process or verifying transaction inclusion in its underlying blockchain.

While the above described method is considered an easy-to-implement and efficient way to demonstrate an item's inclusion in a set, it necessitates revealing the particular item in question. In the context of digital money, however, this contradicts with upholding user privacy. In other words, $\mathcal{P}$ should be able to convince $\mathcal{V}$ that a given coin belongs to a set of valid coins without disclosing the specific coin's identity.

The emerging field of *Zero-Knowledge Proof (ZKP)* protocols serves as a solution to this problem. Previous to this work, we investigated existing software libraries or *frameworks* that help to generate and verify ZKPs [3]. In this work, the attention centers on one specific framework called RISC Zero or `risc0`[1]. It enables verifiable general-purpose computations in zero-knowledge through its *Virtual Machine (VM)*, which emulates a reduced *Instruction Set Architecture (ISA)*. This so-called *Reduced Instruction Set Computer (RISC)* apparently inspired the framework's naming.

The reason for focusing on `risc0` in this thesis is twofold. First, it allows to specify circuits or *proof logic* natively using Rust, which leverages the ease to develop custom ZKPs. Second, it implements the *Scalable Transparent Argument of Knowledge (STARK)* protocol which, in contrast to *Succinct Non-Interactive Argument of Knowledge (SNARK)* systems, does not require a trusted setup and is post-quantum secure.

[1] See RISC Zero homepage for details.

1.1 Background and Motivation

The internet has become an integral component of everyone's life today. The number of business models operating through the World Wide Web continues to grow, expanding user dependency accordingly. Not surprisingly, the progress of digitalization has profoundly impacted the most fundamental tool in society – money. From barter systems to precious metal coins, then to paper securities, and now evolving into digital information, money has undergone significant transformations throughout its long history. At least since the introduction of Bitcoin in 2008 [2], an electronic coin that serves as an alternative to conventional money, central banks have progressed to digitalize or *tokenize* cash.

While established and field-tested cryptographic primitives enable the development of such *Central Bank Digital Currency (CBDC)* securely, it remains challenging to maintain the same level of privacy that exists in physical cash transactions. Like banknotes, users must be able to verify the validity of tokens and that they have not spent previously. In the case of physical cash, the latter is ensured automatically through its physical handover from payer to payee. Although validating the authenticity of physical cash is more complex, embedded security features[2] make accurate counterfeiting nearly impossible.

Avoiding to spend cash more than once in the digital word, however, poses a totally different challenge due to the minimal replication costs of digital information. Bitcoin solved this *double-spending* problem using a distributed public ledger, allowing everyone to oversee every transaction. However, this solution comes with tradeoffs in efficiency and privacy due to redundant computations and pseudo-anonymity, respectively. While CBDC might adopt a decentralized architecture, its goal is to uphold privacy levels akin to physical cash. In either case, verifying a token's validity or single-spend condition remains paramount in digital cash.

A promising tool to fulfill these requirements are ZKPs. For example, a depositor could prove to a bank that a token stems from a valid (public) set of tokens without specifying which token exactly. This would uphold the depositor's privacy and consequently avoids traceability. While the theoretical underpinnings of ZKP systems enjoyed increasing attention in recent decades, their employment in practice remains in its nascent stages. A reason for this lies in the high complexity of ZKPs, making them challenging to implement accurately which in turn hinders widespread adoption.

Luckily, numerous software libraries and frameworks now facilitate the implementation of ZKP. While this is a welcomed development, it introduces new challenges as each framework adopts different paradigms or techniques to formulate, produce, and verify such proofs. Even more critical is the specification of zero-knowledge circuits as there exists no unified format to do so. Consequently, understanding the advantages and disadvantages of these frameworks, let alone creating comparable benchmarks, becomes increasingly challenging. This, in turn, impedes informed decision making.

[2] For instance, see explantory video on the security features of the Swiss banknotes.

We anticipate further progress in creating ZKP software libraries that are ready-to-use, have undergone cryptographic audits, and promote intuitive interfaces. The efforts in this thesis are dedicated to these advancements.

1.2 Contribution

This thesis summarizes the most important properties of ZKPs and highlights the key differences between two famous implementation families, namely SNARK and STARK systems. Also, we provide a summary of promising software libraries or frameworks that help to create and verify ZKPs.

Next, we analyze `risc0` in detail using the above-addressed use case of an IP, i.e., proving a the inclusion of leaf in a Merkle tree data structure. Therefore, we propose the concept of `ZKit`, an extensible toolkit for testing and benchmarking various ZKP frameworks. Besides a *Command Line Interface (CLI)* to execute parameterized benchmarks or generate Merkle tree test data, it also contains functionality to define and exchange IPs in a unified way.

Furthermore, `ZKit` exemplifies how ZKP circuits written in Rust can be ported to the Go[3] ecosystem through a wrapper library. This portability facilitates the integration of ZKPs in existing Go projects such as for example the *Fabric Token-SDK (FTS)*[4].

Last but not least, we share our `risc0` benchmark results on IPs in two different settings. In the first setting, we measured the performance and allocated resources to create one proof for a single IP. In the second setting, we aggregate or batch multiple IPs in a single proof. We compare and interpret the results of these two settings at the end and state our recommendations that we drew from it.

1.3 Related Work

In 2020, Benarroch et al. [4] stressed the importance of comparing ZKP frameworks using benchmarks. Among others, they highlighted the value of performing IP, or *set membership* as they term it, in a zero-knowledge context.

Gong et al. [5] focus on the theoretical distinctions among four popular ZKP schemes, including SNARK and STARK. Their motivation is to equip parties interested in this field with a comprehensive understanding of the strengths and weaknesses inherent in each scheme.

Polybase Labs [6] maintains a comparison website showing where they compare their own framework alongside four others, including `risc0`. Besides insights about time, memory, and proof size, they also indicate the incurred dollar costs from running these benchmarks on commercial cloud infrastructure. While `ZKit` adopts their approach to measure memory consumption, we follow a more modular system architecture to minimize redundancy. Also, the CLI contained in `ZKit` facilitates the (local) benchmark execution.

[3] An statically typed programming language with concurrency support developed by Google, making them ideal for scalable (distributed) applications. See Go homepage for details.
[4] A toolkit for the modular blockchain framework *Hyperledger Fabric*. See GitHub repository.

The *zk-benchmarking* project [7] maintains a test suite that also pursues the standardized framework comparison goal. While test results exist for successive hashing, results for IP use cases are still in progress. Their uniform approach to instantiate benchmark jobs was adopted in `ZKit` but expanded to include additional measurable metrics, such as peak memory consumption.

Ernstberger et al. [8] introduced a configurable benchmarking framework focusing on SNARK-based libraries named *zk-Bench*. As our work, they focused on a modular system architecture to facilitate continuous extensibility efforts. A similar SNARK-oriented project is maintained by the *zk-Harness* [9]. The project *zkp-compiler-shootout* [10] also encompasses STARK-based frameworks, albeit with a specific emphasis on runtime durations. Celer [11] analyzes the performance and resource consumption of a single preimage use case with seven different ZKP frameworks.

Overall, a collective effort is underway to develop tools tailored for handling and comparing the heterogeneous array of ZKP frameworks. While this thesis contributes to this effort, it stands out with distinctive features such as a user-friendly CLI, a system architecture designed for extensibility, and a proof of concept for porting functionality to Go. With the first integrated ZKP framework being the `risc0`, we highlight its practicality and underscore the potential of general-purpose verifiable computations executed in a VM and proven through the transparent and quantum-safe STARK system.

1.4 Outline

The rest of this work is structured as follows. Chapter 2 defines the most essential properties of ZKP systems and highlights the key differences between SNARK and STARK systems. Chapter 3 provides an overview of the `risc0` framework. After an introduction to their VM-based proving paradigm that is based on the reduced ISA called *RISC-V*, we explain the actual proving and verifying procedure and its involved information. In Chapter 4, we declare the chosen system architecture of `ZKit` and discuss its main functionalities as well as strategies for extensibility. Chapter 5 presents and summarizes our benchmark results with `risc0` and interprets them. Finally, we discuss our lessons learned as well as future work in Chapter 6 and conclude with the key insights of this thesis in Chapter 7.

2 Zero-Knowledge Proof Systems

This chapter discusses ZKP systems in general. The theoretical concepts were introduced in the early 90s by Goldwasser, Micali, and Rackoff [12], Blum, Feldman, and Micali [13], and Goldreich, Micali, and Wigderson [14]. Quisquater et al. [15] introduced the famous and intuitive ZKP analogy known as «Ali Baba Cave».

For a start, we summarize the essential theoretical attributes that help characterize the quality of these systems. This is followed by a high-level primer on the inner workings of STARK protocol. Furthermore, we introduce our differentiation between direct and indirect proving approaches. Besides stating the motivation behind these two different approaches, we also provide a summary of promising existing STARK-based ZKP frameworks.

2.1 Characteristics

This section revisits the defining properties of ZKP systems, which are essential for evaluating their characteristics. The content presented in this section originates from our previous work [3, Chapter 3.1] and is replicated to ensure completeness and clarity for the readership. Interested readers are referred to Thaler [16], who explores the theoretic aspects of ZKPs in greater detail.

A proof system defines how $\mathcal{P}$ proves the validity of $\mathcal{S}$ to $\mathcal{V}$ in a verifiable manner. Ideally, valid proofs always succeed in this verification process while invalid proofs can always be disclosed as such and consequently be rejected. Generally speaking, one considers a proof *zero-knowledge* when $\mathcal{V}$ solely learns the boolean outcome of the verification process but nothing else.

Let $\mathcal{S}$ be the statement «*I know that* $x = 10$ *for* $13^x \equiv 2262 \pmod{7919}$ *is valid*». Here, x represents the *witness* of $\mathcal{S}$. A *trivial* proving strategy would involve simply disclosing x such that $\mathcal{V}$ can validate $\mathcal{S}$ directly. Large or secret x as well as computationally expensive validation operations, however, ask for different proving strategies that function with partial or even non-disclosure of x. Meanwhile, the ability to convince $\mathcal{V}$ remains equally important. ZKP are such *non-trivial* proof systems.

Describing Properties

ZKP systems exhibit three central properties, as shown below. In order to address these properties, certain cryptographic assumptions apply. This includes, for example, the existence of secure hash functions or the hardness of the so-called *Discrete Logarithm Problem (DLP)*[1].

- **Complete:** When every valid proof generated by an honest $\mathcal{P}$ over a true $\mathcal{S}$ convinces $\mathcal{V}$ then one considers this system as *complete*. In other words, completeness describes the property that $\mathcal{V}$ will always be convinced when presented with a truthful proof.
- **Sound:** Proof systems that are *sound* ensure the property that incorrect proofs or correct proofs of a false $\mathcal{S}$ always fail to convince $\mathcal{V}$. More specifically, one speaks of *statistical* or *information-theoretic* soundness when this property holds against a computational unbounded $\mathcal{P}$. Otherwise, the term *computational* soundness applies.
- **Zero-Knowledge:** A proof that does not disclose any information about its witness while upholding the ability to convince $\mathcal{V}$ is said to be a ZKP. In other words, $\mathcal{V}$ should not learn anything but the boolean outcome of the proof verification process.

Two further properties of proof systems describe the level of interaction between $\mathcal{P}$ and $\mathcal{V}$ in order to end up with a complete and sound proof outcome. One distinguishes between protocols that prescribe active communication and ones that do not, as stated below.

- **Interactive:** *Interactive* proof systems require live communication between $\mathcal{P}$ and $\mathcal{V}$ in the context of repeated challenge-response rounds with random variables.
- **Non-Interactive:** When $\mathcal{V}$ can verify a proof without the need to exchange messages with $\mathcal{P}$, one speaks of *non-interactive* proof systems. This is especially powerful as proving and verifying can be performed independently from each other without any form of required dialogue.

Generally, every interactive system can be transformed into a non-interactive one using the Fiat-Shamir transformation [18] in the so-called *Random Oracle Model (ROM)*. It assumes that both $\mathcal{P}$ and $\mathcal{V}$ have access to a random function h, which for an input x deterministically returns a random value $h(x) = y$. In practice, h is usually a cryptographic hash function.

The core of back-and-forth communication in interactive protocols lies in the randomly chosen challenges to which only an honest $\mathcal{P}$ can consistently respond successfully in the long run. To transform such protocols into non-interactive ones, $\mathcal{P}$ simulates the challenges by prompting h for random challenges. All prompts inside this ROM will be embedded in the proof such that $\mathcal{V}$ can verify the integrity of these prompts at any later point in time non-interactively. The security model holds as it would be unfeasible for a computationally bounded $\mathcal{P}$ to use h in a self-serving manner facilitating the task to generate proofs dishonestly.

Last but not least, proof systems also differentiate themselves from each other based on the following two remaining properties.

[1] DLP states that finding x for a given b in $a^x \equiv b \mod p$ is disproportionately harder than finding b for a given x [17].

- **Succinct:** A protocol that results in proof size and verification time that grows sublinear to the size of the statement being proven while upholding the security level is considered *succinct*. However, other definitions exist where authors associate *succinctness* with polylogarithmic or quasilinear growth.
- **Transparent:** Proof systems that do not require a trusted setup are considered *transparent*. For example, a trusted setup exists when the public parameters of the proof system are generated as a function of secret trapdoor. In cryptography, a secret trapdoor refers to piece of privileged knowledge[2] that facilitates access to restricted or secure functionalities.

Argument vs. Proof of Knowledge

There exists a slight but important difference in the quality of a system that aims to convince $\mathcal{V}$ that $\mathcal{P}$ knows a valid witness. This difference is addressed below.

- **Argument of Knowledge:** In an argument of knowledge system, $\mathcal{P}$ is known to be computationally bounded and thus only able to create proofs that rely on polynomial-time complexity. This allows $\mathcal{V}$ to assume that the underlying cryptographic assumptions hold. In other words, there is no way for a computationally bounded $\mathcal{P}$ to convincingly prove knowledge of x without *actually* knowing x. A computationally unbounded $\mathcal{P}$, however, could break the underlying cryptographic assumptions and thus generate convincing proofs without actually knowing a valid witness. Argument of knowledge systems are also known as *argument systems* or *computationally bounded proof systems*.
- **Proof of Knowledge:** A proof of knowledge, on the other hand, represents a stronger proving mechanism than an argument of knowledge since a computationally unbounded $\mathcal{P}$ would still fail to convince $\mathcal{V}$ with dishonest proofs despite the ability to break cryptographic assumptions. It follows that proof of knowledge systems possess higher security than argument systems, although the latter is more common in practice.

For the sake of simplicity, we continue to primarily utilizes the term *proof systems* in this thesis for both types of systems and uses the explicit specification where contextually meaningful.

2.2 Implementations

This section provides an overview of how ZKPs can be implemented using software. After briefly revisiting the STARK proving mechanics on a high level, we illustrate what we call the *direct* and *indirect* approach to implement this.

Two common categories or families of succinct ZKP systems encompass SNARK [19] and STARK [20] systems. In reality, the spectrum of different ZKP systems is more complex and out of scope for this thesis. We recommend the works by Ben-Sasson [21] and Tran [22] for more detailed insights into the classifications of such systems. For the sake of completeness, we summarized the key differences between SNARKs and STARKs in Table 2.1.

[2] Secret information generated during the setup phase of a cryptographic system that later could enable a potential adversaries to compromise the system's integrity is also known as *toxic waste*.

Aspect	SNARK	STARK
Proving complexity	$\mathcal{O}(n * \log(n))$	$\mathcal{O}(n * \log^c(n))$
Verifying complexity	$\mathcal{O}(1)$	$\mathcal{O}(\log^c(n))$
Proof size	$\mathcal{O}(1)$	$\mathcal{O}(\log^c(n))$
Setup	trusted	transparent
Post-quantum	insecure	secure

TABLE 2.1: Key Differences between SNARK and STARK
Shows the complexity aspects in relation to input size n [23]. Here, c represents a constant which results in *poly-logarithmic* growth in the case of STARK. One also refers to the proof size as *communication complexity* since it must be shared or communicated to $\mathcal{V}$.

STARK Primer

As stated earlier, the focus of this thesis is STARK proofs. The rationale behind this choice is due to two attractive properties. First, they do not require a trusted setup and are thus deemed transparent. Second, they only rely on secure cryptographic hash functions, which qualifies them as post-quantum secure [3].

In order to create a STARK proof for $\mathcal{S}$ with secret inputs x, $\mathcal{P}$ describes $\mathcal{S}$ through constraints that apply to all valid x. For example, let $\mathcal{S}$ = *«I know the start of a Fibonacci sequence* (x_0, x_1) *s.t.* $a_{1000} = A$*»*[4]. One distinguishes two constraint types:

- **Boundary Constraints:** Define the starting and ending conditions of a computation. This is crucial to enforce certain input and output values. Considering the Fibonacci example above, we have the starting conditions (1) $a_0 = x_0$ and (2) $a_1 = x_1$. In the end, a valid result is constrained by (3) $a_{1000} = A$. This totals to three boundary constraints.

- **Transition Constraints:** Define all valid or legitimate transitions from one state n to the next state $n + 1$. This enforces the desired relation between input and output values of a computation. In the Fibonacci example, the only transition constraint existing is defined as $a_n + a_{n+1} = a_{n+2}$, stating that every next element must be the sum of the previous two elements.

$\mathcal{P}$ executes the computations of $\mathcal{S}$ once to create a so-called *trace* or *execution* table, i.e., a detailed record of every involved register (columns) during the course of execution (rows) of the given computation represented as finite elements on a prime field. $\mathcal{P}$ extends the trace by evaluating an interpolated polynomial on a much larger domain, resulting in a Reed-Solomon code[5] for the trace.

Ben-Sasson et al. [20] defined this trace *extension* by means of a so-called *blow-up factor* k. Large k increases the prover complexity but reduces the verification effort for a given security level, and vice-versa [16]. The extended trace table and the identified constraints together are also known as *Algebraic Intermediate Representation (AIR)* of $\mathcal{S}$. Honest $\mathcal{P}$ will satisfy all constraints at any stage in their trace table.

[3] Also termed a *weak* cryptographic assumption since such hash functions are readily available. On the other hand, *strong* assumptions hinge on intractable problems, which erodes confidence in a system's security.
[4] The Fibonacci sequence is defined as $a_n = a_{n-1} + a_{n-2}$.
[5] Error-correcting codes, named after their founders Reed and Solomon [24], that add redundancy through polynomial-based techniques to facilitate error detection and correction.

For mathematical and computational purposes, all constraints will be transformed into polynomials. For instance, the boundary constraint (1) $a_0 = x_0$ becomes $P_1(x) = a_0 - x$ with $P_1(x_0) = 0$. This transformation enables validation of constraint satisfaction by ensuring that all constraints evaluate to zero. It is important to note that all constraints that define any valid x to $\mathcal{S}$ are public knowledge and must also be known by $\mathcal{V}$.

$\mathcal{P}$ then linearly combines the constraint-derived polynomials in a so-called *Compositional Polynomial (CP)*. By successively applying a set of mathematical operations[6] on the CP, $\mathcal{P}$ halves its degree until the point where the polynomial becomes a constant. $\mathcal{P}$ commits to every intermediate result using a Merkle tree. The same commitment scheme was also performed for the extended trace and the derived CP beforehand and all root hashes are sent to $\mathcal{V}$.

The verification process contains two parts. First, $\mathcal{V}$ validates the correctness and completeness of the underlying constraint system, i.e. verifying whether one talks about the same $\mathcal{S}$. Given this is the case, $\mathcal{P}$ is repeatedly asked to decommit certain values, allowing $\mathcal{V}$ to validate the computations that should lead to a constant. $\mathcal{V}$ will be convinced after statistically enough challenge rounds. As mentioned before, this protocol can be employed non-interactively using the Fiat-Shamir transformation in the ROM.

Direct vs. Indirect Proving Approaches

While the constraints in the above-mentioned Fibonacci example are rather trivial, this is not the case in more realistic ZKP use cases. In fact, the *direct* encoding of $\mathcal{S}$ as AIR can be extremely challenging. The reason for this lies in its individual character and the complexity to drive a correct and sound AIR from a given problem. Also, the individual character of AIRs aggravates reusability. Small changes to the underlying problem can necessitate major adaption in its AIR, which induces higher development costs. And even when a thorough encoding has been found, it may have significant implications on performance.

For clarity reasons, we state the following notation. Once more, let $\mathcal{S} =$ «*I know a secret* ω *s.t. for public input* x, $f(x, \omega) = 0$. We use the terms *executing* or *computing* $\mathcal{S}$ to refer to the evaluation of $f(x, \omega)$, while *proving* corresponds to the generation of the ZKP of $\mathcal{S}$. The specification of f, in that case, is what we denote as *proof logic*, as it defines the underlying logic for whatever needs to be proven.

In 2014, Ben-Sasson et al. [19] proposed an alternative to this *direct* approach by employing a RISC architecture as VM. Here, the computations of $\mathcal{S}$ are executed inside the VM. This VM serves as the execution environment for the proof logic, i.e., the circuits defining $\mathcal{S}$. The transitions between the (virtual) machine states is dictated by its ISA, which effectively functions as a set of constraints. The history of all states exhibited during a computation serves as trace, whereas the state initialization and all transitions are attested through the ISA. In other words, instead of proving $\mathcal{S}$ directly using a suitable AIR, one proves it *indirectly* using the register history of a VM that executed $\mathcal{S}$. Figure 2.1 on the next page illustrates both approaches on a high-level basis.

[6] In STARK [20] referred to as *Fast Reed-Solomon Interactive Oracle Proofs of Proximity (FRI)* protocol. Details of the FRI protocol are beyond the scope of this thesis.

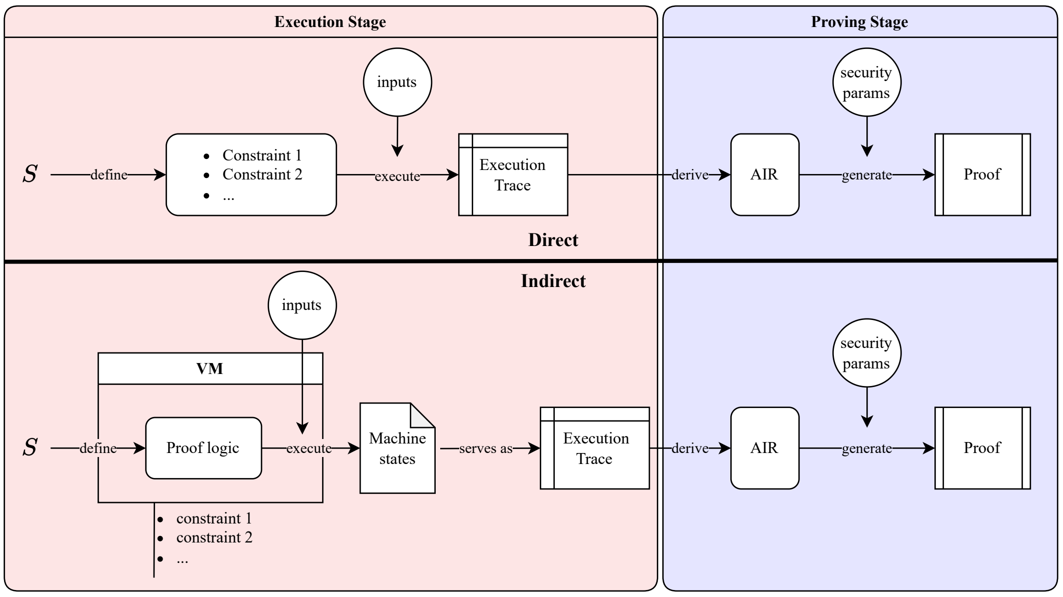


FIGURE 2.1: Direct vs. Indirect ZKP Approach
Other than proving $\mathcal{S}$ directly, it can be proven indirectly by executing it inside a VM. The time series of all machine states of that VM, i.e. the VM's execution trace of $\mathcal{S}$, serves as a basis for the subsequent STARK proof. The indirect approach allows for generalized constraints that are independent of $\mathcal{S}$, which is seen as advantage. A disadvantage, however, is the increase overhead and system complexity.

One significant advantage of such VM-based proving approaches is the generalization of the underlying AIR. As a result, one can generate ZKPs for any general computation, as long as it can be executed inside the VM. One therefore also speaks of *Zero-Knowledge Virtual Machine (zkVM)*, however, we continue to use the term VM for the remainder of this work. We refer interested readers to the work by Dokchitser and Bulkin [25] that explains in detail how to implement such zkVM.

On the other hand, this indirect approach introduces two disadvantages. Firstly, it amplifies the system complexity of ZKP frameworks. This increases the difficulty for both developers seeking to comprehend the underlying processes and prospective cryptographic audits. Both represent crucial aspects for deployments in productive environments.

The second disadvantage lies in the incurred computational overhead. VM initialization, for example, already increases the total number of clock cycles required to prove $\mathcal{S}$. On top of that, a single operation inside the proof logic may result in more than one cycle inside the VM due to its reduced ISA. It follows that a direct approach is more likely to be computationally efficient than an indirect one for $\mathcal{S}$ with a small number of constraints.

Since many ZKP use cases involve hashing, researchers also investigate into the developed of ZKP-friendly hash functions Ben-Sasson, Goldberg, and Levit [26] and BaarkingDog [27] that aim for a more efficient AIRs. However, their detailed examination falls outside the scope of this thesis.

Framework	VM	Proof Logic	License	References
Miden	yes	Miden ASM	MIT	[28]
RISC Zero	yes	Rust / C++	Apache-2.0	[29, 30]
Stone Prover	no	Cairo0	Apache-2.0	[31, 32]
Triton	yes	Triton ASM	Apache-2.0	[33]
Valida	yes	C	Apache-2.0 & MIT	[34]
Winterfell	no	Rust	MIT	[35]
Zilch	no	ZeroJava	MIT	[36]

TABLE 2.2: STARK-based ZKP Frameworks
Lists a collection of promising projects alongside their license and references. Cases that rely on VMs imply an indirect proving approach. Proof logic specified the circuit formulation language used in the framework.

Existing ZKP Frameworks

We dedicate the remainder of this section to an overview of existing ZKP frameworks implementing a STARK system. A collection of the most promising ones are shown in alphabetical order in Table 2.2. The «VM» column indicates whether the framework follows a direct (no) or indirect (yes) approach. The «Proof Logic» column specifies the language used to formulate the underlying $\mathcal{S}$.

Miden and *Triton*, for example, require proof logic formulations in a language similar to *Assembly (ASM)*. ASM serves as a low-level bridge between human-readable instructions and the raw commands for the *Central Processing Unit (CPU)*. Through mnemonic codes[7], it translates fundamental operations like arithmetic, logic, and data movement into instructions the CPU can execute.

Other frameworks, such as *Valida* or *Zilch*, allow to formulate proof logic using a higher-level language, similar to or derived from popular programming languages. The resulting formulations are eventually compiled into ASM-based instructions. *Stone Prover*, on the other hand, introduces CairoZero or *Cairo0*, a programming language designed explicitly for formulating ZKPs.

RISC Zero and *Winterfell* allow developers to specify the proof logic more naturally using established languages such as Rust, or in the case of the former also `C++`. Relying on general-purpose programming languages introduces two advantages. The first one concerns the existing community that fosters further development in that particular language independent from any ZKP frameworks. The other lies in the high accessibility for developers in that space, as they can benefit from previous experiences with these languages. This language-conformity, combined with the ability to prove general computations, thanks to the indirect approach, were the two determining reasons for our focus on RISC Zero in this thesis.

[7] For example, the 32-bit register `EAX` stores the value 2 after executing «`MOV EAX, 5`» and «`SUB EAX, 3`».

3 Risc Zero Framework

RISC Zero or `risc0` is a ZKP system for general-purpose computations. Applying our previously stated notation, it aims to create STARK proofs for any $\mathcal{S}$ whose proof logic can be executed or computed inside their VM. The produced history of chronological machine states acts as execution trace, attesting the computation's integrity in a verifiable manner. More specifically, the VM emulates a *RISC-V*[1], i.e., an open-source reduced ISA standard. It allows $\mathcal{P}$ to keep the witness of $\mathcal{S}$ secret and thus prove in zero-knowledge. This chapter introduces the RISC-V architecture to the readership, followed by elucidations on the proving and verifying process with `risc0`.

3.1 RISC-V Instruction Set Architecture

RISC-V was designed with simplicity and modularity in mind. Unlike *Complex Instruction Set Computing (CISC)* architectures, RISC architectures aim to streamline the instruction set to a minimal set of simple and orthogonal instructions. This modular design philosophy enhances the efficiency of instruction execution, making it easier to optimize compilers and design efficient hardware. What started as a project at UC Berkley [37] has gained popularity and is available in its fifth version (hence the 'V' in the name).

The RISC-V ISA is based on a fixed number of general-purpose registers and a set of instructions, each designed to perform a specific, simple operation. This simplicity facilitates more accessible hardware (or VM) implementation and promotes a more straightforward understanding of the instruction set. Its standard or base module exists in 32, 64, and 128-bit architecture. For example, the `RV32I` base integer instruction set refers to the 32-bit version and features 32 general-purpose registers. In its embedded version (`RV32E`), the number of general-purpose registers is reduced to 16 for efficiency purposes.

Thanks to the modular concept, a base variant can be extended to enable additional functionalities such as integer multiplication/division (`M` code), floating-point operations (`F` or `D` code for single resp. double precision), and many more. For a deeper understanding of other RISC-V variants, we encourage readers to explore the official RISC-V manual [38]. Additionally, for more foundational insights into the instruction processing in `RV32I`, we recommend the excellent book by Winans [39] which is available in draft form.

The VM in `risc0` implements the `RV32IM` instruction set, i.e., the base set `RV32I` extended with the `M` module. It operates in a single-threaded environment without preemption[2]. In the event of an exceptional occurrence, for instance, a misaligned memory access, the VM simply terminates execution without executing any exception handlers. Also, it follows a strict execution order to ensure sequential consistency.

[1] Pronounced as «risc five».
[2] Preemption refers to the operating system's ability to interrupt and switch tasks in execution.

3.2 Proving & Verifying Process

Now that we understand the VM's architecture used in `risc0`, this section discusses the proving and verifying process to complete the picture. Bruestle and Gafni [29] elaborates the technical and mathematical aspects in greater detail. However, it is still a work in progress.

We summarize the end-to-end process using our own notation in Figure 3.1, focusing on the high-level information exchange between the subprocesses or stages. This involves, at the beginning, the execution stage (red), followed by the proof stage (blue), and finally, the verification stage (green).

Execution Stage

In the beginning, $\mathcal{P}$ defines the proof logic L for a particular $\mathcal{S}$. `risc0` allows to specify L in Rust or `C++` and also refers to it as *guest method*. Guest methods qualify as public information and every $\mathcal{V}$ possesses its own copy L'.

We further denote the *witness* ω as suitable input for L. It is important to note that L is not entitled as private information and must be known by any verifying party. On the other hand, ω must remain private in order to accomplish a ZKP. However, it is also possible to fully or partially disclose ω to $\mathcal{V}$ if desired. We revisit this aspect later in an example.

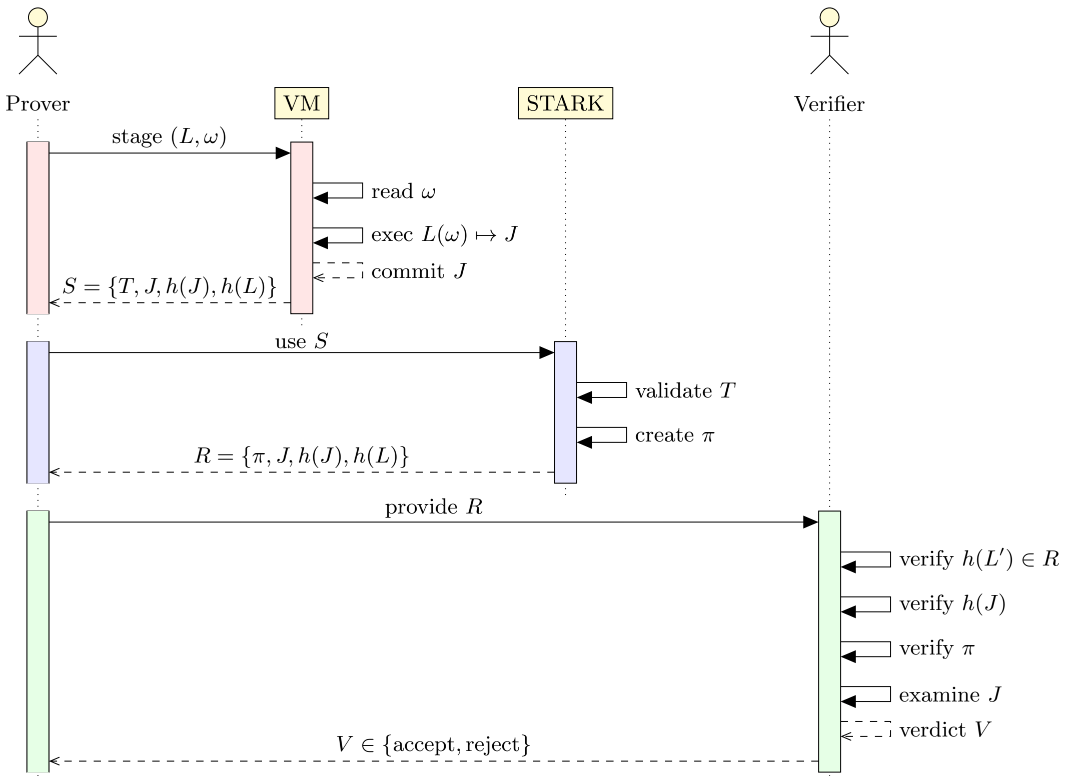


FIGURE 3.1: Sequence Diagram of `risc0` Proof & Verify Process
Visualized the information flow during execution (red), proof (blue), and verification (green) stage.
The objects VM and STARK represent the relevant software components inside the `risc0` framework.
While the specified order is crucial, the individual stages may occur out of sync.

$\mathcal{P}$ stages (L, ω) onto the VM which, after its initialization, executes L using ω as input. In practice, `risc0` compiles L into an *Executable and Linkable Format (ELF)* file in order to execute it using the `RV32IM` instruction set. The result of $L(\omega)$ can be written or committed to the so-called *journal* J. L may also commit any intermediate or final result(s) to that journal such that $\mathcal{V}$ can examine them later. Although committing values to the journal is not a mandatory step for the proving process, it is usually pursued to some extent to allow the validation of a computation's outcome.

Consider a modified Fibonacci sequence with $x_0 = 1$ as the first element. Assume we would like to prove that we know a second element x_1 such that the tenth element is $x_9 = 259$. The guest method in Listing 1 exemplifies how a corresponding L for this particular $\mathcal{S}$ could look like.

Listing 1 Example Guest Method in Rust

```
1  pub fn main() {
2      let mut x1: u32 = env::read();    // secret second element
3      let mut x0: u32 = 1;              // known first element
4      for _ in 2..10 {
5          let tmp = x0 + x1;
6          x0 = x1;
7          x1 = tmp;
8      }
9      env::commit(&x1);                 // commit result to journal
10 }
```

After initialization, the guest method reads the (previously staged) private ω on line 2. In order to conclude with the target of $x_9 = 259$, the required private information in this case is $x_1 = 7$. The first element (line 3) as well as the number of iterations (line 4) are hard-coded. However, they could also be passed as additional (public) ω. We commit the result of this computation, namely x_9, at the end to J (line 9) to allow its examination during the verification stage. As we are interested in a ZKP, we do not provide any information about ω in J, albeit technically possible. Note that `risc0` employs the verb *committing* here to denote the act of providing or writing information to J. This usage deviates from the conventional interpretation of the verb in the realm of cryptography.

We receive the so-called *session* $S = \{T, J, h(J), h(L)\}$ as a result of this execution phase[3]. It subsumes the following objects:

- **Trace** T: The execution trace, i.e., all encountered VM states during the execution of $L(\omega)$. The more clock or execution cycles our program $L(\omega)$ requires to execute, the longer T becomes.
- **Journal** J: The object containing committed intermediate or final results from our computation $L(\omega)$. Allows $\mathcal{V}$ to examine relevant parameters, intermediate or final results.

[3] Note that the session S differs from our notation for statements $\mathcal{S}$.

- **Digests** $h(J)$ **and** $h(L)$: The hash values of J and L using the a cryptographic hash function (`SHA2-256` by default). They enforce the infeasibility for undetectable post-execution alteration to either J or L. `risc0` also refers to $h(L)$ as *ImageID*.

Proof Stage

S serves as a basis for generating the actual STARK proof π. Beforehand, `risc0` checks the validity of T, i.e., all state transitions align with the constraints derived from the `RV32IM` instruction set. The framework nicely abstracts this stage using one line of code that returns the so-called *receipt* R. It is used to propagate J, $h(J)$, and $h(L)$ from S alongside π to any prospective $\mathcal{V}$.

R is considered public information and will be exchanged as a whole for the subsequent verification stage. It is by default stored as binary file but can also be exchanged in a more human-readable format through `JSON` serialization. The latter, however, is less storage efficient. Also, it is essential that $T \notin R$ as it would allow inferences about ω, which is unwanted in a ZKP. `risc0` refers to π also as the *seal* of R since it attests the computational integrity of J using the STARK protocol.

As for the execution stage, `risc0` expresses the proof's computational complexity in cycles. Figure 3.2 illustrates this execution-prove-cycle relation over a few selected observations. An observation, in that sense, comprised performing the execution and proof stage for an arbitrary $\mathcal{S}$.

The cycle count is shown in logarithmic scale over some selected observations with growing trace length. The minimal required amount of proof cycles in `risc0` is 2^{16} due to VM initialization and clean-up. This overhead together with the fact that some operations in the `RV32IM` ISA required more than one cycle to complete makes it difficult to pre-determine the amount of proof cycles required for a given $L(\omega)$. What is determinable, however, is that the proof cycle count always pads to a higher power of two as it allows a more efficient Reed-Solomon encoding. This padding explains the step-wise growing pattern of the proof cycle data series noticeable in Figure 3.2.

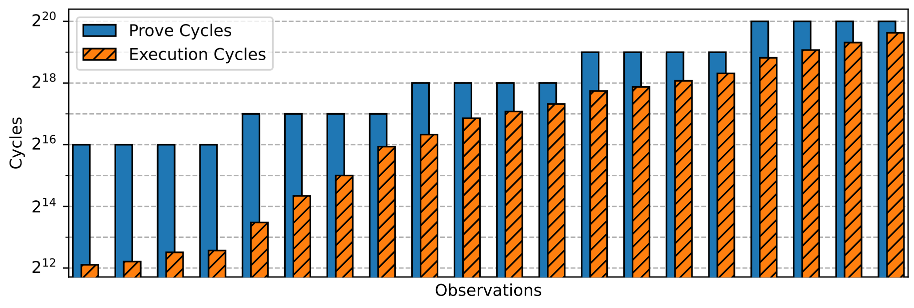


FIGURE 3.2: Execution-Prove-Cycle Relation
Illustrates the relationship between execution and proof cycles in `risc0`. The latter is derived from the execution cycle count, consistently representing a padded value to a higher power of two, but at minimum 2^{16} due to VM initialization and finalization. This padding introduces a discernible step-wise growth pattern. We intentionally avoided using one of the data series as the x-axis tick to enhance pattern clarity.

The risc0 framework automatically splits larger $L(\omega)$ executions into so-called *segments*. By default, segments accommodate $\leqslant 2^{20}$ cycles and are paginated such that a next segment starts with the VM state where the previous segment terminated.

The segmentation feature, termed *continuation*, is a promising tool for enhancing proving efficiency in risc0 [40]. This encompasses, for example, parallel segment proving and the ability to control *Random Access Memory (RAM)* consumption by defining user-specific segment sizes. For this thesis, we adhere to default settings, reserving exploration of these parallelization capabilities for future research.

Verification Stage

So far, we elucidated the execution and proof stage which is both conducted by $\mathcal{P}$. Since this involved a considerable amount of different objects, we recall the meaning for our letter notations to enhance clarity.

The verifying party possesses its own copy of the proof logic L, which we denote here as L'. The verification process contains the following four steps, whereas the outcome of every step must be positive to justify proceeding to the next one.

1. Verify that receipt R was indeed generated on the basis of L', given that $L = L'$. When $h(L') \notin R$, the presented proof is subject to a different (not pre-consented) $\mathcal{S}$.
2. Verify that no post-execution alterations of journal J exist by recomputing $h(J)$.
3. Verify the actual STARK proof π.
4. Examine the information committed to J and derive the final verdict V from it[4]. In our Fibonacci example from Listing 1, we would still reject R if $x_9 \neq 259$.

[4] Note that the verdict V differs from our notation for verifier $\mathcal{V}$.

4 ZKit Test and Benchmark Suite

The previous chapters contributed to a common base of understanding for the readership. We delved into ZKP systems, with an emphasis on the STARK protocol and illustrated how `risc0` enables us to indirectly prove general-purpose computations in zero-knowledge using its VM. In this chapter, we introduce the system architecture and functionalities of `ZKit`, our extensible test and benchmarking suite for ZKP frameworks.

4.1 System Architecture

We developed `ZKit` predominantly in Rust[1]. This relatively young programming language has experienced a notable surge in popularity over the past few years [41]. In this section, we briefly underscore commendable features of Rust and explain the rationale behind selecting this language. Subsequently, we elucidate our design decisions in `ZKit` and describe the purpose of all containing components. Figure 4.1 provides an overview of the system architecture.

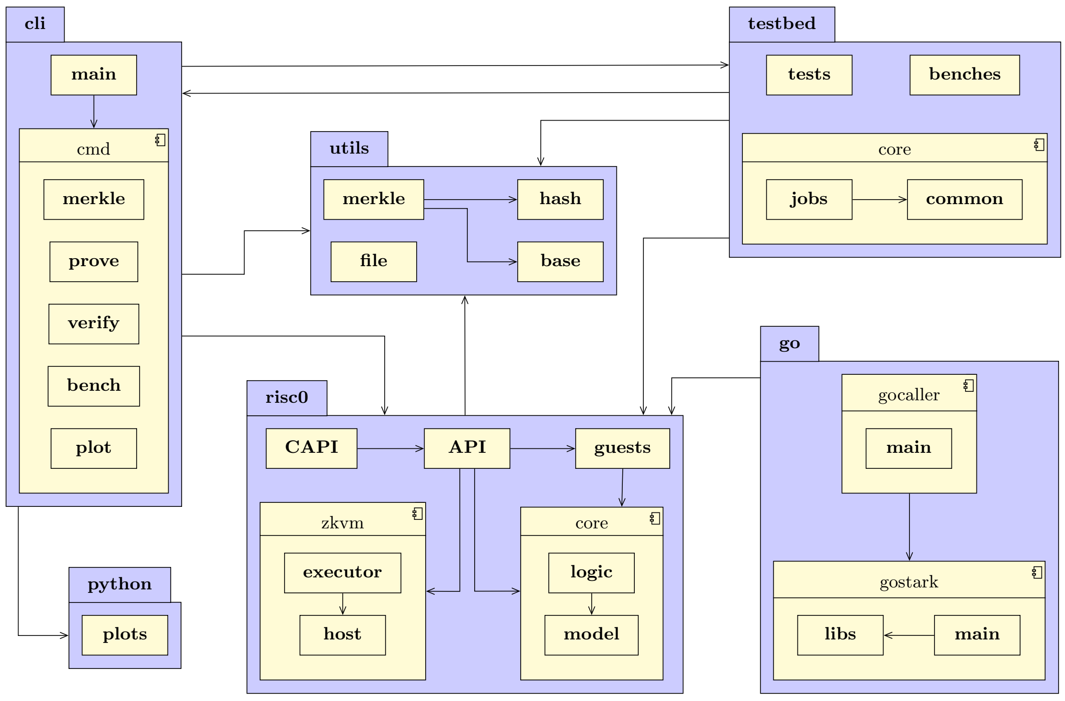


FIGURE 4.1: `ZKit` Component Diagram
Visualizes the interdependencies between packages (blue).
Each package contains several modules or module groups (yellow).

[1] More specifically, we use the *nightly* channel or version of Rust to benefit from latest features. See Rust's homepage and documentation for details.

Why Rust

The main reason for choosing Rust as predominant programming language in `ZKit` is that most promising ZKP frameworks are built and/or compatible with Rust. This applies, for example, the majority of the projects listed in Table 2.2 (vic. Miden, `risc0`, Triton, Valida, Winterfell). Therefore, we chose to build `ZKit` using the same language to maximize interoperability. Besides this, Rust also supports a feature called *Foreign Function Interface (FFI)*, enabling cross-language operability. We utilize this feature in our Go-wrapper endeavour. Details on that follow later in this section.

To better understand the prevalence of Rust in this cryptographic domain, we emphasize two notable aspects [42]. One of the key factors contributing to Rust's popularity in cryptography is its emphasis on memory safety. Similar to `C++`, for example, pointer-based development is encouraged. However, Rust's so-called *ownership* and *borrowing* system prevents common programming errors such as null pointer references. It is enforced on compile-time and allows Rust not to have a garbage collector, which positively affects run-time performance. The resulting confidence in correctly operating code is especially desired in critical systems, such as in human-machine interactions (aircraft, elevators, etc.).

The second compelling aspect concerns performance. Its zero-cost abstractions and memory allocation control allow to optimize performance-critical systems. This feature is particularly beneficial in cryptographic applications where computational efficiency is essential. Overall, Rust is a memory-safe, multi-paradigm, high-performant, and developer-friendly programming language ideal for applications aiming for robustness and security.

Components & Dependencies

We structure the different components of `ZKit` in packages, which further contain modules. Figure 4.1 visualizes all components and their interdependencies, indicated through arrows. The □ symbol declares a module group. Unless explicitly specified, we handle modules in the same manner as module groups in our explanations.

In this subsection, we state each package's intention and describe their dependencies. The chosen presentation order is designed to provide an explanation without requiring anticipation in each instance, ensuring a coherent narrative. Where deemed relevant, we mention dependencies to other external or third-party libraries. Libraries in Rust ecosystem are commonly referred to as *crates*. Such external crates, however, are not modeled in Figure 4.1 for the sake of better readability.

utils contains general functions with a high potential for reusability across `ZKit`. The hash module provides abstractions for various cryptographic hash functions to call them generically. Functions concerning random value generators or encoding translations (e.g. binary to hexadecimal) are located in base. All functionality related to constructing Merkle trees and IPs are contained in the merkle module. Besides consuming hash and base functionalities, we rely on the third-party Merkle tree implementation called `rs-merkle` in the background. As the name suggests, file provides functionalities for file operations (e.g. read/write access).

risc0 bundles everything related to the eponymous ZKP framework at focus in this thesis. We define all the proof logic in the core module group, differentiating between object structures in model (the *what*) and processing in logic (the *how*). The guests module consumes this core eventually to define the required guest methods. We import the zkvm module group from the `risc0` framework to instantiate and host VM instances used to executed these (guest) programs. The API module exposes the interface for executing and verifying ZKPs. CAPI wraps the relevant functionalities from the API in a FFI conform manner, making them accessible to other programming languages.

testbed represents the core of our test and benchmarking suite. In tests and benches, we ensure integrity and analyze the performance of non-ZKP-related functionality in `ZKit`. This includes, for example, functionality from utils or cli. Both modules are designed exclusively for *Continuous Integration (CI)* purposes and are intended to be excluded in release builds. The core module group contains the ZKP-related benchmarks, also referred to as jobs. Job commonalities, such as sample data, interfaces, or input sizes (also known as *magnitudes*), are normalized in common. These benchmarks are designed to be included in release builds as they ship alongside the CLI.

cli serves as main user interaction tool. We can create Merkle tree data structures and IPs for specific leaves using the merkle module. The prove and verify modules allow the creation and verification of ZKPs. At the moment, the CLI solely communicates to the API in risc0. This communication will expand to other *Application Programming Interfaces (APIs)* by extending `ZKit` to further ZKP frameworks. Lastly, bench contains the functionality to trigger parameterized benchmark jobs and visualize the results using plot. Automated plotting, however, is still under development at the time of writing. Note that all module names correspond to first level CLI command names.

python entails relevant functionality to visualize benchmark results using Python. As the name suggests, this is one of mentioned exceptions where we do not use Rust. Although plotting crates exist in Rust, we discovered them as less convenient and powerful than established libraries in Python.

go acts as a proof of concept to port the implemented ZKP functionality written in Rust to Go. We compile risc0 into a dynamically linked library and embed it with a suitable header file inside gostark, i.e., the Go wrapper. On the Go side, we re-implement the interface specified in that header file to use the ported library through its FFI. Eventually, we can use the Go wrapper identically as every other Go library by simply importing and calling the available functions. We demonstrate this in gocaller, that imports and consumes gostark in a pure Go manner. The following subsection elaborates on this porting and wrapping strategy in detail.

Go-Wrapper

As mentioned above, Rust supports interoperability with other programming languages through its FFI. We use this feature to port the ZKP functionality to our Go wrapper. Our motivation for portability is to facilitate the integration of ZKPs in existing Go projects using minimal efforts. In particular, we anticipate the integration in the FTS. The following paragraphs elucidate our approach to support the overall comprehensibility of future developers.

The official package manager and build tool in the Rust programming languages is called *Cargo*[2]. It allows to specify configuration settings and metadata for a given Rust project in the so-called `Cargo.toml` file, which resides in the project's root folder. In other software development technologies, one would also refer to such a file as *manifest*. One particular attribute in this manifest concerns the `crate-type`. It specifies the types of output artifacts that should be generated during the compilation of a Rust project. These artifacts include, for instance, static/dynamic libraries or executables.

In order to port the ZKP functionalities to Go, we additionally build the risc0 package with the crate type `cdylib`, which stand for «C dynamic library». As stated above, we exposed the relevant interfaces to create and verify ZKPs in the CAPI module using the FFI functionalities. The last thing to consider before we can consume the functionality of the C dynamic library in Go is the interface's declaration in a header file[3]. We created the compatible header file manually, however, one can derive them automatically using a tool called *cbingen* [43]. Finally, both the dynamic library and the header file are referenced in a multi-line comment at the beginning of the invoking Go file (e.g. `main.go`).

Figure 4.2 summarizes the required steps to access functionality developed in Rust (R) from Go (G). The step prefixes identify to which language the instruction below apply to:

R1 Define public API using the FFI.

R2 Declare the desired dynamic library build target in the manifest.

R3 Build the project to generate the artifacts.

G1 Embed generated dynamic library from (R3) in the Go package's source.

G2 Provide a compatible header file, declaring the interface from (R1).

G3 Reference the library in the required header comments and consume the functions through the FFI.

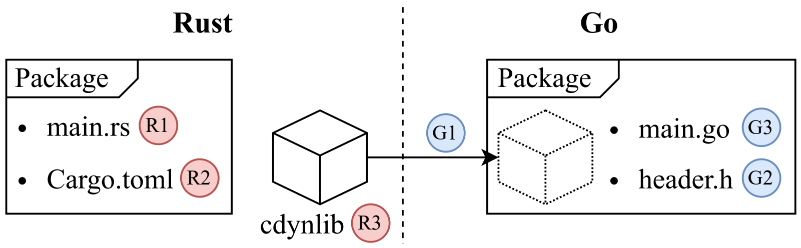
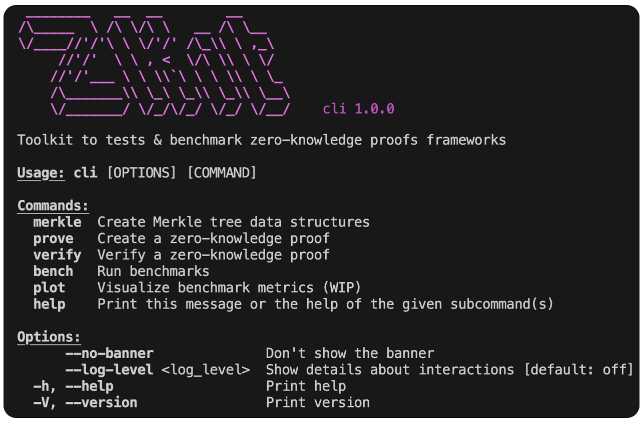


FIGURE 4.2: Rust to Go Compile Process
Shows a generic high-level model on how to port Rust (R) packages to Go (G).
Details for the steps R1 to R3 and G1 to G3 are stated in text above.

[2] See the Cargo Guide for more details.
[3] Used to share code definitions between source files in `C/C++` programming.

4.2 CLI Functionalities

The primary mean to interact with `ZKit` is the CLI tool, shown in Figure 4.3. We can use it to create and verify ZKPs for the two different use cases or *disciplines* that are already implemented. The first discipline is called *preimage*. It allows $\mathcal{P}$ to prove knowing an input to a cryptographic hash function for a public digest, without revealing this input. The second discipline concerns IPs and is thus called identically (*ip*). It allows $\mathcal{P}$ to prove knowing a certain leaf data of a Merkle tree with public root hash.



```
                                   cli 1.0.0

Toolkit to tests & benchmark zero-knowledge proofs frameworks

Usage: cli [OPTIONS] [COMMAND]

Commands:
  merkle  Create Merkle tree data structures
  prove   Create a zero-knowledge proof
  verify  Verify a zero-knowledge proof
  bench   Run benchmarks
  plot    Visualize benchmark metrics (WIP)
  help    Print this message or the help of the given subcommand(s)

Options:
      --no-banner              Don't show the banner
      --log-level <log_level>  Show details about interactions [default: off]
  -h, --help                   Print help
  -V, --version                Print version
```

FIGURE 4.3: Screenshot of the `ZKit` CLI
Shows the help screen, listing the available first-level commands.
Each command further structures into subcommands.

Although hashing is an integral component to prove the inclusion of a leaf in a Merkle tree, we examined this discipline separately for two reasons. First, proving a preimage is a common showcase discipline in many ZKP frameworks thanks to its simplicity. Second, examining different hash functions atomically provides valuable insights for better design decisions in the more complex IPs discipline. More details on these design decision will follow in the next chapter, where we present our benchmark results.

The remainder of this section provides an overview of the CLI's functionality by means of the IP discipline. The user experience in more trivial preimage discipline is analogical to it and thus not particularly discussed.

Merkle Tree Data

For a start, the CLI allows the generation of Merkle tree data structures for custom depths. One can provide the leaf data explicitly or rely on random generators. More interesting than an entire tree is generating an IP for a distinct leaf. The Bash[4] command shown in Listing 2 below accomplishes this.

Listing 2 Bash Command to Create IP

```bash
cli merkle prove -e "alice bob paul vanja" -l "bob" -a sha256 -o ip.json
```

[4] Unix command language interpreter.

It first generates the entire Merkle tree with the four specific leaf elements (`-e`) using the `SHA2-256` hash algorithm. Once the root hash and all intermediate node hashes are known, it proves the inclusion of the leaf «bob». As result, we receive the IP as a JSON output (`-o`).

Listing 3 exemplifies the content of the constructed IP. It structures in two parts. The first part contains all relevant information to recompute the root hash of the Merkle tree. This contains the hash function to be used (line 1) and the root hash itself (line 2). We represent all digests in hexadecimal format to support human readability, however, they are processed internally as byte arrays. The leaf index (line 3) states the exact position of the provable leaf among all other leaves. Here, «bob» is the second leaf, which corresponds to the zero-based index 1. To verify this IP, we recompute the hash value of the actual leaf data (line 4) and then continue with successively concatenating the intermediate node hashes that define our authentication path (line 6–7). This array of intermediate node hashes is stated in the expected order, i.e., from first (top) to last (bottom).

Listing 3 Example IP as JSON

```
1   "hash_function": "sha256",
2   "root_hex": "01e94053710c6b7fa55a97f76cab16d1040639ca6c3d6748449798772e6b229d",
3   "leaf_index": 1,
4   "leaf_data": "bob",
5   "auth_path": [
6       "2bd806c97f0e00af1a1fc3328fa763a9269723c8db8fac4f93af71db186d6e90",
7       "f802fe0db602361c5ef043b1f2f280ff3c13c25d0b72bf2e0dc701a928f47742"
8   ],
9   "meta_data": {
10      "leaves": [ "alice", "bob", "paul", "vanja" ],
11      "leaves_digest": [
12          "2bd806c97f0e00af1a1fc3328fa763a9269723c8db8fac4f93af71db186d6e90",
13          "81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce9",
14          "0357513deb903a056e74a7e475247fc1ffe31d8be4c1d4a31f58dd47ae484100",
15          "52fd7e6e755e924643b2304102f15f71b64c38ed9f7134ef3e1f84160b5b5e2c"
16      ],
17      "ap_size": 2
18  }
```

The second part of the IP data structure in Listing 3 carries optional meta data. This includes all original leaf data (line 10) and their digests (line 12–15), in respective order. At the end, we state the authentication path size (line 17), which in this case corresponds to the tree depth.

Neglecting the optional meta data, we defined this IP data structure in the most minimal possible way. This presumes, however, that the underlying Merkle tree qualifies as a *perfect binary tree*, i.e. trees with 2^n leaves. This particular tree class allows to derive the concatenation order of current and intermediate hash solely from the leaf index. Otherwise, every intermediate node hash must be accompanied explicitly with its concatenation direction.

The reason for this minimal IP interface design is, that it allows a more efficient guest method implementation for the actual ZKP. Generally, the simpler the execution of a guest method with respect to a given input, the smaller the resulting trace and the faster the subsequent STARK proof.

Algorithm 1 below illustrates the verification process used in the guest method, adapted from [3, Chapter 4.2]. It consumes three objects as input, namely the data d and index of the provable leaf d_i, as well as the array of intermediate node hashes P_{auth} acting as authentication path. Thanks to the perfect binary tree structure, we can replicate the all correct concatenation directions solely from d_i using bitwise `AND` operations (&) followed by right bit shifts ($\gg$) to discard the least significant bit. The IP succeeds when the recomputed root hash H_R^* equals the expected or target root hash. `ZKit` implements this algorithm accordingly.

Algorithm 1 Inclusion Proof Verification in Perfect Binary Tree Situations

```
Input: Leaf data $d$, leaf index $d_i$, intermediate node hash array $P_{auth}$
Output: Recomputed root hash $H_R^*$
1:  procedure RecomputeRootHash ($d$, $d_i$, $P_{auth}$)
2:      $H \leftarrow h(d_i)$                          ▷ create leaf node hash
3:      $n \leftarrow len(P_{auth})$                   ▷ get authentication path size
4:      $i \leftarrow d_i$                             ▷ initialize concatenation direction indicator
5:      for $H_p \in P_{auth}$ do
6:          $lastbit \leftarrow i \& 1$                ▷ bitwise AND operation
7:          $i \leftarrow i \gg 1$                     ▷ right bit shift, discarding least significant bit
8:          if $lastbit = 1$ then                      ▷ decision on concatenation order
9:              $H \leftarrow h(H \mid H_p)$           ▷ right concatenation
10:         else
11:             $H \leftarrow h(H_p \mid H)$           ▷ left concatenation
12:     $H_R^* \leftarrow H$                           ▷ last hash value represent root hash
13:     return $H_R^*$
```

What is not shown in Algorithm 1 for brevity reasons are the hash function selection and the comparison of H_R^* against the specified target root hash from the IP file. In our guest method implementation, we commit H_R^* alongside the comparison outcome to the journal for examination purposes at verification time.

Proving

The in Listing 3 presented IP file serves as input for the actual STARK proof, as shown in Listing 4. We therefore specify the ZKP framework to be used for the proof (`-f`). At the time of writing, this parameter can be omitted since `risc0` is the only framework available in `ZKit` and thus assumed by default. We point to the previously generated IP file that serves as sole input (`-i`) and specify a file name to save the STARK proof (`-o`). In the case of `risc0`, this output is the receipt R (see Figure 3.1).

Listing 4 Bash Command to Create ZKP of IP

```
cli prove ip -f risc0 -i ip.json -o mystarkproof
```

This command triggers the following actions behind the scenes. First, the CLI parses the content from the IP file (see Listing 5) into two objects, vic. `PrivateInput` and `PublicInput`. The `PrivateInput` subsumes the leaf data, leaf index, and the array of intermediate node hashes. As the object name suggests, these information must remain secret and will not be shared with any $\mathcal{V}$. The `PublicInput` subsumes the required hash function name and target root hash, both considered public knowledge that is shared with $\mathcal{V}$.

Next, both objects are propagated to the risc0 package using the API module. What follows are the execution (red) and proof stage (blue) described in Figure 3.1, whereas the latter represents the time- and resource-expensive operation. The resulting receipt R is returned to the CLI package which coordinates its serialization into a binary file.

We emphasize that the distinction between public and private input using designated objects is not strictly necessary. Alternatively, one could directly serialize the IP as an object instead of transforming the contained data fields into two separate objects. However, we made a deliberate choice to define the API in this explicit manner for two reasons. Firstly, it enhances intuition regarding what constitutes secret information that should not be shared with any other party. Secondly, the two objects may be composed in different processes running asynchronously. For example, the `PrivateInput` can be composed directly on $\mathcal{P}$'s device, while the `PublicInput` is sourced from another (public) system. The latter may also occur less frequently.

Verifying

The command shown in Listing 5 will verify the previously generated R. It requires pointing to the binary file R and specifying the expected root hash (`-r`).

Listing 5 Bash Command to Verify ZKP of IP

```bash
cli verify ip -i mystarkproof.bin \\
    -r 01e94053710c6b7fa55a97f76cab16d1040639ca6c3d6748449798772e6b229d
```

It executes the described steps in the verification stage (see green process in Figure 3.1). The provided root hash is employed to ascertain the final verdict V. Other than the proving command, this verifying a STARK proof concludes in milliseconds. In a future productive setting, one might transmit R over a potentially insecure communication channel from the proving party to the verifying party. Nevertheless, owing to its computational efficiency, the proving party can effortlessly self-verify the generated proof before transmission.

Benchmarking

Besides creating and verifying ZKPs, the CLI serves also the purpose to measure their time and resource consumption by means of benchmark jobs. In this sense, a job is defined as a distinct instance of a discipline, featuring various flavors and executable with diverse input sizes. As earlier stated, we refer to a set of input sizes as magnitudes. To facilitate subsequent analysis of benchmark results, magnitudes are categorized into five classes, namely `XS`, `S`, `M`, `L`, and `XL`.

For example, different hash functions or implementation strategies could act as flavors. Magnitudes, on the other hand, could refer to different input byte sizes or – in the case of Merkle trees – tree depths. More details on the existing jobs in `ZKit` with the concrete flavor and magnitude specification follow in the next chapter.

Listing 6 demonstrates how to execute all predefined jobs in all flavor variants (`-j`) for all magnitudes (`-s`). Benchmark results are written as comma-separated values to the specified output file (`-o`).

Listing 6 Bash Command to Execute Benchmarks

```
cli bench -j all -s all -o result.csv
```

Note that the execution time for the aforementioned command may range from several minutes to a few hours. To enhance user experience, we incorporated a progress bar and provide a `--dry-run` option, allowing users to obtain a quick overview of the *Job-Flavor-Size (JFS)* multiplicity. The following list outlines the attributes and metrics gathered for all benchmarks.

- **Machine:** Identifies the host computer that executed the benchmarks. This is especially useful when working with `ZKit` on multiple devices or virtual environments.
- **Epoch:** To receive confident results, benchmarks should be executed repetitively in the same setting. We use a time stamp to mark and therewith group results from each run or epoch.
- **Framework:** Specifies the underlying ZKP framework used for a given job. At the moment, the only available one is `risc0`.
- **Job Name:** Identifies the concrete job using a hard-coded name.
- **Job Flavor:** Declares the variant of the job. Variants are also hard-coded as they are highly job-specific. For example, in the preimage discipline, we use different hash functions as job flavors.
- **Job Size:** Specifies the input size for a given job. While this is always represented as an integer value, the meaning of this value is job-depended. For instance, this could be the number of bytes for a preimage job or tree depth in IP jobs.
- **Job Size Alias:** Translates the associated size into a magnitude class (`XS`, `S`, `M`, `L`, and `XL`). Facilitates subsequent data analysis and visualization. Similarly to the job size, this categorization is hard-coded due to its job-specific nature.
- **Execution Duration:** Indicates the required time to execute the proof logic inside the VM. Expresses in milliseconds.
- **Prove Duration:** Indicates the required time to create the STARK proof from the execution trace. Expresses in milliseconds.
- **Verify Duration:** Indicates the required time to verify a previously created STARK proof. Expresses in milliseconds.

- **Execution Cycles:** Number of clock cycles required to execute the proof logic for a given input size inside (either directly or indirectly via VM). The value of this metric positively correlates with the execution duration.
- **Prove Cycles:** The number of clock cycles required to create the STARK proof. This value always exceeds the number of execution cycles due to the blow-up factor and Reed-Solomon encoding. Identically to execution cycles, the amount of prove cycles positively correlates with the prove duration. We exemplified the execution-prove-cycle relationship in `risc0` previously in Figure 3.2.
- **Peak Memory:** Expresses the highest number of used RAM while executing of a specific JFS combination. This is accomplished by spawning a separate thread for each JFS execution that periodically measures the consumed memory and returns the all-time high.

4.3 Extensibility

The fundamental idea of `ZKit` is to serve as a test and benchmark suite for multiple ZKP frameworks. This section highlights three essential aspects to consider for future extension efforts.

General Integration

Future ZKP frameworks are meant to be integrated similarly to the risc0 package. This means, they contain all necessary modules for creating and verifying ZKPs and expose these functionaries through a package-specific API. Extending the cli involves consuming those additional API. Ideally, the CLI looks similar to the examples in Listing 4 or Listing 5, where we indicate the chosen ZKP framework using the `-f` argument.

A more complex endeavor, on the other hand, is an elegant way to reuse already existing proof logic (e.g. guests). The reason for this lies in the highly ZKP framework-dependent strategy or technique to define proof logic in the first place, as previously shown in Table 2.2 (column «Proof Logic»). For instance, envision incorporating Zilch as the next ZKP framework into `ZKit`, with the aim of utilizing it for the IP discipline. This integration would necessitate implementing Algorithm 1 in ZeroJava, the designated language for defining proof logic in Zilch. In cases where reusability is possible, we encourage encapsulating proof logic in a distinct (new) package, akin to utils.

Benchmark Disciplines

As outlined in Section 2.2, ZKP frameworks adopt diverse paradigms for encoding proof logic and may differ in their approaches to both direct and indirect proving implementations. This diversity may necessitates the implementation of benchmark disciplines in multiple versions or languages, resulting in potential redundancy. Established benchmark jobs, however, can quickly scale in variance through the chosen flavoring and magnitude approach.

Moreover, it may not be guaranteed that all future ZKP framework extensions uniformly support the currently defined benchmark metrics. In cases where obtaining metrics such as the number of execution or prove cycles is unfeasible, we suggest to constrain the subsequent result analysis exclusively to runtimes.

Ports to Go

So far, the presented Go wrapper serves the purpose to port the CAPI of the risc0 package to Go. In case porting functionality to Go is also required for additional ZKP frameworks, we suggest two options.

The first option entails replicating the go package and managing two separate compile pipelines. This approach offers dual benefits. One is that each wrapper exclusively encapsulates functionalities from a specific ZKP framework, enabling more precise utilization. The other concerns the independent compile pipelines, allowing for more efficient CI processes. The disadvantages that this first option introduces, however, is the duplication of boilerplate code due to the FFI, which necessitates semi-automatically maintenance.

The second option involves integrating the APIs of multiple ZKP frameworks in one single go package. While this allows to minimize boiler plate code, it would increase the complexity and size of the Go wrapper. Also, the integrity of larger packages that bundle several different modules becomes more fragile compared to smaller more encapsulated packages that are easier to maintain as a whole.

5 Benchmark Results

This chapter presents the collected benchmark results focusing on the `risc0` framework. As explained in the preceding chapter, `ZKit` acts as the tool to define and execute these benchmark jobs in a flexible manner. In particular, we analyses two different disciplines, whereas one of them exists in two slightly derived version and thus is summarized in this thesis. Specifically, these disciplines concern preimages and IPs. Before delving into the details, we provide clarification on the selected system setting and procedure.

5.1 Setting

All benchmarks were conducted on an Ubuntu VM with a 2.3 GHz CPU and 32 Gigabytes RAM. To minimize potential side effects during these executions, only the components essential for `ZKit` were installed and no other processes ran concurrently with the benchmarks. Furthermore, each benchmark execution underwent three repetitions to alleviate unexplained fluctuations and ensure more reliable averaged results. Given the high consistency of the results, additional repetitions were considered unnecessary.

We implemented three distinct jobs covering the two previously described disciplines to prove the efficiency of creating and verifying ZKPs in `risc0`. Table 5.1 provides an overview of the available jobs and specifies the flavor and magnitude details.

The `preimage` job is centered around the name-giving discipline, utilizing various cryptographic hash functions as flavors. The magnitudes of this job indicate the number of input bytes supplied to each respective function.

Job	Purpose	Flavors	Magnitudes
`preimage`	Proving knowledge of a preimage for a given digest	Various cryptographic hash functions	Number of input or preimage bytes.
`ip`	Proving the inclusion of a single data leaf in a Merkle tree with a give root hash	Various cryptographic hash functions	Depth of the (perfect binary) Merkle tree
`ip_batched`	Proving the inclusion of multiple (batched) data leaves in a Merkle tree with a give root hash	Various batch sizes	Depth of the (perfect binary) Merkle tree

TABLE 5.1: Available Benchmark Jobs in `ZKit`
Provides an overview of the implemented benchmark jobs within `ZKit` used to analyze the performance and resource consumption of `risc0`. All jobs offer flexibility in their execution, allowing for variations through different flavors and magnitudes.

The `ip` job involves proving a single leaf using different hash functions as flavors. In this context, the magnitudes specify the tree depths of the perfect-binary Merkle trees containing the targeted leaf.

Last but not least, the `ip_batched` job focuses on the same discipline as in the `ip` job, but with a focus on proving multiple leaves at once. In other words, we generate a single STARK proof for an execution trace that involves verifying an array or *batch* of IPs. The flavoring in this job indicates the number of IPs processed within one proof. We refer to this number also as *batch size*. Similar to the single setting, the magnitudes in this context correspond to the tree depths.

Considering that the preimage job already analyses the efficiency of hash functions atomically, revisiting the same flavor classes during the more complex `ip` job is deemed redundant for the remainder of this chapter. Instead, we direct our attention to the results of the `ip_batched` job executions, employing the hash function identified as the most efficient in the `preimage` job. This approach eventually enables a comparison between the single and batched IP job variants, whereas we assess a range of batch sizes. In essence, we aim to answer the question of whether creating a single ZKP for a batch of IPs can be more efficient than proving the same number of IPs sequentially using multiple ZKPs.

The rest of this chapter structures as follows. Section 5.2 summarizes the results from the `preimage` job and discusses the selection of the superior hash function in terms of efficiency. We used the selected hash function to run the `ip_batched` job and present the results in Section 5.3. Finally, we analyze the relative performance gain of batched settings.

5.2 Preimage

As the name suggest, this job simply entails calculating a hash value for a secret preimage. The chosen cryptographic hash function, acting as flavors, are the following:

- **sha2-256:** Belongs to the `SHA-2` hash function family [44]. Widely employed in practice for data integrity verification, cryptographic applications, and blockchain applications. We selected this function based on its widespread popularity.
- **sha2-256-o:** An optimized version of `sha2-256` specifically designed for the `RV32IM` ISA used in `risc0`'s VM. We included this function as a flavor in order to validate the promised optimized performance.
- **blake3:** High-speed cryptographic hash function based on the `BLAKE2` family, offering parallelism and security [45]. We chose this function for its competitive performance compared to other functions and its potential for acceleration on `RISC-V` ISAs [46].
- **keccak-256:** Member of the `SHA-3` family [47] and acknowledged as one of the most secure hash functions. This choice is simply motivated by the fact to include an representative of the `SHA-3` family as flavor.
- **sha3-256:** Another member of the `SHA-3` family [47] and represents a derived version of `keccak-256`, optimized for performance. The inclusion of this function aligns with the rationale for `keccak-256`.

The magnitudes in this job represent the number of input bytes of the hashed message and are defined as powers of two. The reason for this pattern stems from the desire to probe a wide range of workloads in a reasonable time, considering the computational complexity of ZKPs. Ranging from the smallest (`XS`) to largest (`XL`), the magnitudes are $\{64, 256, 1'024, 4'096, 16'384\}$ input bytes. In practice, we randomly generate an array of bytes with the according length.

We organized the benchmark analysis into two aspect groups. The first group addresses resource consumption in general, while the second concentrates on the resulting performance.

Resource Consumption

Figure 5.1 provides a summary of resource costs with four aspects depicted as subfigures. The different data series represent various flavors in the preimage job, i.e., hash functions.

The first aspect shown in subfigure (a) concerns the required amount of clock cycles to execute the proof logic with a given witness (previously denoted as $L(\omega)$) insides the VM. It becomes evident that there are significant differences in efficiencies among the chosen hash functions. For instance, `sha2-256-o` requires roughly between 9 to 90 times fewer cycles across all magnitudes compared to the two most expensive ones, `keccak-256` and `sha3-256`. The middle field accommodates blake3.

This behavior is directly reflected in the proof cycles illustrated in subfigure (b). As mentioned in Section 3.2, we use `risc0` with the default segment size of 2^{20} cycles. Executions requiring nearly as many (accounting for VM initialization and finalization) or more cycles will be split into two ore more segments. Each segment is capable of accommodating $\leqslant 2^{20}$ cycles.

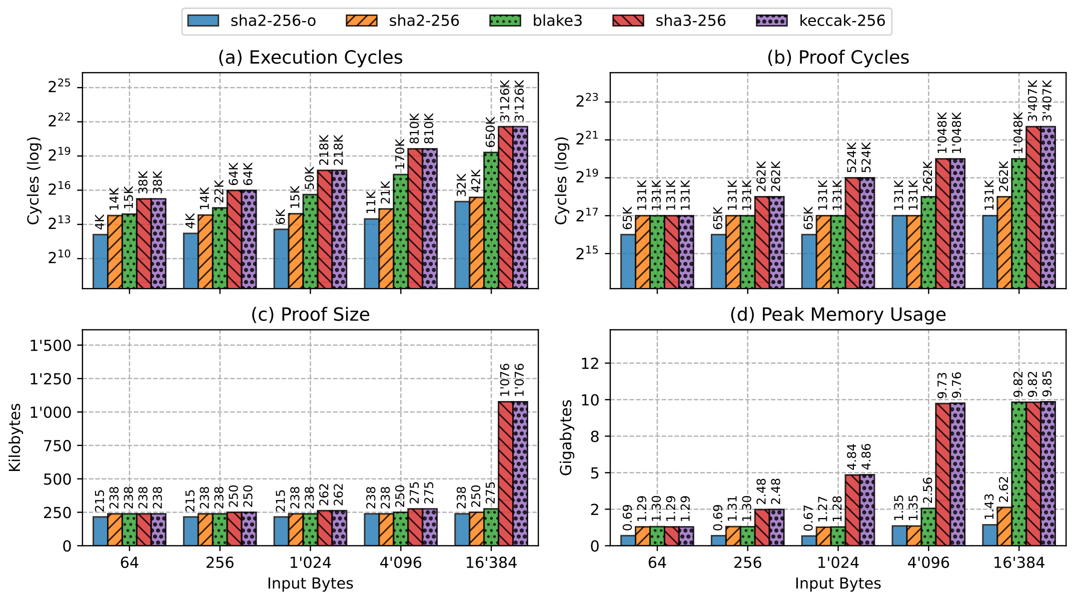


FIGURE 5.1: Resource Analysis of Preimage Benchmark Job.
Subfigures (a) and (b) illustrate execution and proof cycles using a logarithmic y-scale. Growth in proof size is depicted in (c), while (d) displays peak RAM consumption for each setting. Cycle data labels are shown in thousands with the remaining digits truncated. Labels for proof size and RAM usage are rounded to whole integers and hundredths, respectively.

An instance of such a split occurred, for example, in the observations of `keccak-256` and `sha3-256` when using 16‘384 input bytes. In both cases, the execution trace was divided into four segments which totaled $3 * 2^{20} + 2^{18} = 3'407'872$ proof cycles.

The proof size, illustrated in subfigure (c), deviates from the expected growth pattern. Despite the doubling of input bytes, it only results in a slightly larger proof file size after initially remaining constant. This observation aligns with the poly-logarithmic proof size complexity design in STARK systems, as discussed earlier (see Table 2.1).

Lastly, subfigure (d) provides insight about measured peak RAM consumption for each setting. As expected, this consumption directly correlates with the number of proof cycles, which, in turn, is derived from the number of execution cycles. However, RAM consumption seems to reach its maximum of around 10 GB for two flavors at just 4‘096 input bytes. It remains at this level despite a much higher amount of proof cycles at 16‘384 input bytes. The rationale behind this capped development lies in the segment size.

Overall, we conclude that the evolution of execution and proof cycles as well as peak memory consumption (up to certain limit) positively correlates to the exponentially growing input bytes. In contrast, the proof size deviates from this trend, remaining at a relatively low level and exhibiting poly-logarithmic growth, as expected for STARK systems.

Performance

Next, we explore how the observed resource consumption translates into runtimes. Figure 5.2 summarizes the runtime durations for all three ZKP stages.

Similar to before, each subplot categorizes the observations over different magnitudes on the x-axis. On the y-axis, the height of the bar represents the required amount of seconds.

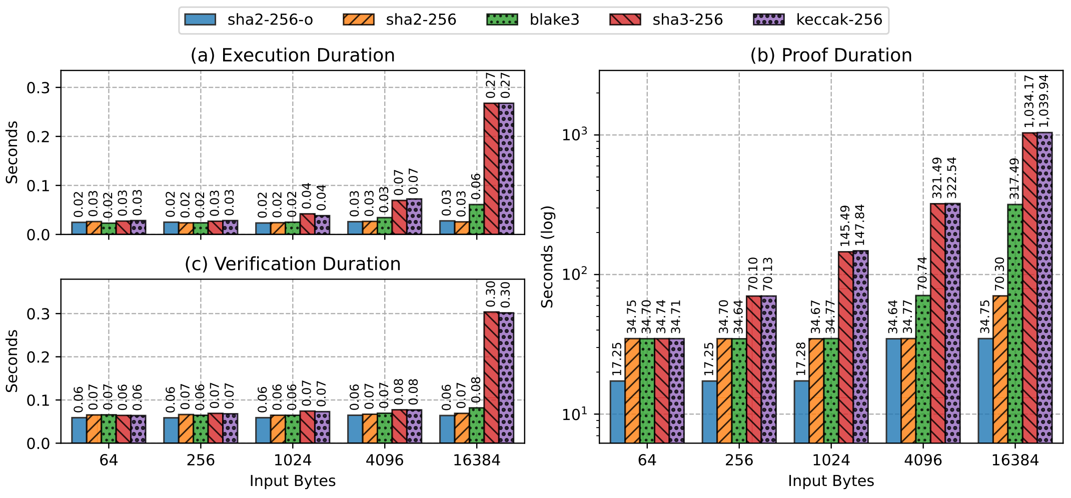


FIGURE 5.2: Performance Analysis of Preimage Benchmark Job
Subfigure (a) illustrates the execution time within the VM for generating the trace table. The creation time of the STARK proof is depicted in subfigure (b) with a logarithmic y-scale. Subfigure (c) addresses the verification durations. All data labels are rounded to hundredths.

Subfigure (a) shows the execution duration, i.e., the required time to calculate the hash value and generate the trace. As anticipated, this resides within the millisecond range and increases proportionally with the exponentially growing input size.

The time required to generate the actual STARK proof is visualized in subfigure (b). Note that the bar heights are projected on logarithmic y-scale for better readability given the wide range. The subfigure exhibits high resemblance to the proof cycle visualization in Figure 5.1, reflecting the positive correlation between cycle count and duration.

Subfigure (c) illustrates the required time to verify the generated proofs. Thanks to the poly-logarithmic verifying complexity in STARK, the required time for this task evolves nearly constant to the exponentially growing input sizes. This also mirrors the development pattern of the proof size.

Takeaways

The performance and resource analysis of the `preimage` benchmark job results distill to four takeaways.

Firstly, execution and verification times are orders of magnitude faster than proof generation time. While execution and verification typically take milliseconds, proving required up to $1'039.94 \div 60 \approx 17$ minutes. This disparity, however, is inherent to STARK systems.

Secondly, cycle count directly translates into runtime durations. This emphasizes the importance of exploring different proof logic variants as demonstrated with the selected flavors. As expected, the optimized hash function `sha2-256-o` for the `RV32IM` ISA outperformed all other tested hash functions in all aspects. Consequently, we exclusively employ `sha2-256-o` in the benchmark jobs related to IPs.

Thirdly, we experienced how segment sizing can effectively limit peak RAM utilization. As mentioned previously, we operate `risc0` with the default segment size of 2^{20} cycles. This allowed us to remain below 10 GB of RAM consumption over all magnitudes.

Lastly, our observations indicate that execution, proof, and verification times scale proportionally to the exponentially growing magnitudes. This is not the case for peak memory consumption as it does not exceed a certain maximum derived from the chosen segment size. With these insights in mind, we proceed to examine the benchmark results on the more sophisticated task of IP proving.

5.3 Inclusion Proof

This section summarizes the results from remaining two benchmark jobs, namely, `ip` and `ip_batched`. In particular, we compare the single IPs as specified in Algorithm 1 with its batched version. The latter only differentiates from the fact that it stages an array of the inputs specified in Algorithm 1. Both employ the `sha2-256-o` hash function as it qualified as the most efficient option in the previous section.

For conciseness, we consolidate the results from both jobs in the same figures. The flavors indicate the number of conducted IPs per STARK proof. Note that the flavor «1 IP» originally corresponds to the results from the `ip` job, which actually allows for different hash function acting as flavors. However, since we focus on one hash function only, this flavoring approach is deemed superfluous in this case. While it is technically feasible to introduce a batch size of 1 in the ip_batched job, it is marginally less efficient than proving a single IP due to the more complex object staging.

Similar to the `preimage` job, we perform measurements with exponentially increasing magnitudes, which, in this case, correspond to tree depths. Ranging from smallest (XS) to largest (XL), the chosen depths are 4, 8, 16, 32, 64. As mentioned previously, the IP implementation in `ZKit` is tailored for perfect binary trees. As a consequence, tree depth corresponds to the length of the authentication path.

Resource Consumption

As in the previous section, we initially address the resource consumption in Figure 5.3 using the same four aspects as before.

As anticipated based on insights from the previous section, all metrics exhibit growth corresponding to the exponentially increasing tree depth. The only deviation from this pattern is seen in peak memory usage, which once again does not surpass the 10 GB RAM mark due to the segment size.

Certain executions were again subject to segmentation as can be observed subfigure (b) at tree depth 64. For example, proving 8 IPs required three segments totaling to $2 * 2^{20} + 2^{18} =$ 2'359'296 proof cycles. Proving 16 IPs in tree with the same depth required even five segments, resulting in the total of $4 * 2^{20} + 2^{19} =$ 4'718'592 proof cycles. As mentioned

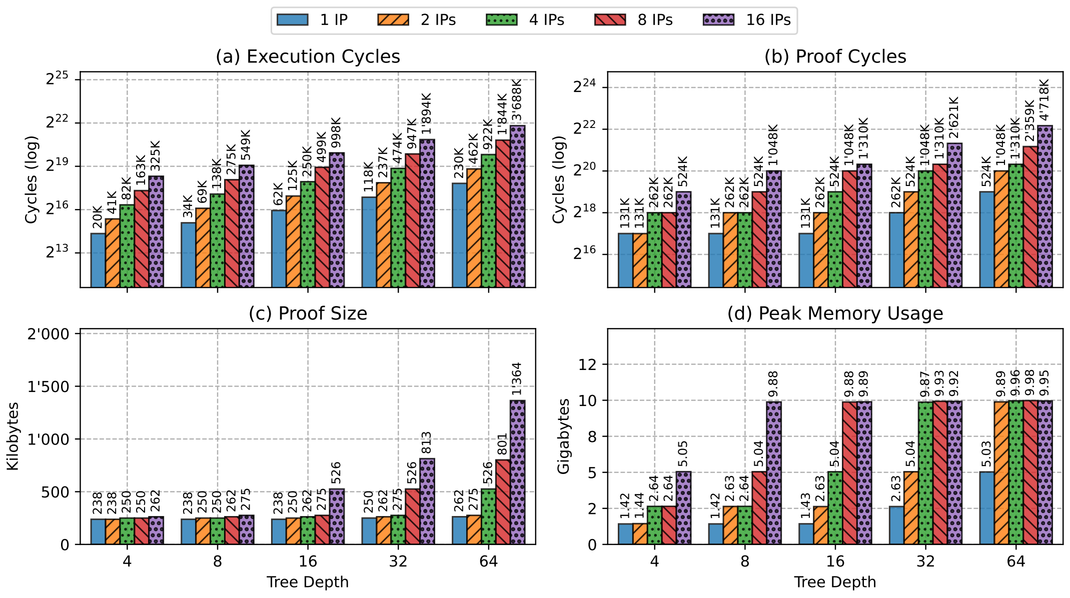


FIGURE 5.3: Resource Analysis of Single & Batched IP Benchmark Job
Subfigures (a) and (b) illustrate execution and proof cycles using a logarithmic y-scale. Growth in proof size is depicted in (c), while (d) displays peak RAM consumption for each setting. Cycle data labels are shown in thousands with the remaining digits truncated. Labels for proof size and RAM usage are rounded to whole integers and hundredths, respectively.

earlier, segmentation introduces additional cycle overhead due to paging. This overhead is not contained in the expressed execution cycles and explains why, for instance, an execution of 1'844K cycles required three segments although 1'844K $\leqslant 2 * 2^{20}$.

Examining subfigure (a), one sees that inter-flavor cycle ratio remains consistent under growing tree depths. In other words, the different flavors exhibit a consistent gradation[1] over all magnitudes. Intuitively, executing a program that performs an action x times more frequently is also x times computationally more expensive. Thus, dividing the cycle counts of each flavor > 1 IP by the cycle count of 1 IP roughly corresponds to the batch size. At tree depth 64, for instance, this evaluates to $\{462K, 922K, 1'844K, 3'688K\} \div 230K \approx \{2.01, 4.01, 8.01, 16.03\}$.

Interestingly, the same flavor gradation does not hold for proof cycles, as illustrated in subfigure (b). Repeating the calculation from before for the same tree depth confirms this as it evaluates to $\{1'048K, 1'310K, 2'359K, 4'718K\} \div 524K \approx \{2.00, 2.50, 4.50, 9.01\}$. Consequently, proving a batch of x actions necessitates $< x$ times more computational effort. Put differently, the effort to prove batched IPs grows asymmetrically with increasing batch sizes. This advantageous property can be harnessed to improve efficiency.

Performance

Continuing with performance-related aspects, we examine how the execution and proof cycles translate into runtime. Figure 5.4 visualizes the results of these aspects.

Both (a) execution and (c) verification duration reside again in the millisecond range and evolve according to the exponentially growing tree depth, shown in the x-axis. As in the `preimage` job, both durations exhibit positive correlation with observations related to resource consumption. Specifically, there exists a direct proportional relation between execution cycles and duration as well as proof size and verification time.

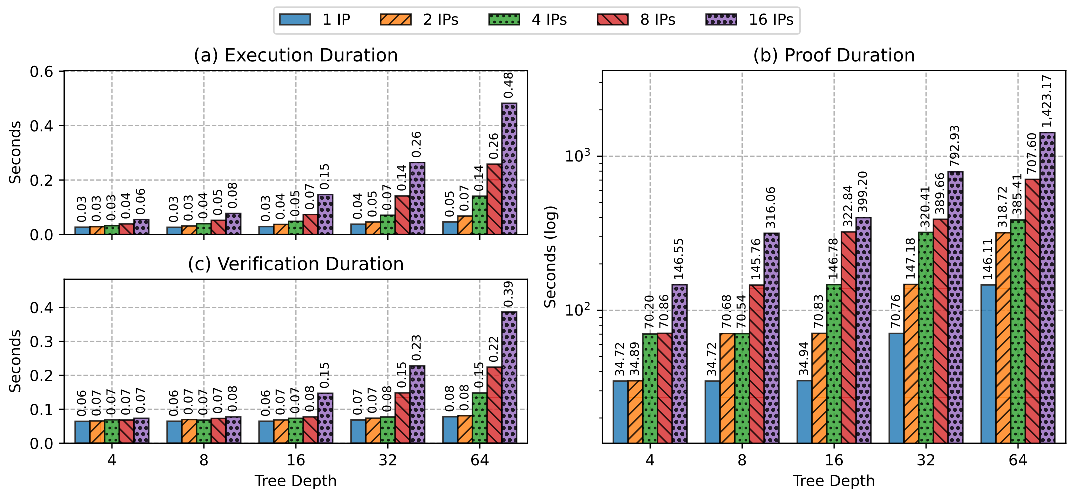


FIGURE 5.4: Performance Analysis of Single & Batched IP Benchmark Job
Subfigure (a) illustrates the execution time within the VM for generating the trace table. The creation time of the STARK proof is depicted in subfigure (b) with a logarithmic y-scale. Subfigure (c) addresses the verification durations. All data labels are rounded to hundredths.

[1] Gradation denotes a step-by-step progress characterized by a series of regular incremental advancements.

These relatively quick runtimes, however, do not extend to the proof duration, as shown in subfigure (b). It requires about $34.72 \div 60 \approx \frac{1}{2}$ minute in the simples and around $1'423.17 \div 60 \approx 24$ minutes in the most complex case to generate the STARK proof. Nevertheless, the previously stated insight regarding the asymmetric relationship between batch size and cycle complexity applies here again. At three depth 8, for instance, proving a batch of 8 IPs at once requires only $145.76 \div 34.72 \approx 4.2$x more time than proving a single IP and not 8x. This positive finding underscores the efficiency gains achievable through batching.

Takeaways

Analyzing the IP benchmark jobs using the previously declared most efficient `sha2-256-o` hash function infers the following two takeaways.

Firstly, the observations regarding the relationships between the examined aspects made in the more straightforward `preimage` job could be revisited, speaking for consistency. This includes, for example, the notion that more cycles result in longer runtime and that the segment size limits peak memory usage.

The second takeaway concern the asymmetric relationship between proving IPs in a batched setting versus sequentially in a single setting. This implies a significant impact on overall performance, considering that proving constitutes the most time-consuming stage in ZKPs. Consequently, we dedicate the remainder of this section to a more detailed exploration of this property.

Performance Gain of Batched Settings Relative to Single Settings

To analyze the discovered asymmetric relationship between batched and single IP proofs more thoroughly, we determined the multiplication factors for all flavors > 1 IP over all measured tree depths relative to 1 IP in Figure 5.5. This relative analysis is only conducted on the duration aspects, as they derive from cycle count or proof size and convey the most intuitive or tangible message to the readership. Each data point represents the multiplication

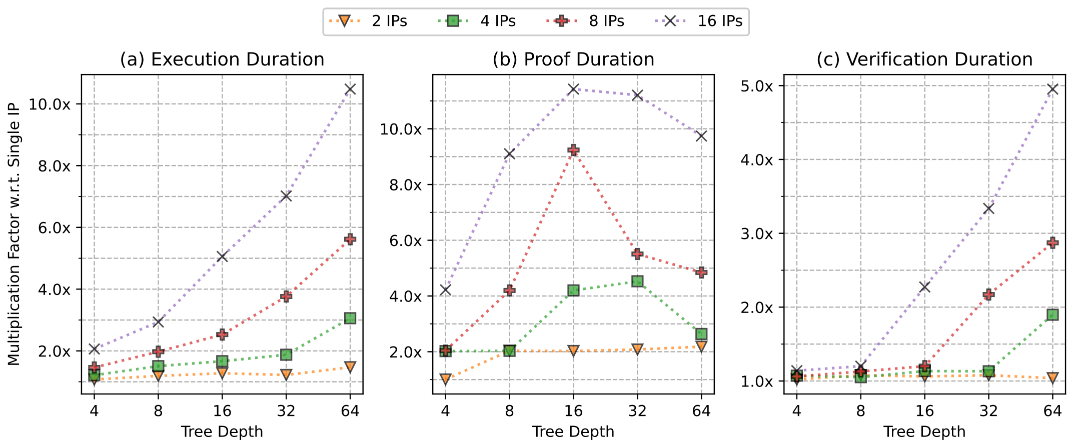


FIGURE 5.5: Comparative Duration of Batched IPs Relative to Single IP.
All subfigures visualize data derived from the duration ratio between batched and single IP setting.
This relative analysis is conducted on (a) execution, (b) proof, and (c) verification duration.

factor shown on the y-axis, resulting from dividing the measurements from the batched setting by those from the single setting.

The multiplication factors for (a) execution and (c) verification duration exhibit a similar pattern, increasing proportionally with the rising tree depth. The batched-single-ratio for verification duration, however, scales at a slower rate than for execution. For instance, up until tree depth 8, all batch sizes require more or less the same amount of time than verifying a proof containing a single IP. In contrast, the multiplication factor of the flavors in execution duration diverges from the beginning.

Even more interestingly is the proof duration in subfigure (b). For example, proving 8 IPs at tree depth 4 requires approximately 2x as much time than proving a single IP. Consequently, the batched proof is $\frac{8}{2} = 4$x faster than proving 8 single IPs sequentially. Even in the most computationally expensive setting of proving 16 IPs at tree depth 64, batching yields approximately $\frac{16}{9.8} \approx 1.6$x faster runtime than the single strategy.

This speed-up factor, however, does not scale consistently, as the hump between tree depth 16 an 32 suggests. At tree depth 8, for instance, proving 8 IPs requires more than 9x the duration of a single IP proof. Ergo, the batched setting is less performant than the single setting in this case. The reason for this hump lies in the step-wise proof cycle relationship prevalent in `risc0`.

To gain a deeper understanding of the individual performance gains or losses, we introduce the following notation. Let t_n be the time required to prove a batch of n IPs, while t_s denotes the time to prove a single IP. Accordingly, $t_s * n$ presents the projected time to prove the same number of IPs sequentially as in the batched setting that lasts for t_n. To determine the relative performance gain that arises from proving batched IPs, we calculate the ration between $t_s * n$ and t_n. Equation 5.1 states this formally.

$$\text{Percentage Gain} = G_p = \frac{t_s * n}{t_n} - 1 \tag{5.1}$$

The results are shown in Table 5.2. It indicates all G_p for all evaluated batch sizes and tree depths, structured in the three focused aspects shown in Figure 5.5.

Green cells denote positive gains, which means that proving batched IPs is faster than proving the same amount of IPs one after the other. Conversely, red cells indicate negative gain, implying that the batched setting is slower compared to the single proving strategy. Boldfaced values highlights the extrema for each individual aspect.

Across all experimental settings, batching consistently delivers faster execution and verification runtimes. Verifying a proof at batch size 16 and tree depth 4 exhibits the most remarkable performance improvement with $G_p = 1'303.64\%$. In other words, batching is approximately 14x faster in this case[2].

However, certain proof duration settings exhibit performance degradation when utilizing batching. This observation corresponds to the aforementioned hump depicted in Figure 5.5, subfigure (b). For example, proving 8 IPs at tree depth 16 incurs a 13.41% performance

[2] The factor is determined by $1 + G_p * 0.01$

Aspect	Batch Size n	G_p per Tree Depth					Average
		4	8	16	32	64	
Execution Duration	2	86.05%	68.09%	56.76%	63.77%	35.96%	62.12%
	4	229.9%	165.55%	140%	113.21%	**30.81%**	135.89%
	8	447.01%	305.13%	216.36%	112.71%	42.45%	224.73%
	16	**675.76%**	444.83%	216.36%	127.99%	52.7%	303.53%
Proof Duration	2	99.03%	-1.76%	-1.33%	-3.85%	-8.32%	16.76%
	4	97.84%	96.84%	-4.77%	-11.66%	51.64%	45.98%
	8	**292.01%**	90.53%	**-13.41%**	45.28%	65.19%	95.92%
	16	279.09%	75.74%	40.06%	42.78%	64.26%	100.39%
Verification Duration	2	95.94%	85.65%	88.35%	**85.52%**	92.59%	89.61%
	4	272.95%	280.39%	252.73%	253.45%	110.81%	234.07%
	8	653.17%	608.68%	566.09%	268.54%	178.57%	455.01%
	16	**1’303.64%**	1’232.19%	603.85%	379.53%	223.04%	748.45%

TABLE 5.2: Prove Duration Gain of Batched IPs Relative to Single IP
Cells highlighted in green indicate that a batched setting is faster to prove compared to the equivalent work in a single setting. Red cells highlight cases in which a batched setting is slower to the single setting.
Bold font emphasizes the extrema per aspect.

loss compared to processing the same workload sequentially. Nonetheless, when weighed against the overall gains, these measured losses are relatively modest in terms of percentage.

Based on the average values in the right-hand column, one can conclude two points. First, batch-processing multiple IPs can significantly enhance performance in `risc0` over all three ZKP stages. Second, the performance gain generally tends to increase with larger batch sizes. However, given the hardly pre-determinable execution cycles and the step-wise prove cycle development, detailed testing of individual settings remains essential as performance improvement is not universally guaranteed.

6 Discussion

In this chapter, we discuss important aspects of the presented content, intertwined with personal reflections and recommendations. In particular, the discussion focuses on the experiences made with Risc in the context `ZKit`. An outlook to future work concludes the chapter.

RISC Zero

Our experiences with the `risc0` framework were eminently positive. The available libraries and tools makes it accessible for new developers in this space and are intuitive to handle. Also, there exists an attentive community and insightful documentation and explanation material.

Solely during the course of this thesis, numerous `risc0` upgrades were published which speaks for the velocity this project evolves. This, however, may not be compared to other open-source projects as it is pushed primarily by the RISC Zero Incorporation[1]. A major focus in the advancement of `risc0` seems to be the interoperability with *Bonsai*[2], a blockchain-based infrastructure which is, among others, offered to outsource the computationally expensive proof creation. Although this endeavour is still in development, it seems to geared towards a future source of income.

Unfortunately, we could not see the same advancement velocity targeted to technical and mathematical specifications. The mentioned material document [29] in this regard resides since the start of this thesis in draft mode. These efforts towards comprehensibility, however, are inevitable to pave the way for eventual cryptographic audits. Both is essential for `risc0` to qualify as a candidate framework for productive systems.

ZKit

Although the development of a toolkit consumed lot of efforts, it turned out to be an rewarding automation environment for probing with ZKPs. Especially in complex systems, automation allows to effortlessly ensure correctness and allows for consistency. Our initial development effort to setup `ZKit` was later more than rewarded, as additional costs to test other settings or implementations were negligible small. Although we did not do this, but technically one could also integrate the CLI into CI pipelines for the purpose of further automation.

As addressed in Section 4.3, extending `ZKit` with further ZKP frameworks may infers re-implementing existing logic due to the various proving paradigms. While `ZKit` serves as an

[1] See company information and filings in the EDGAR database.
[2] See Bonsai overview for details.

integration environment for such heterogeneous systems, establishing and executing comparable benchmark disciplines across multiple frameworks can therefore present a delicate challenge. Developers bear the responsibility of ensuring a consistent implementation of a specific discipline under benchmarking across all frameworks. This concerns, for example, that two programs that both verify an IP minimize their differences in total operations, given the same input. In cases where substantial differences cannot be avoided, we recommend to refrain from comparing benchmark results between frameworks. Otherwise one risks conclusion making based on comparisons of apples and oranges.

At the time of writing, `ZKit` is not available open-source as it is subject to contractual agreements. We anticipate, however, to open-source `ZKit` in future after completing ongoing refinements and optimizations. This approach ensures a cleaner and more robust codebase for the sake of an enhanced user experience for both users and contributors.

Future Work

Future work bears substantial potential through various trials. First and foremost, we anticipate including further ZKP frameworks into `ZKit`. This would not only enrich the enrich the variety of different ZKP paradigms but also audit the practicability and reasonableness of the proposed `ZKit` system architecture. This also implicates completing and extending the automated plot functionality to account for inter-framework result analysis.

Next, additional benchmark jobs could analyze a greater variety of disciplines or implementation strategies. For example, IP functionality could be extend to also account for non-perfect binary trees which in turn increases the complexity of the shared IP file. Also, one could investigate performance differences among different devices or machines, including smartphones.

To delve further into benchmarking `risc0`, for instance, one could examine various segment sizes and their implications on runtimes. Another interesting avenue for exploration involves examining the potential to outsource the VM to dedicated RISC-V hardware components. Such an approach might mitigate the computational overhead needed for VM initialization and cleanup.

Futhermore, future work may concern examine the presented porting strategy to Go. Posable question involve whether the measured performances replicates in the Go wrapper library and if there exists more efficient porting strategies than using FFI. Equally important are investigations into brining ZKP functionality to other mainstream programming languages such as Python or `C#`. At the end, it is the accessibility to ZKP technologies that bears the highest potential to advance wide adoption and thus should attract high attention.

7 Conclusion

Implementing ZKPs in practical applications poses considerable challenges, demanding thorough testing and ongoing optimizations. This is especially true for STARK systems, the primary focus of this thesis as they do not require a trusted setup and solely rely on secure hash function, making them post-quantum secure. While the efficiency of verifying STARK proofs is commendable, their creation currently requires a considerable amount of time and resources, which as of today impedes practical applications. In the realm of digital cash, for example, such practical applications involve integrating ZKP in software wallets that eventually run on standard smartphones. Thus, we anticipate outsourcing to trusted custodial services or personal home severs acting as a backend could offer a viable solution to this problem in the long run.

Existing ZKP frameworks strive to enhance accessibility for developers by abstracting the complexity of these proof systems. We discussed a few promising frameworks while emphasizing the trade-offs between direct and indirect proving approaches. The indirect proving approach embraced in RISC Zero introduces a notable advantage in the simplicity of circuit definition through the use of general-purpose programming languages. Despite the accompanying increase in system complexity and overhead, this approach provides a crucial level of flexibility that allows fast adaption to new requirements.

Furthermore, our IP experiments with RISC Zero have notably demonstrated that consolidating multiple actions into a single STARK proof results in substantial efficiency gains compared to proving each action individually. This discovery underscores the practical importance of adopting a batching strategy in ZKP applications to enhance overall efficiency.

Finally, the complexity of defining ZKP circuits underscores the utility of our test and benchmark suite `ZKit`. It allowed us to implement and analyze use cases the wide range of different influence factors, such as chosen hash function, tree depth, or batch size in an automated and replicable manner. We envision `ZKit` as an useful tool for developers, empowering them to extract decision-supporting insights about ZKP frameworks and streamline their development processes.

Bibliography

[1] R. C. Merkle. "Method of providing digital signatures". U.S. pat. 4309569A. Univ Leland Stanford Junior. Jan. 5, 1982. URL: https://patents.google.com/patent/US4309569A/.

[2] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". Oct. 31, 2008.

[3] R. Bögli. "Zero-Knowledge Inclusion Proofs". Focus Project 2. Rapperswil-Jona: Eastern Switzerland University of Applied Science (OST), Aug. 21, 2023.

[4] D. Benarroch et al. "Community Proposal: A Benchmarking Framework for (Zero-Knowledge) Proof Systems". In: *QEDIT, Tel Aviv-Yafo, Israel, Tech. Rep* (Apr. 9, 2020). URL: https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-benchmarking.pdf.

[5] Y. Gong et al. "Analysis and comparison of the main zero-knowledge proof scheme". In: *2022 International Conference on Big Data, Information and Computer Network (BDICN)*. 2022 International Conference on Big Data, Information and Computer Network (BDICN). Jan. 2022, pp. 366–372. DOI: 10.1109/BDICN55575.2022.00074.

[6] Polybase Labs. *zk-bench*. Nov. 27, 2023. URL: https://zkbench.dev/ (visited on 11/27/2023).

[7] *zk-benchmarking*. Delendum, Nov. 27, 2023. URL: https://github.com/delendum-xyz/zk-benchmarking/ (visited on 11/27/2023).

[8] J. Ernstberger et al. *zk-Bench: A Toolset for Comparative Evaluation and Performance Benchmarking of SNARKs*. Cryptology ePrint Archive, Paper 2023/1503. 2023. URL: https://eprint.iacr.org/2023/1503.

[9] *zk-Harness*. zkCollective, Nov. 27, 2023. URL: https://github.com/zkCollective/zk-Harness (visited on 11/27/2023).

[10] *zkp-compiler-shootout*. Anoma, Nov. 12, 2023. URL: https://github.com/anoma/zkp-compiler-shootout (visited on 11/27/2023).

[11] Celer. *The Pantheon of Zero Knowledge Proof Development Frameworks (Updated!)* Celer Network. Aug. 5, 2023. URL: https://blog.celer.network/2023/08/04/the-pantheon-of-zero-knowledge-proof-development-frameworks/ (visited on 11/30/2023).

[12] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof-System". In: *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. STOC '85. New York, NY, USA: Association for Computing Machinery, Dec. 1, 1985, pp. 291–304. ISBN: 978-0-89791-151-1. DOI: 10.1145/22145.22178.

[13] M. Blum, P. Feldman, and S. Micali. "Non-Interactive Zero-Knowledge and Its Applications". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*.

STOC '88. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1988, pp. 103–112. ISBN: 978-0-89791-264-8. DOI: 10.1145/62212.62222.

[14] O. Goldreich, S. Micali, and A. Wigderson. "Proofs that Yield Nothing But Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems". In: *Journal of the ACM* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/116825.116852.

[15] J.-J. Quisquater et al. "How to Explain Zero-Knowledge Protocols to Your Children". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by G. Brassard. Vol. 435. New York, NY: Springer New York, 1990, pp. 628–631. ISBN: 978-0-387-97317-3. DOI: 10.1007/0-387-34805-0_60.

[16] J. Thaler. "Proofs, Arguments, and Zero-Knowledge". In: *Foundations and Trends® in Privacy and Security* 4.2–4 (2022), pp. 117–660.

[17] R. Bögli. "A Security Focused Outline on Bitcoin Wallets". Focus Project 1. Rapperswil-Jona: Eastern Switzerland University of Applied Science (OST), Mar. 7, 2023. URL: https://eprints.ost.ch/id/eprint/1103/.

[18] A. Fiat and A. Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by A. M. Odlyzko. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1987, pp. 186–194. ISBN: 978-3-540-47721-1. DOI: 10.1007/3-540-47721-7_12.

[19] E. Ben-Sasson et al. "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture". In: *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, 2014, pp. 781–796. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson.

[20] E. Ben-Sasson et al. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Paper 2018/046. 2018. URL: https://eprint.iacr.org/2018/046.

[21] E. Ben-Sasson. *A Cambrian Explosion of Crypto Proofs*. NAKAMOTO. Jan. 8, 2020. URL: https://nakamoto.com/cambrian-explosion-of-crypto-proofs/ (visited on 08/04/2023).

[22] A. M. Tran. "Theoretical and practical introduction to ZK-SNARKs and ZK-STARKs". MA thesis. Masaryk University, Faculty of Informatics, 2022. URL: https://is.muni.cz/th/ovl3c/.

[23] L. T. Thibault, T. Sarry, and A. S. Hafid. "Blockchain Scaling Using Rollups: A Comprehensive Survey". In: *IEEE Access* 10 (2022), pp. 93039–93054. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3200051.

[24] I. S. Reed and G. Solomon. "Polynomial Codes Over Certain Finite Fields". In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (June 1960), pp. 300–304. ISSN: 0368-4245, 2168-3484. DOI: 10.1137/0108018.

[25] T. Dokchitser and A. Bulkin. *Zero knowledge virtual machine step by step*. Cryptology ePrint Archive, Paper 2023/1032. 2023. URL: https://eprint.iacr.org/2023/1032.

[26] E. Ben-Sasson, L. Goldberg, and D. Levit. *STARK Friendly Hash – Survey and Recommendation*. Cryptology ePrint Archive, Paper 2020/948. 2020. URL: https://eprint.iacr.org/2020/948.

[27] BaarkingDog. *ZK-Friendly Hash Functions*. Zellic. May 30, 2023. URL: https://www.zellic.io/blog/zk-friendly-hash-functions (visited on 09/21/2023).

[28] *Miden VM*. Version v0.7.0. Polygon, Oct. 11, 2023. URL: https://github.com/0xPolygonMiden/miden-vm (visited on 12/02/2023).

[29] J. Bruestle and P. Gafni. "RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity". Draft. July 29, 2023. URL: https://www.risczero.com/proof-system-in-detail.pdf (visited on 08/05/2023).

[30] *risc0*. RISC Zero, 2023. URL: https://github.com/risc0/risc0 (visited on 08/05/2023).

[31] StarkWare. *Open-Sourcing the Battle-Tested Stone Prover*. StarkWare. Aug. 22, 2023. URL: https://medium.com/starkware/open-sourcing-the-battle-tested-stone-prover-1fe71aaab3b7 (visited on 12/04/2023).

[32] *stone-prover*. StarkWare, 2023. URL: https://github.com/starkware-libs/stone-prover (visited on 12/04/2023).

[33] J. F. Sauer. *triton-vm*. Version v0.35.0. Oct. 5, 2023. URL: https://github.com/TritonVM/triton-vm (visited on 12/04/2023).

[34] M. Gillet et al. *Valida*. 2023. URL: https://github.com/valida-xyz/valida (visited on 08/05/2023).

[35] *Winterfell*. Meta, 2023. URL: https://github.com/facebook/winterfell (visited on 08/05/2023).

[36] *Zilch*. Trustworthy Computing Group, 2023. URL: https://github.com/TrustworthyComputing/Zilch (visited on 08/05/2023).

[37] K. Asanović and D. A. Patterson. *Instruction sets should be free: The case for RISC-V*. EECS Department, University of California, Berkeley, Aug. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[38] RISC-V Foundation. "RISC-V Instruction Set Manual". URL: https://github.com/riscv/riscv-isa-manual (visited on 12/07/2023).

[39] J. Winans. *RISC-V Assembly Language Programming*. Draft v0.18.1-0-g23bd3ad. Oct. 19, 2022. URL: https://github.com/johnwinans/rvalp/.

[40] T. Zerrell. *Using Continuations to Prove Any EVM Transaction*. RiscZero. May 22, 2023. URL: https://www.risczero.com/news/continuations (visited on 12/30/2023).

[41] S. Verdi. *Why Rust is the most admired language among developers*. The GitHub Blog. Aug. 30, 2023. URL: https://github.blog/2023-08-30-why-rust-is-the-most-admired-language-among-developers/ (visited on 12/06/2023).

[42] S. Klabnik and C. Nichols. *The Rust Programming Language*. 2nd edition. San Francisco: No Starch Press, 2023. 527 pp. ISBN: 978-1-71850-311-3.

[43] *cbindgen*. Mozilla, Dec. 13, 2023. URL: https://github.com/mozilla/cbindgen (visited on 12/15/2023).

[44] National Institute of Standards and Technology (NIST). *Secure Hash Standard (SHS)*. FIPS 180-4. U.S. Department of Commerce, Aug. 4, 2015. DOI: 10.6028/NIST.FIPS.180-4.

[45] J. O'Connor et al. *BLAKE3: one function, fast everywhere*. available online, Nov. 2, 2021. URL: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf (visited on 12/28/2023).

[46] X. Zou et al. "Accelerating Blake3 in RISC-V". In: *2023 2nd International Conference on Computing, Communication, Perception and Quantum Technology (CCPQT)*. 2023 2nd International Conference on Computing, Communication, Perception and Quantum Technology (CCPQT). Xiamen, China: IEEE, Aug. 4, 2023, pp. 296–301. ISBN: 9798350342697. DOI: 10.1109/CCPQT60491.2023.00057.

[47] National Institute of Standards and Technology (NIST). *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS 202. U.S. Department of Commerce, Aug. 4, 2015. DOI: 10.6028/NIST.FIPS.202.

List of Abbreviations

AIR	Algebraic Intermediate Representation	8
API	Application Programming Interface	19
ASM	Assembly	11
CBDC	Central Bank Digital Currency	2
CI	Continuous Integration	19
CISC	Complex Instruction Set Computing	12
CLI	Command Line Interface	3
CP	Compositional Polynomial	9
CPU	Central Processing Unit	11
DLP	Discrete Logarithm Problem	6
ELF	Executable and Linkable Format	14
FFI	Foreign Function Interface	18
FRI	Fast Reed-Solomon Interactive Oracle Proofs of Proximity	9
FTS	Fabric Token-SDK	3
IP	Inclusion Proof	1
ISA	Instruction Set Architecture	1
JFS	Job-Flavor-Size	25
RAM	Random Access Memory	16
RISC	Reduced Instruction Set Computer	1
ROM	Random Oracle Model	6
SNARK	Succinct Non-Interactive Argument of Knowledge	i
SPV	Simple Payment Verification	1
STARK	Scalable Transparent Argument of Knowledge	i
VM	Virtual Machine	1
ZKP	Zero-Knowledge Proof	1
zkVM	Zero-Knowledge Virtual Machine	10

List of Figures

List of Tables

List of Listings

List of Algorithms