Roman Markus Holler, BSc

Fitting Huge Datasets into Zero-Knowledge Proof Systems

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Christian Rechberger, Univ.-Prof. Dipl.-Ing. Dr.techn.

Institute of Applied Information Processing and Communications
Inffeldgasse 16a, 8010 Graz, Austria

Graz, May 2022

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Acknowledgements

I want to thank my parents and family for supporting me throughout my studies and the present master's thesis. Furthermore, I want to thank my colleagues Gerald Birngruber and Thorsten Mattausch for our excellent collaboration as fellow students and workmates at TU Graz over the past years.

Starting with the second semester of my bachelor studies, I joined PSI Metals, and special thanks go to my boss David Binder, who always supported me in my studies. He always pointed out that my studies have the highest priority, not the company's tasks. Thanks to Hannes Stiebitzhofer from S1Seven, who came up with this thesis' topic and strongly supported the forthcomings on the practical industry aspects of this thesis.

Moreover, I want to thank Christina, who always supported me and had to spend several lonely evenings while I was working on the thesis.

Finally, I want to thank my supervisors. Christian Rechberger always believed in the meaningfulness of the thesis' topic and maintained positive vibes. Mario Barbara had a hard time making me understand the area of zero-knowledge proofs, and Roman Walch supported me with tooling, answering basic questions, and providing help for working with Griffin.

Abstract

Zero-knowledge (zk) proofs are cryptographic techniques that allow proving statements without leaking anything beyond the validity of the statements. In many use cases, e.g., in the area of supply chains, it would be beneficial to utilize zk-proofs with comparably large inputs. However, feeding massive inputs into zk-proof systems has received little attention over the past years. We base our work on an example use case provided by our industry partners, where we use a video as private input for the zk-proof system and compute statistics on the video in zero-knowledge. The statistics are public information, while the video remains private. We bind the public statistics to the private video by hashing the video inside the zk-proof system and making this hash public.

We build our work upon zk-STARKs (scalable transparent arguments of knowledge) and use the Rust library Winterfell, which implements the STARK protocol. We explore various paths for designing an efficient STARK, and our contribution is a guideline on how to achieve this goal. We elaborate on computing hashes inside a STARK and show how to solve problems arising from computing statistics inside a STARK. Furthermore, we provide performance numbers for understanding the magnitude of the complexity of our example problem.

The library Winterfell is actively maintained, new features were released in the past, and more features are expected in the future. We already know that these upcoming features will provide a significant performance boost for our implementations.

Keywords: zero-knowledge · proof system · huge input · Winterfell · statistics · hash · Rescue-Prime · Griffin · arithmetization friendly hash functions · STARK · zk-STARK · plookup · lookup table · randomized AIR with pre-processing · RAP · Fiat-Shamir

Kurzfassung

Null-Wissens-Beweise sind kryptografische Techniken, die es ermöglichen, Aussagen zu beweisen, ohne weitere Details abgesehen von der Gültigkeit der Aussagen zu verraten. In vielen Anwendungsfällen, z.B. im Bereich der Lieferketten, wäre es von Vorteil, Null-Wissens-Beweise mit vergleichsweise großen Eingabedatenmengen zu verwenden. Der Einspeisung von großen Eingabedatenmengen in Null-Wissens-Beweis-Systemen wurde jedoch in den letzten Jahren wenig Aufmerksamkeit geschenkt. Wir stützen unsere Arbeit auf ein Anwendungsbeispiel, das uns von unseren Industriepartnern zur Verfügung gestellt wurde, bei dem wir ein Video als privaten Input für das Null-Wissens-Beweis-System verwenden und Statistiken basierend auf dem Video in Null-Wissen berechnen. Die Statistiken sind öffentliche Informationen, während das Video privat ist. Wir verknüpfen die öffentlichen Statistiken mit dem privaten Video, indem wir innerhalb des Null-Wissens-Beweis-Systems den Hash des Videos berechnen und diesen Hash öffentlich machen.

Wir bauen unsere Arbeit auf zk-STARKs (Null-Wissen Skalierbare Transparente Wissensargumente) auf und verwenden die Rust-Bibliothek Winterfell, die das STARK-Protokoll implementiert. Wir untersuchen verschiedene Ansätze, um ein effizientes STARK zu entwerfen. Unser Beitrag ist ein Leitfaden, wie man dieses Ziel erreichen kann. Wir erläutern die Berechnung von Hashes innerhalb eines STARKs und zeigen, wie man die Probleme löst, die sich aus der Berechnung von Statistiken innerhalb eines STARKs ergeben. Darüber hinaus liefern wir Leistungskenngrößen, um die Raum- und Zeitkomplexität unseres Beispielproblems zu verstehen.

Die Bibliothek Winterfell wird aktiv gepflegt, neue Funktionen wurden in der Vergangenheit veröffentlicht und weitere Funktionen werden in Zukunft erwartet. Wir wissen bereits jetzt, dass diese kommenden Funktionen einen erheblichen Leistungsschub für unsere Implementierungen bringen werden.

Schlagwörter: Null-Wissen · Beweissystem · große Eingabedatenmengen · Winterfell · Statistiken · Hash · Rescue-Prime · Griffin · Arithmetisierungsfreundliche Hashfunktionen · STARK · zk-STARK · plookup · Nachschlagtabelle · Randomisierte Arithmetische Zwischenrepräsentation mit Vorbearbeitung · RAP · Fiat-Shamir

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Transparency and verifiable trust enjoy great popularity nowadays. The move toward relying on facts rather than simply trusting somebody is also perceivable in the German proverb, "*Vertrauen ist gut, Kontrolle ist besser*" (engl. "*trust is good, verification is better*"). The benefit of verification demonstrates its power for relations between multiple parties. An example is the area of supply chains, where consumers strive for complete transparency, as they want to know what they are buying. However, producers and resellers want to keep some secrets due to competitive reasons - e.g., the fluffy cake's recipe of a renowned bakery.

These requirements seem to contradict each other at first glance. A great solution to this problem are zero-knowledge (zk) proofs that increase transparency and trust while keeping the producer's confidential details secret. Consumers might be interested in particular characteristics of the fluffy cake (Is the cake made from organic food? Do the ingredients conflict with the consumer's intolerances and allergies?), but the bakery refuses to disclose the ingredients used. Zk-proofs allow consumers to retrieve their desired details while hiding the actual ingredients and recipe. Herewith we satisfy both parties' requirements: transparency and privacy.

This example demonstrates only one use case, and the same principle applies to numerous areas in B2B and B2C businesses [43, 73, 65, 67, 58, 57, 62, 77]. Examples include the pharmaceutical industry [48, 67], automotive and aerospace industry, luxury goods [33], mining of precious materials [48], food producers [48], and metal producers [76]. A closer look at metal producers reveals that vast amounts of data are recorded during production. Similar to the bakery example, the production processes are confidential, yet we want to publish selected statistics of the recorded data.

Our goal is to determine whether implementing such a system is feasible when dealing with huge datasets and state-of-the-art zk-proof systems. The research area of zk-proof systems aims to develop new, more performant solutions, especially on succinctness and compression for scalable blockchain solutions. Our question has found little attention yet besides these main research areas. As feasibility is a subjective metric dependent on each specific use case, we provide numbers and estimates for the performance of an implementation of a real-world example and leave the decision of the meaningfulness to our industry partners.

We use the Rust library Winterfell [24, 46], which implements the zk-STARK protocol [10]. As we strive for a performant solution, we propose, discuss, and implement various approaches. One main finding is that randomized AIRs with pre-processing

(RAPs) [5] would significantly boost the performance of our solutions, but native RAPs support in Winterfell is being worked on and not available yet while writing the present thesis [41].

A video of an additive manufacturing process serves as our example use case. We want to keep the video private but provide statistics of pixel values within a region of influence as our public outputs of the zk-proof system. We bind the output to the video by computing the hash value of the video within the proof system and using the hash as a public output. As hash functions, we compare Rescue-Prime [2, 75] (the current number one state-of-the-art choice for STARKs [11]) to Griffin, a hash function being designed at the institute[1] while writing this thesis. Griffin has a faster plain performance than Rescue-Prime, so it outperforms Rescue-Prime in all cases where plain hashing is the bottleneck.

Meanwhile, Griffin was released to the public [39]. However, during working on the thesis, Griffin was not public yet, and we used an older and slightly different version than the official one.

The example use case confronts us with computing arithmetization-unfriendly operations, such as the minimum and maximum of a set of values. We solve this issue by implementing the plookup [31] table lookup approach within zk-STARKs. The lookup approach also boosts the performance of computing the hash but requires native RAPs support. As native RAPs are not available yet within Winterfell, we use the Fiat-Shamir [26] heuristic to derive the randomness required for the table lookups.

Outline. Chapter 2 highlights the necessary background for the topics presented in this thesis, while Chapter 3 introduces the example use case, the problem statement, and the motivation that raises this problem statement. Chapter 4 elaborates on the technology in use and the solutions and problems within our setting. Chapter 5 provides a detailed explanation of the implementation of our solutions, and Chapter 6 presents the performance results. Chapter 7 discusses the meaning of the results and provides a guideline on how to design a proper STARK when dealing with massive datasets. Finally, Chapter 8 concludes this work by revisiting all essential aspects and results.

[1] Institute of Applied Information Processing and Communications: `https://www.iaik.tugraz.at/`

Chapter 2

Background

The technologies used in this thesis get briefly outlined in this Chapter, but only to the extent required to understand the solution. Literature pointers provide hints for the reader when certain aspects get omitted but might still be interesting to learn.

Section 2.1 describes modular arithmetic and finite fields, which are used in many zero-knowledge proof systems. Our solution uses STARKs, and one central building block thereof is polynomials, where Section 2.2 expands on them. Section 2.3 elaborates on cryptographic components used in the solution or which play a vital role inside the context of the thesis. Finally, Section 2.4 introduces zero-knowledge proof systems, and Section 2.5 summarizes related work targeting to improve the performance of "zk-unfriendly" operations.

2.1 Modular Arithmetic

The human understanding of numbers does not provide the idea of the existence of the smallest or largest number. There is always a larger/smaller number than the present number we inspect. This is the concept of infinite fields. Examples thereof are the natural numbers, the real numbers, and the complex numbers. However, there exist other types of fields, being finite fields [68].

A finite field is built upon elements such as whole numbers $(0, 1, 2, ..)$ and two operations, e.g., addition and multiplication. A so-called modulus restricts the set of possible numbers. All computations are performed modulo the modulus, and Equations 2.1 and 2.2 show the computational rules for addition and multiplication for the field elements x, y, and

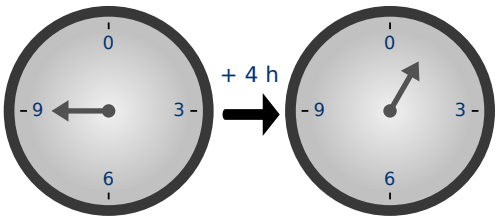

Figure 2.1: Modular math example: The analog clock [87].

z using modulus a. Figure 2.1 shows a real-world example of a finite field - the analog clock [68].

$$z \equiv x + y \pmod{a} \tag{2.1}$$

$$z \equiv x \cdot y \pmod{a} \tag{2.2}$$

The underlying finite field for the analog clock example uses whole numbers, and the modulus is 12. If we add one hour to our current time of 12 o'clock, we end up with 13 o'clock, which is 1 o'clock ($12 + 1 \equiv 1 \pmod{12}$). Note that the number 12 would not exist in theory, as 12 equals 0 in the finite field with modulus 12. In modular math theory, we would have 0 o'clock instead of 12 o'clock [68].

$\mathbb{F}_p$ denotes finite fields built upon whole numbers with modulus p, where $p \in \mathbb{P}$ — so-called prime fields. Elements inside this field can take any value $\in \{0, 1, 2, .., p-2, p-1\}$. Prime fields have unique properties we need for our protocols. No deep understanding is required thereof as we design our solution in the "user space" of the protocol. We do not modify the protocol itself [68].

Computations within a finite field behave as we know them from infinite fields, like natural numbers, except for wrapping around at the modulus. Subtraction is the inverse operation of addition, and division is emulated by multiplication with the multiplicative inverse element [68].

Observation. For a, b, and $c \in \mathbb{N}$, where all three numbers are smaller than $p \in \mathbb{P}$, and b divides a yielding c, the statement given in Equation 2.3 holds. We can use the same numbers (a, b, and c) and interpret them as elements of $\mathbb{F}_p$, where the statement in Equation 2.4 holds.

$$c = \frac{a}{b} \tag{2.3}$$

$$c \equiv \frac{a}{b} \pmod{p} \tag{2.4}$$

We omit the details of the properties of finite fields and ask the reader to consult the literature for an extensive definition of all rules and properties [68, 50].

2.2 Polynomials

A univariate polynomial is a function of the form $P(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + ... + a_{d-1} \cdot x^{d-1} + a_d \cdot x^d$. We will only focus on univariate polynomials where x is the variable, the $\{a_i\}_{[d]}$ are the coefficients, and d is the degree of the polynomial. Figure 2.2 shows a degree 5 polynomial in $\mathbb{R}$, where the horizontal axis is x, and the vertical axis is $P(x)$ [51].

Polynomials may be defined over any field, including prime fields. A polynomial of degree d has at most d roots. The roots of a polynomial are assignments of x that let $P(x) = 0$ [49, 51].

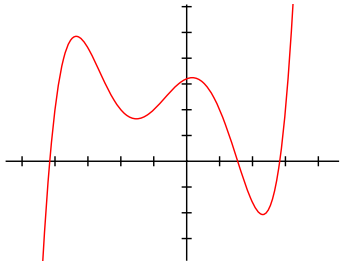

Figure 2.2: Example univariate polynomial of degree 5 [89].

Schwartz-Zippel Lemma. Evaluating a non-zero polynomial $P(x)$ defined over $\mathbb{F}_p$ at a randomly chosen position $x = \phi \in \mathbb{F}_p$ yields $P(\phi) = 0$ with probability $\leqslant \frac{d}{|\mathbb{F}_p|} = \frac{d}{p}$ [44, 63, 18].

We can utilize the Schwartz-Zippel lemma to check if two polynomials are equal probabilistically. The polynomials $P(x)$ and $Q(x)$ (both having degree d) are equal w.h.p. if the polynomial $R(x) = P(x) - Q(x)$ equals 0 for a randomly chosen $x = \phi$. The degree d of the polynomials and the domain size play an important role, as a higher degree and smaller domain size reduce the probability that the polynomials are equal [44, 63, 18].

2.3 Cryptographic Building Blocks

We now introduce some cryptographic building blocks regarding their properties and purpose. We describe the structure and what these components achieve, but not how they work. Nigel P. Smart uses color coding in his book [68] to distinguish private values from public values. Blue values are public, and everybody may know them. On the other hand, red values are private, and we want to keep these values secret at all costs. We also adopt this color coding scheme in the present thesis [68].

2.3.1 Hash Functions

Hash functions transform an arbitrary length input to a fixed-sized or a variable-sized output. Without going into the details, we will focus on hash functions producing fixed-size outputs [68].

Cryptographic hash functions are a special class of hash functions that aim at making the output appear random. Hash functions are deterministic, and therefore, the output is a pseudorandom value dependent on the input. Furthermore, a minimal change in the input (e.g., flipping one bit) results in an entirely different hash. The output shall be unpredictable based on knowing the input. Another property we require from hash functions is preimage resistance, meaning that given a hash value (output), it is computationally infeasible to find an input that hashes to the given output. There exist more properties that ensure our assumptions hold, and we ask the reader to look up the details in the literature [68].

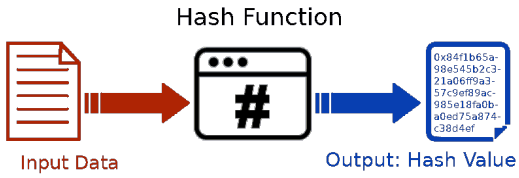

Figure 2.3: Interfaces and core principles of a hash function [15].

The properties of a cryptographic hash function can be compared to the real world regarding humans and their fingerprints. Every human has a unique skin pattern on each finger, allowing one to uniquely identify them among their peers. However, the fingerprint does not reveal any information about its human (gender, skin color, illnesses, age) up to a certain degree. Knowing the fingerprint makes it difficult to find its owner, the human being. The input to a hash function is the human, and the fingerprint is the hash function's output.

We often refer to *data* and its *hash* (or *hash value*), where the *data* is the input to some hash function, and the hash function's output is the *hash* of the data [68].

We utilize hash functions for the following tasks:

- Computing the "digital fingerprint" of data for committing to that data (see Section 2.3.3).
- Deriving pseudo-randomness in the Fiat-Shamir approach (see Section 2.3.4).

As we can commit to data using hash functions, we could argue that the data (input) is private. These thoughts led to describing the input to be private (red) and the output to be public (blue) in Figure 2.3, which shows the concept of a hash function. The # symbol is often used to denote hash functions [68].

Arithmetization-Friendly Hash Functions. Classical cryptographic hash functions (such as SHA2/3 [71]) make heavy use of bitwise operations. These operations are costly to express in many constructions of zk-proof systems, and the community tries to find ways to reduce the complexity of implementing such operations (see Section 2.5). However, these optimization approaches cannot solve the problem of computing hashes within zk-proof systems. This led to a new research area focussing on designing arithmetization-friendly hash functions (Arithmetization is one component when developing a zk-proof; see Section 2.4) [10, 2].

Many attempts have been made to develop a proper hash function, but today's solutions suffer from bad plain hashing performance. Examples of such designs are the Rescue and Vision family [2, 75], Poseidon [40], and Reinforced-Concrete [7]. While working on this thesis, a team at our institute[1] is designing a new hash function called Griffin. Rescue-Prime is the best choice for zk-STARK friendly hash functions, according to a

[1]Institute of Applied Information Processing and Communications: `https://www.iaik.tugraz.at/`

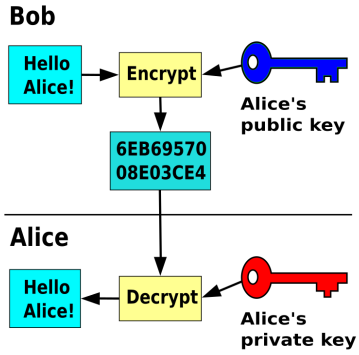

Figure 2.4: Encryption using asymmetric cryptography [38].

survey conducted by StarkWare Industries [11]. We will use Rescue-Prime in our solution and compare Rescue-Prime to Griffin.

Meanwhile, Griffin was released to the public [39]. However, during working on the thesis, Griffin was not public yet, and we used an older and slightly different version than the official one.

Arithmetization friendly hash functions make use of operations that are natively supported by the finite field in use. Furthermore, zk-proof systems like STARKs allow multiplication, but exponentiating with large numbers becomes infeasible. Therefore, arithmetization friendly hash functions also focus on operating with small exponents (this is only partially true - Section 4.3.1 introduces Rescue-Prime and how to perform arithmetization) [10, 2].

2.3.2 Asymmetric Encryption and Signatures

Symmetric encryption uses an algorithm that turns plaintext into ciphertext and vice-versa using one key. This scheme is very efficient compared to asymmetric encryption in terms of throughput performance. Asymmetric encryption uses two keys: the private key and the public key. Having access to one key is not enough to perform encryption and decryption. The public key is known to everybody, and the private key is kept secret [68].

The asymmetric scheme can be used in two ways:

- **Encryption.** Anyone knowing Alice's public key can encrypt a message for Alice. Alice is the only one who can decrypt and read the message, as she owns the private key (see Figure 2.4) [68].
- **Signatures.** Alice can use her private key to encrypt a message. Anyone knowing the public key can decrypt the message and read the content. Since Alice's public key decrypts the message, the public is convinced that Alice wrote the message, as she is the only one possessing the private key. This procedure signs the message (see Figure 2.5) [68].

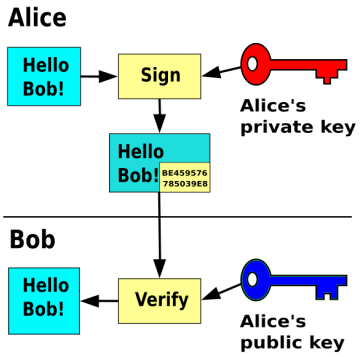

Figure 2.5: Signing messages using asymmetric cryptography [29].

Other signature schemes exist besides asymmetric encryption, and we omit further details [68].

We know that asymmetric encryption is inefficient for processing large amounts of data. However, we still want to sign messages efficiently. Suppose Bob wants to sign a vast amount of data. Instead of encrypting the data, he can compute the hash of the data and sign the hash. As the hash value is a fingerprint of the data, we know that Bob wants to sign the data behind the hash value [68].

2.3.3 Notarization

A notary is a trusted third party acting as a person who establishes trust between two or more parties having trust issues. The word's origin relates to a person who writes down (notes) facts that hold in a legal setting. For example, a notary can accredit that two parties have signed a contract [88].

We borrow this terminology and define notarization as signing a hash value and publishing this hash value to a trusted third party (e.g., a public ledger (blockchain) or an authority). Notarization is a cryptographic signature extended by a commitment of time. We can prove that we have signed a message at a certain point in time, and we cannot deny the signature, as public ledgers are (usually) immutable. This principle is similar to the classical notary, who can prove that we have signed a message.

Parties interact via signing transactions with their private key in the blockchain setting, and the public key serves as their public identity. Transactions get written to the public ledger, the blockchain. So notarization can be simplified to writing the hash value to the blockchain using the private key for signing the transaction [56].

2.3.4 Fiat-Shamir Heuristic

Interactive protocols rely on the principle that the prover claims a statement, and the verifier interactively provides a challenge based on this claim. In a naive non-interactive setting, the prover would select the challenge himself and can therefore prepare a statement

that fits his challenge. As he chooses the challenge himself, the challenge may be chosen before claiming his statement, allowing the prover to cheat arbitrarily. Interactive protocols rely on the property that the verifier observes all interactions with the prover (the verifier would notice brute-forcing) and that the challenge is unpredictable for the prover [26].

The Fiat-Shamir heuristic allows transforming an interactive protocol into a non-interactive variant by utilizing cryptographic hash functions as a secure source of pseudo-randomness. The prover needs to define the statement first and commit to it by hashing the statement. The resulting hash defines the challenge. As the output of a cryptographic hash function is unpredictable, the challenge is unpredictable, and we have similar assumptions as in the interactive setting [26].

2.4 Zero-Knowledge Proofs

In 1985, Goldwasser, Micali, and Rackoff [36, 35] presented the idea of zero-knowledge (zk) proofs. This technology opened up new areas of applications. However, zk-proof systems are very inefficient regarding space and time complexity. The research area around zk-proof systems is highly active, and many advancements have been accomplished in the past ten years. In 2022, the problem of being very inefficient in space and time complexity remains, even though the latest proof systems significantly improve upon the previous approaches.

In 2011, the term SNARK (Succinct non-interactive argument of knowledge) and its technology were characterized by Bitansky et.al [12].. First widespread realizations of SNARKs include the Pinocchio Protocol [61] (2013), the work by Jens Groth, referred to as Groth16 [42] (2016), and Bulletproofs (which is not a pure SNARK) [14, 59] (2018). One of the latest SNARK realizations is PLONK [32] (2019).

Another technology called STARKs (Scalable transparent arguments of knowledge) was introduced in 2018 by Ben-Sasson et.al [10].. SNARKs and STARKs aim to solve the same problem (to be a zk-proof system), but the underlying principles are fundamentally different.

Before comparing the different technologies, we first define what a zk-proof is and what cryptographic properties are required.

A zero-knowledge proof aims to convince the verifier that a statement is true without the prover revealing anything else besides the statement being true. One fictional example within the ongoing COVID-19 pandemic is the proof of being vaccinated, tested, or recovered from an infection when visiting certain places, like restaurants, in the European Union [16]. We call this scenario fictional because the actual implementation does not comprise zk-proofs.

Example. Assume we want to visit a restaurant, where we need to convince the security guard that we pose a low epidemiological threat to other people. We can show the security guard a certificate issued by governmental authorities that this is the case. The certificate hides whether we are vaccinated, tested, or recovered. The security guard learned that we are allowed to enter the restaurant but nothing else.

The above example describes a potential use case for zk-proofs but does not help understand how zero-knowledge is created. There are many ways to develop a zero-knowledge protocol, and we will now provide another real-world example of a simple protocol.

Example. [60] Alice is color blind, and Bob is not. Alice and Bob are interested in two balls with identical shapes, textures, and materials, but one is green and the other is red. Due to Alice's color blindness, she cannot tell the balls apart. Bob can perceive colors and can distinguish them.

Bob claims he is not color blind, and we will now prove this statement without revealing anything else. We play a game consisting of multiple rounds, and each round follows the same pattern: Alice shows both balls to Bob. Then she hides the balls behind her back, shuffles them, and shows the shuffled balls to Bob. Bob must now state whether the balls are in the same hands as before the shuffling procedure or not.

If Bob can perceive colors, he will always be able to provide the correct answer. Alice knows if the balls changed hands or not because she performs the shuffling operation. If Bob cannot perceive colors, he has a 50 percent chance of making the correct guess. After one round, the chances are 50:50 that Bob can see colors or that he was just lucky with guessing correctly. Note that a wrong guess immediately reveals that Bob is trying to fool Alice.

We now play this game over multiple rounds. Assuming that Bob always answers correctly, the probability that he is lucky halves with every round played. After having played ten rounds, the probability that Bob was just lucky is already less than 0.1 percent. This "lucky percentage" describes the error probability of any protocol, where the prover always has a slight chance of being able to create a fake proof. However, these percentages can be pushed down arbitrarily by increasing the number of rounds in the case of the presented protocol.

This example protocol proves that Bob is not color blind without revealing which ball has which color. Alice knows the balls must have a different color, as Bob can distinguish them, but Alice did not learn which ball is green and which one is red.

Core Properties of Zero-Knowledge Proofs. As we now understand this examplatory real-world protocol, we can define the three core properties of zero-knowledge proofs: [42]

- **Completeness.** The honest prover can always convince the verifier about the correctness of the statement.
- **Soundness.** The malicious prover cannot convince the verifier about a false statement (up to a negligible probability).
- **Zero-Knowledge.** The verifier does not learn anything else except the statement's correctness.

In the color-blind example, completeness is satisfied because Bob can always provide the correct answer if he is not color-blind. If Bob is color blind, it is doubtful that he always guesses correctly. However, there is a tiny chance that he will succeed, therefore

satisfying the soundness property. The protocol also fulfills the zero-knowledge property, as Alice does not learn anything about the balls, except that one of them has green color and the other one red color. She does not know which ball has which color [60].

Arguments and Proofs of Knowledge. A proof of knowledge has statistical soundness, meaning that a computationally unbound prover cannot create fake proofs according to the soundness property. In the real world, there are no computationally unbound parties. An argument of knowledge has computational soundness, meaning that the soundness property holds for computationally bound provers but not for computationally unbound provers [13, 45].

Succinct Non-interactive ARguments of Knowledge (SNARK). Succinctness refers to the size of the proof and the time it takes to verify that proof. Succinct protocols focus on small proofs and a fast verification time. Non-interactivity refers to the prover creating the proof independently without interacting with the verifier (the color-blind example is an interactive protocol) [80, 79, 84].

Traditional SNARKs are created by taking the computation we want to perform in zero-knowledge and transforming it into an algebraic circuit. This algebraic circuit operates on finite field elements and allows addition and multiplication. The circuit gets defined over variables and constraints connected via different gates (a gate performs multiplication or addition). The prover needs to provide valid variable assignments that satisfy this circuit [80, 79, 84].

A trusted setup phase is required for every instance of a SNARK. This setup phase generates the randomness required to achieve all cryptographic properties. As we have seen for the Fiat-Shamir heuristic (see Section 2.3.4), the prover must not generate the randomness independently, as nobody can check if this was done correctly. This trusted setup phase is one major drawback of SNARKs. Furthermore, SNARKs rely on strong cryptographic assumptions and the SNARKs protocol is not post-quantum secure [80, 79, 84].

Scalable Transparent ARguments of Knowledge (STARK). Scalability means providing better performance for more extensive computations than other state-of-the-art solutions. Furthermore, scalability is an essential topic in the blockchain space, where the goal is to increase the overall number of transactions on the blockchain in an easy-to-verify manner. STARKs provide compression, meaning that the verification time is exponentially faster than prover time and plain computational time - very appealing in the blockchain world. Transparency refers to being independent of a trusted third party - no trusted randomness needs to be generated compared to SNARKs. Moreover, STARKs rely on a handful of cryptographic assumptions and are post-quantum secure [10, 82, 83, 81].

STARKs are constructed using polynomials over finite fields. Computations must be transformed into a polynomial-like structure where constraints on that polynomial encode the computation. This construction allows skipping some computational steps compared to SNARKs, where every addition and multiplication requires its gate. The security assumptions only rely on hash functions and the math behind polynomials [10, 82, 83, 81].

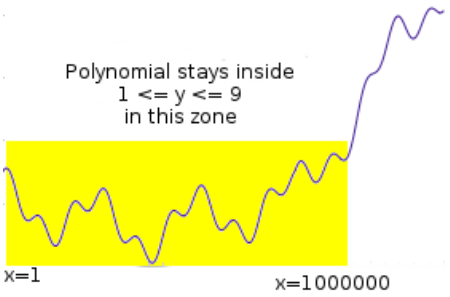

Figure 2.6: Sketch of $P(x)$ [82].

2.4.1 Scalable Transparent Arguments of Knowledge (STARK)

We now explain the inner workings and general idea of the STARK protocol. There are many variations on implementing such a protocol, and the "true and only" solution does not exist. The following explanation is based on the blog posts from Vitalik Buterin [82, 83, 81], and the missing details in the following description are to be looked up in these blog posts. First, we explain the inner workings of the protocol, and then we describe how the user defines a STARK with the tools provided by the protocol.

Inner Workings of the Protocol

We now explain the inner workings of the protocol by using an example polynomial $P(x)$ defined over a finite field, and we want to show that the evaluations of $P(x)$ are within the range 1 to 9 for a given interval of x values, e.g., from zero to one million (see Figure 2.6). The straightforward way to show this is to provide the polynomial definition and to evaluate the polynomial on each position (in total, one million evaluations) to check if each evaluation is inside the range 1 to 9. This simple approach is expensive, and we now show how the STARK protocol makes the verification of this statement more efficient [82].

Transformation into a Polynomial Equality Check. Alongside the polynomial $P(x)$, we introduce an additional polynomial $C(x)$, the constraint checking polynomial. $C(x)$ is defined as $C(x) = (x-1)(x-2)...(x-9)$ and evaluates to zero if x is in the range 1 to 9 and is non-zero otherwise (see Figure 2.7). We can now transform our verification step to check if $C(P(x))$ evaluates to zero for x in one to one million [82].

Furthermore, we introduce the polynomial $Z(x)$, which is constructed in a similar way to $C(x)$, and $Z(x)$ evaluates to zero when x is in the interval of one to one million. We now use the fact that when two polynomials share the same roots (values of x that let the polynomial evaluate to zero), one of those polynomials is a multiple of the other one. We can now reformulate the problem as $C(P(x)) = Z(x)D(x)$, where $D(x)$ is a helper polynomial that can be computed by dividing $C(P(x))$ by $Z(x)$ [82].

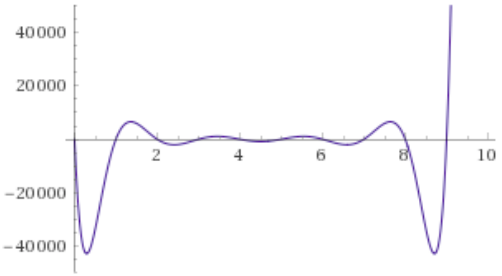

Figure 2.7: Sketch of the constraint checking polynomial $C(x)$ [82].

If $P(x)$ is inbetween 1 to 9 for x values in one to one million, then we can come up with a polynomial $D(x)$ such that $C(P(x)) = Z(x)D(x)$. Note how we transformed the initial problem into checking that two polynomials are equal. This equality check gets performed utilizing the Schwartz-Zippel lemma introduced in Section 2.2. The prover commits to the polynomials by computing one Merkle Tree (see Figure 2.8) containing all polynomials. A Merkle Tree is a hash tree that behaves similar to a hash function but provides the possibility to perform membership checks of leaf elements (see Figure 2.9). The verifier randomly chooses a selected number of Merkle Branches, and the prover needs to provide a valid opening of these branches. Each leaf contains the evaluations of all polynomials at a particular position. The verifier needs to check that these evaluations satisfy the condition $C(P(x)) = Z(x)D(x)$ [82].

Appropriate steps need to be taken to transform the protocol into a non-interactive one according to the Fiat-Shamir transformation introduced in Section 2.3.4. We omit further details on this transformation [82].

The Merkle Tree gets constructed on a larger domain than the domain used for our checks. Let us choose the domain of one to one billion, and the prover computes the evaluation of all polynomials on every position within one to one billion. The prover then constructs the Merkle Tree on all evaluations. The challenge provided by the verifier allows then to perform the Schwartz-Zippel check on the polynomials. Figure 2.10

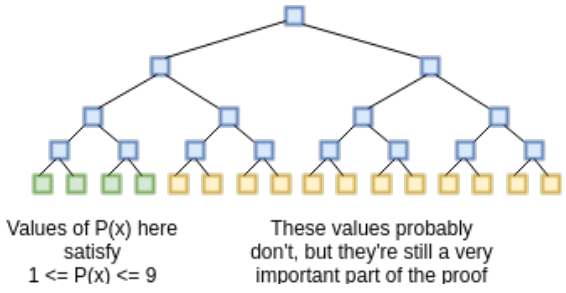

Figure 2.8: Merkle Tree for our $P(x)$ example [82].

Figure 2.9: Membership proof (Merkle branch) [82].

visualizes this approach. The number of checks is a parameter of the protocol and influences the protocol's security [82].

Checking that the Evaluations belong to low-degree Polynomials. So far, we have performed a Schwartz-Zippel check to prove that two polynomials are equal. However, we did not check whether the evaluations belong to a low-degree polynomial. This step is crucial as this heavily impacts the security of the Schwartz-Zippel check. We require the polynomials to be close to a specific degree we know ahead of time. The degree is essential, as a vast degree reduces the security of the Schwartz-Zippel check [82, 83].

We use the FRI protocol to prove that our evaluations belong to a low-degree polynomial. This protocol provides a proof that a given set of evaluations is close to a polynomial of a specific target degree. This protocol utilizes various properties of prime fields, and the choice of the prime field heavily impacts the protocol's performance. We omit the details of the protocol, as the vital aspect to understand is that we are proving that our polynomials are close to the degree we want them to be. Checking that the evaluations are polynomials makes the Schwartz-Zippel check meaningful [83, 10].

Assembling the Protocol. We have now briefly outlined how we transform a problem into a polynomial equality check problem. The provided example is straightforward, and we want to note that the protocol consists of transition constraint checks and boundary constraint checks. Transition constraints get evaluated between two adjacent states, and the idea is to define a multivariate constraint checking polynomial $K(x_{current}, x_{next})$, where $x_{current}$ describes the x value for the current state, and x_{next} describes the value

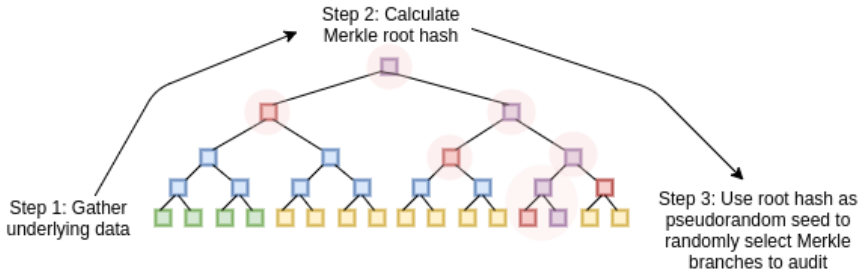

Figure 2.10: Schwartz-Zippel check utilizing randomness from the Merkle Tree [82].

of x for the next state. We can construct the univariate constraint checking polynomial $C(x) = K(P(x), P(x+1))$. Furthermore, boundary constraints ensure that the starting, ending, and intermediate positions are correct [82, 83, 81].

An example would be a STARK that proves that $P(x)$ forms the Fibonacci sequence. We could let $C(x) = P(x+2) - P(x+1) - P(x)$. Note that $C(x)$ evaluates to zero for all x in a given range when $P(x)$ forms the Fibonacci sequence. Additionally, we need to ensure that the beginning of the sequence is correct. We need to reveal the value of $P(1)$ [82].

To summarize, we construct polynomials and commit to them via a Merkle Tree. The commitment serves as a pseudorandom value to perform a Schwartz-Zippel check on these polynomials. Additionally, we need to check that our evaluations belong to a polynomial, and we do this by using the FRI protocol, which produces a proof. We also need to bind some polynomial evaluations to specific values, which form the boundary constraints (we omit the details) [82, 83, 81].

The verifier needs to check the openings of the Merkle Tree (that the opening belongs to the Merkle Tree root and that the content of the opening satisfies the Schwartz-Zippel check). Additionally, the FRI proof needs to be verified by the prover as well as the bindings of the boundary constraints [81].

Arithmetizing Computations in Practice

We have introduced the inner workings of the STARK protocol. However, we did not explain every aspect in detail. With the general understanding provided in the previous section, we move on to a more general view of STARKs that allows engineers to use the protocol.

The Algebraic Execution Trace (AET) is a two-dimensional array comprising variables (columns) and steps (rows). Each cell in this two-dimensional array (also called table) is a finite field element. A step describes the assignment of variables at a given point in time. The AET plays the role of the polynomial $P(x)$ from the previous section, where we defined $P(x)$ as a univariate polynomial. However, the AET can be considered a multivariate polynomial (one column is one parameter). We omit further details on transforming the univariate protocol to a multivariate version. The goal of the previous section was to provide a brief overview of the protocol structure, and we do not need to understand the multivariate protocol in detail [10].

Transition constraints ensure that two adjacent steps fulfill specific properties. The transition constraints are defined globally and are applied between every pair of steps in the AET. The equivalent of transition constraints would be the constraint checking polynomial $C(x) = K(P(x), P(x+1))$ from the previous section. The number of transition constraints may be chosen arbitrarily, and transition constraints must be equal to zero on one side of the equation [10].

Table 2.1: Example AET for computing the Fibonacci sequence.

step i	a	b
0	1	1
1	2	3
2	5	8
3	13	21
..	..	..

$$0 = a'' - b' - a' \tag{2.5}$$

$$0 = b'' - a'' - b' \tag{2.6}$$

The notation for defining transition constraints uses the variable names modified by a single quote (denoting the current state) and a double quote (denoting the next state). For example, the current state of the variable a is denoted by a' and the next state by a''.

Table 2.1 shows the AET for computing the Fibonacci sequence, and Equations 2.5 and 2.6 show the transition constraints. It would be handy to have three states available within the transition constraints to define the Fibonacci sequence in a clean and readable way. As this is not the case in the definition of the STARK protocol [10] (we only have a current and a next state, but not a third state), we need to spread our sequence across two columns so that we have access to all values we need. The Fibonacci sequence is defined in the AET as $\{a_0, b_0, a_1, b_1, ..\}$.

$$1 = a_0 \tag{2.7}$$

$$1 = b_0 \tag{2.8}$$

Boundary constraints are defined as assigning a specific value to a variable at a specific step. The value of a variable a at step i gets denoted by a_i. Equations 2.7 and 2.8 define boundary constraints for our Fibonacci example [10].

The union of the AET and all the constraints is referred to as the Algebraic Intermediate Representation (AIR) [10].

2.5 plookup

In 2020, Ariel Gabizon and Zachary J. Williamson (from the Aztec team [6], creators of PLONK [32]) presented a solution for lookup tables [31] in the SNARK research area.

Table 2.2: XOR truth table as an example for a lookup table.

a	b	$c = a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

The motivation is to make "SNARK unfriendly" operations more efficient. An example is binary operations between two n-bit values.

In the present thesis, we use the idea of plookup to arithmetize such problems inside a STARK. We do not use the plookup protocol. We use our own construction to show the equality of polynomials $F(\beta, \gamma)$ and $G(\beta, \gamma)$ using the Schwartz-Zippel lemma introduced in Section 2.2 and the Fiat-Shamir Transform described in Sections 2.3.4 and 4.3.8.

The idea is to pre-compute a lookup table of valid values. We check if specific values are part of the table for a concrete instance of a SNARK. We will now give a simple example to demonstrate the idea of this approach [31].

Example. Suppose we have three 1-bit variables, a, b, and c, where c is the XOR ($\oplus$) of a and b. We create a truth table with valid combinations of our variables as shown in Table 2.2. This truth table now also serves as our lookup table. For any instances of a, b, and c, we check if the tuple (a, b, c) exists as a row in the lookup table. If the row exists, our tuple (a, b, c) satisfies the relation $a \oplus b = c$.

The rest of this section defines the construction of this lookup technique within a SNARK or a STARK setting. Section 2.5.1 is rather informal and helps the reader understand the principles of plookup. Sections 2.5.2 and 2.5.3 formally explain the math we use in the solution of the present thesis.

2.5.1 Multiset Equality Checks

Let $\{t_i\}_{i\in[d]}$ be our lookup table with d elements, where each t_i is an element of $\mathbb{F}_p$. Let $\{f_i\}_{i\in[n]}$ be the values we want to check against our table $\{t_i\}_{i\in[d]}$. Each f_i is also an element of $\mathbb{F}_p$. As shorthand we write f for $\{f_i\}_{i\in[n]}$ and t for $\{t_i\}_{i\in[d]}$. We want to show that the values in f form a subset of the values in t. If $f \subseteq t$ then all the values of f are in t and therefore the values in f are considered valid [31].

We perform this subset check via multiset-equality checks. Let us introduce s, the multiset union of f and t, sorted by t. Then we look at the differences of adjacent values inside s and t, resulting in the difference sets s' and t' where $|s| = |s'| + 1$ and $|t| = |t'| + 1$ ($|\cdot|$ denotes cardinality, the number of elements in the multiset). If s' and t' are multiset equal ignoring zeroes and s is the multiset union of f and t sorted by t, then $f \subset t$ [31, 3].

Example. Let $f = \{1, 3, 4\}, t = \{1, 3, 4, 7\}$. We construct s which is the multiset union of f and t and we sort s by the values in t. $s = \{1, 1, 3, 3, 4, 4, 7\}$. Let us now construct the difference set s' from s, where s' is defined as $\{s_1 - s_0, s_2 - s_1, .., s_{n+d-1} - s_{n+d-2}\} = \{0, 2, 0, 1, 0, 3\}$. We construct t' in the same way as s'. $t' = \{t_1 - t_0, t_2 - t_1, .., t_{d-1} - t_{d-2}\} = \{2, 1, 3\}$. We already see that s' and t' comprise the same elements ignoring zeroes. By adding n zeroes to t', s' and t' are multiset equal [31, 3, 54].

This construction is not secure yet. We need to introduce randomness to make the check sound. We choose random $\beta \in \mathbb{F}_p$, and instead of calculating $s' = \{s_{i+1} - s_i\}_{i \in [n+d-1]}$, we calculate $s' = \{s_i + \beta \cdot s_{i+1}\}_{i \in [n+d-1]}$. When $s_i = s_{i+1}$, the differences are not zero anymore, but the differences are a multiple of $1 + \beta$ (due to $s_i + \beta \cdot s_i = s_i \cdot (1 + \beta)$). This allows us to decide if $s_i = s_{i+1}$ [31].

2.5.2 Polynomial Equality Checks

Let f, t, and s be multisets, as defined in Section 2.5.1. We now interpret f, t, and s as vectors, each element in each multiset has a unique and fixed index (ascending integer enumeration of multisets). Let $\beta, \gamma \in \mathbb{F}_p$. We now define bi-variate polynomials F and G over f, t, s, β, and γ as [31]

$$F(\beta, \gamma) := (1 + \beta)^n \prod_{i \in [n]} (\gamma + f_i) \prod_{i \in [d-1]} (\gamma(1 + \beta) + t_i + \beta t_{i+1}) \tag{2.9}$$

$$G(\beta, \gamma) := \prod_{i \in [n+d-1]} (\gamma(1 + \beta) + s_i + \beta s_{i+1}) \tag{2.10}$$

The authors of plookup [31] prove the following statement about F and G:
Observation. $F \equiv G$ iff

1. $f \subset t$, and
2. s is the multiset union of f and t, sorted by t.

In practice, we can check if $F \equiv G$ by choosing random $\beta, \gamma \in \mathbb{F}_p$ and checking if $F(\beta, \gamma) - G(\beta, \gamma) = 0$. According to the Schwartz-Zippel lemma (as introduced in Section 2.2) this check fails w.h.p. if $F \not\equiv G$.

2.5.3 Vector Lookups

The approach so far only works for one-dimensional elements since we compare multisets consisting of finite field elements. We can use the following randomized approach to support tuple checks.

Choose random $\alpha \in \mathbb{F}_p$. Let w be the number of columns in the lookup table. We now reduce one row to one single finite field element with $t := \sum_{i \in [w]} \alpha^i t_i$ and $f := \sum_{i \in [w]} \alpha^i f_i$ for each tuple $\{f_i\}_{i \in [w]}$ and $\{t_i\}_{i \in [w]}$ in our tuple sets f_{tuple} and t_{tuple}. With this reduction we can now use the approach described in Section 2.5.2. The security of this compression technique again relies on the Schwartz-Zippel lemma described in Section 2.2 [31].

Chapter 3

Problem Definition and Context

This chapter defines the problem we aim to solve and elaborates on the context and motivation that raise this problem. Our industry partners also provided us with an example use case on which we base our practical work.

3.1 Problem Statement

As the prover, we want to use a non-interactive zero-knowledge proof to prove that a public output y was created by applying a public algorithm $\mathcal{A}$ on private data x. By also computing the hash value $H(x)$ of private data x within the proof system, we bind public output y to private input x by including the hash $H(x)$ as part of the public output y.

The result is private x, public $\mathcal{A}$, $H(x)$, y, and a proof π. The proof π cryptographically ensures the desired relationship between the components. Any verifier is convinced that y is the output of applying $\mathcal{A}$ to the input x, which hashes to $H(x)$. Figure 3.1 illustrates this idea, where blue variables describe public information, and red variables private information. The prover computes $H(x)$, y, and π and publishes $\mathcal{A}$, $H(x)$, y, and π. The verifier is anybody who has access to the public values.

Furthermore, we have the requirement that the input data x is large (multiple Gigabytes). Verifying the proof shall be doable on a client (e.g., laptop or desktop PC) within a reasonable amount of time (e.g., less than 10 seconds). The size of the proof shall be small, where small refers to less than 10 Megabytes. The proof will be stored on disk and transferred over the network. However, the proof will not be stored on a blockchain. Therefore, we do not care about the proof size too much. (Other use cases of zk-proofs require small proof sizes and a simple verification step, as the solution is part of blockchain protocols like Zcash [9], which is not the case for our scenario).

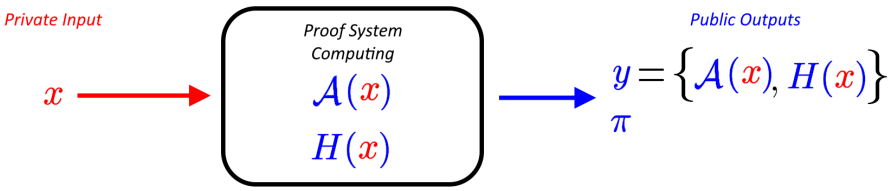

Figure 3.1: Overview and Design of the Proof System.

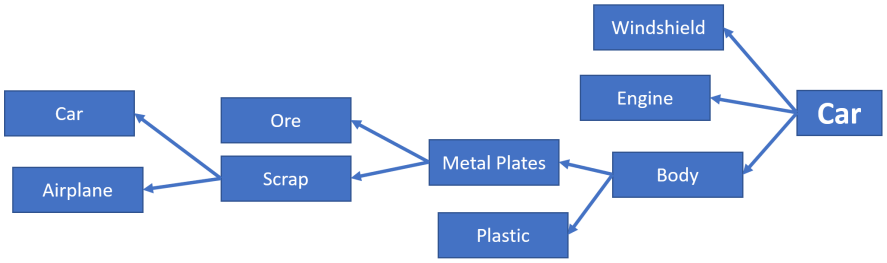

Figure 3.2: Example of a certification tree. Blue boxes describe one certificate. Arrows visualize pointers to input products.

The goal is to develop a performant solution to the given problem statement, trying to minimize the computational efforts of the prover. Today's proof systems focus on the succinctness of the proof, making the verification step efficient; therefore, we probably do not need to optimize on the verifier side. The prover requires powerful hardware to create such proofs, as a client (laptop, desktop PC) does not provide sufficient computational power for today's proof systems.

3.2 Context and Motivation

The design of the proof system described in Section 3.1 provides privacy in a general and abstract supply chain setting. This section briefly outlines the enhancements provided by this design.

Context. Our industry partner S1Seven [64] focuses on realizing a digital supply chain. The purpose is to foster trust between business partners by providing more transparency of the involved products, their origin, quality, and means of production. The implementation uses certificates (using cryptographic signatures) for each product, a digital identity. Certificates hold information about the quality of the product, the origin of the input products, and the treatment/production steps for transforming the input products into the output product. Input products get referenced by their certificate, serving as their digital identity.

The result of the vision of a fully digital supply chain is a certification tree, where each product refers to its input products, and those input products refer to their input products. Figure 3.2 shows an example certification tree for a car. A tree of certificates provides excellent potential for various use cases, such as automated checks/computations on the certification tree, for example:

- What is the total CO_2 footprint of this product? (Summation of all CO_2 emissions of the children)

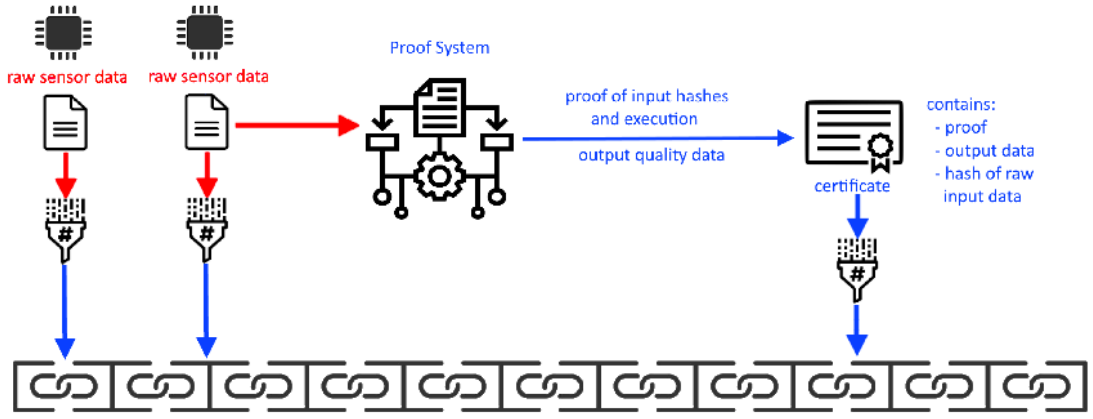

Figure 3.3: Context of the proof system within the notarization phase.

- Do all components conform to the standards accepted by our company? (e.g., aircraft and car: are all sub-components, such as metal plates, screws, engine, windshield, seats, .. certified to fit our requirements?)

- Statistics/Fun: Enumerating all companies/countries involved in creating the product.

Certification may go beyond the lifetime of a product, as scrap has a digital identity too. For example, the certificate for an aluminum snow shovel might link back to a car, which has been scrapped and recycled.

Other areas benefit from this certification tree design as well. The pharmaceutical industry could prove the origin of its products and effectively mitigate counterfeits [48]. The food industry may provide in-depth details about the origin, quality, and standards of the products. Consumers strive for more transparency in this area, one example being the referendum to disclose the origin of food products in Austria [69].

One major use case is the certification and traceback of green steel [85]. The steel industry is one of the main contributors to global CO_2 emissions since the traditional primary production route reduces iron ore using coke, where CO and CO_2 are by-products. The ore may also be reduced using hydrogen instead of coke, significantly reducing CO_2 emissions. The problem with green steel is that it cannot be distinguished from steel coming from the carbon-reduction route. A digital identity helps to tell green steel apart from steel coming from the carbon reduction route.

Motivation. Being transparent with the production steps raises concerns regarding confidentiality and competitiveness. Companies often have secret and optimized production processes to produce more efficiently or of higher quality than their competitors. We now have two seemingly contradictory requirements. We wish for transparency in the production processes, but we want to keep the processes private.

This requirement calls for the use of zero-knowledge-proof systems. Our industry partner S1Seven [64] utilizes blockchain notarization of certificates (see Section 2.3.3). They came up with a concept within the scope of the steelmaking process: The idea is to notarize the plain data captured by sensors during production. This first step serves as a commitment to the measurements. Evaluating sensor data allows for deducing the quality of the material. Computing the quality gets done inside a proof system, which ties the resulting quality data together with the hash of the private sensor data, as the hash gets computed within the proof system.

The resulting output data (qualitative statements and hash of input data) is placed inside the product certificate, alongside a proof of computation. The certificate gets notarized. We linked the derived qualitative data and the confidential sensor data via the input's hash value, where the idea is visualized in Figure 3.3.

A strong assumption empowers this approach. The system of sensors and their blockchain interface need to be trusted. The system presented in Figure 3.3 moves the trust from the producer who computes the qualitative data to the sensors. However, the sensors are also controlled by the producer. We have moved the trust from the producer to the producer (yes: in fact, we did not create any security improvements yet). An open research question is how to make the sensors trustworthy, such that the producers cannot create fake statements about their product's quality and that we effectively move the trust from the producer to the sensors. The sensors shall play the role of an independent notary who documents their observations in the production process. This notary must not be controlled or influenced by the producer.

Steelmakers need to provide an inspection certificate when selling products. These inspection certificates traditionally require mechanical tests on the material to be sold. In 2021, the standard DIN EN 10373 [78] was published, allowing the provision of qualitative data using computational models inside the inspection certificate, which perfectly fits the proof system design demonstrated in Figure 3.3. This proof system enhances the new standard by increasing transparency and trust, as the standard allows deriving qualitative data without mathematical proofs. Furthermore, the standard provides potential solutions to the open research question on how to increase trust in the hardware (sensors).

3.3 Example Use Case: Statistics on a Video

Our industry partners Technologies4you [28] and DI Heinz Basalka (EPU) [27] provided us with an exemplary problem instance. They focus on deducing qualitative statements about additive manufacturing (3D printing) processes by observing the welding process. Metrics like electric current and voltage, wire feeding speed, and welding arm movement speed get recorded for later evaluation. Another important metric is a video of the welding process captured by an infrared camera.

Additive manufacturing of metal works by deposition welding, where multiple layers get added on top of each other. Many mistakes can be found early on without doing destructive tests by observing the production process. Defects like holes or malformed

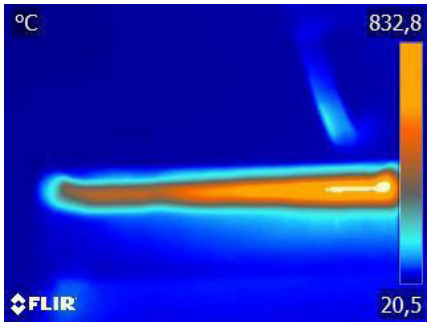

Figure 3.4: Welding image obtained by FLIR thermal imaging camera [53].

regions can easily be spotted by evaluating the video. Figure 3.4 shows an infrared image of a welding process.

A simple metric for deducing the quality of the additive manufactured product is statistics on the temperature of the hot solid area. The temperature should be as constant as possible. Statistics, like the average, standard deviation, minimum, and maximum, already provide a good hint on the quality of the product. Deriving quality from the video provides us with the example use case for our problem:

- The video is sensitive data, as it shows the product's construction in detail. This information must be kept private for competitive reasons.
- Statistics on the temperature in the hot regions give a good hint about the quality. We want to publish these statistics.
- Video recordings can be considered huge input data, at least in a zero-knowledge-proof setting.

Formally, we want to build a zero-knowledge proof system that computes statistics (standard deviation, average, minimum, and maximum) on a predefined, public region of influence (ROI) of the video. Figure 3.5 shows a possible example of an ROI for the hot solid areas, where a green ellipse delineates the ROI. Alongside calculating the statistics, we want to compute the hash of the whole video within the proof system for linking the statistics to the private video. The hash of the video serves as a public commitment to the video.

The video provided by the industry partner has a resolution of 382x288 pixels, a framerate of approx. 12 frames per second, a total number of 14794 frames, and the playback time of the video is 18 minutes and 28 seconds. One pixel describes the temperature in tenths of degrees centigrade and is stored as a 16-bit unsigned integer. Therefore, the size of the uncompressed video is around 3104 MB (382x288x14794x2 Bytes).

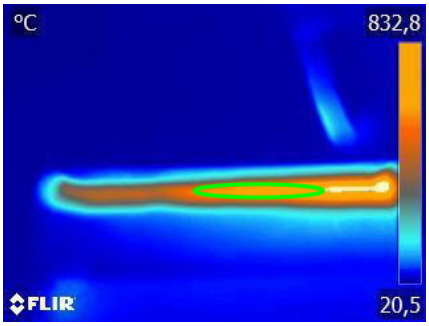

Figure 3.5: Example region of influence (green ellipse) for computing statistics on a thermal image for a welding process [53].

Our industry partner provided us with a compressed version (gzip) of the video using the Hierarchical Data Format (HDF)[1], where the file size is approx. 1119 MB. We only work with the uncompressed version for simplicity, as an additional gzip-decompression step within the proof system is not expedient for today's proof systems.

[1] `https://www.hdfgroup.org/`

Chapter 4

Methodology

This chapter elaborates on the solutions to the problems given in Chapter 3. Section 4.1 discusses the choice of technology and library, and Section 4.2 introduces the library we use, which is Winterfell. Section 4.3 discusses how the requirements from Chapter 3 can be realized with Winterfell and points out some limitations. The various ideas presented in this chapter get crafted into unified implementations in Chapter 5.

4.1 Choice of Technology

We have introduced various zk-protocols in Section 2.4. Today's technologies are dominated by SNARKs and STARKs. We decided to work with STARKs for the following reasons: [10, 34, 52]

- STARKs are transparent, meaning they do not require a trusted setup phase, as SNARKs do. We want to be independent, and not having a trusted setup is more attractive.
- STARKs allow skipping computational steps compared to SNARKs, where each computation requires its own gate.
- STARKs rely on a handful of security assumptions and are post-quantum secure, whereas SNARKs have stronger security assumptions and are not post-quantum secure.
- Proofs of STARKs are bigger than proofs of SNARKs. SNARK verification time also outperforms STARK verification time. However, the difference in verification time is negligibly small, and the verification step is speedy for both technologies.

In August 2021, Facebook released the comprehensive STARKs library Winterfell on GitHub [46]. We picked Winterfell, as the library appears very attractive and general. Section 4.2 introduces the library and shows how to use it.

4.2 STARKs Library Winterfell

Ben-Sasson et al. defined STARKs in 2018 [10], and the library Winterfell [24] closely sticks to the protocol definition found in the STARKs paper, causing Winterfell to be a very general implementation. We use Winterfell 0.3.0 and Rust 1.59.0.

Winterfell is written in Rust, and Section 4.2.1 briefly introduces the language. The library nicely abstracts away the intricate parts of the protocol and leaves the programmer with defining the Arithmetic Intermediate Representation (AIR), where Section 4.2.2 highlights how to implement the AIR. Finally, Section 4.2.3 elaborates on basic building blocks for writing simple programs. We introduced STARKs in Section 2.4.1, explaining the general structure and functionality of the protocol. The *StarkWare Team* published the *ethSTARK Documentation* [72], which uses terminology similar to Winterfell's implementation. This resource might be considered additional material for understanding the STARK protocol.

4.2.1 Rust

We briefly introduce some terminology and basic concepts of the language. There exist some object-oriented concepts which are defined over Structs and Traits. Structs are similar to Structs known from C, where a Struct defines a datatype and is built up of various members. Classes do not exist in Rust. A Trait is similar to an Interface known from Java, where a Trait defines function signatures (and member variables). A Trait definition defines the interface, and a Trait gets implemented for Structs, defining each Struct's (different) behavior. A Class known from C++/Java/C# would be similar to a Struct in Rust having implementations for various Traits [30].

Rust comprises many more features, and we ask the reader to consult the Rust documentation for further details [30]. We require the terminology of Traits and Structs, as we will use these terms in Section 4.2.2.

4.2.2 Implementing a STARK

Winterfell is available as a Crate [22] on `crates.io`[1] and can be used by adding a dependency to the `Cargo.toml` file [23]. Alternatively, the source code is available on GitHub [24]. We will now explain the most important parts of using Winterfell, and detailed information can be found at the Rust documentation of Winterfell [23].

Developing a STARK is ideally performed on paper first, as working with Winterfell feels like translating an already known STARK to Rust-Winterfell source code. Engineers who develop solutions while writing code shall be warned that designing a STARK by writing code using the Winterfell library is very difficult. Colleagues at our institute reported similar problems with other proof systems, for example in the SNARKs area.

We tend to compare the current state of proof systems with Assembler vs. the high-level language C. In C, engineers use variable names and functions to define logic. For example, defining a function call in Assembly Language (e.g., x86) involves creating the stack frame (allocating space for the function) and resolving variable names to indices within the current stack frame. Developing a STARK within Winterfell requires accessing variables (AET columns) via indices in the state array. The width of the AET must also be explicitly defined. This feels similar to Assembly Languages, where variables are accessed via indices in the stack frame, and the stack frame size must also be defined.

[1] `https://crates.io/`

The current state of the art may be improved by domain-specific languages that provide benefits like C does to Assembly Languages, calling for future work (see Section 7.2).

Once a STARK is fully defined on paper, we can translate it to Winterfell. A complete definition of a STARK includes:

- The algebraic execution trace (AET): trace width and trace length.
- Transition Constraints: Algebraic relations between two adjacent states (rows) in the AET and the polynomial degree of these constraints. An algebraic relation between two states must form a low-degree polynomial, allowing addition and multiplication.
- Boundary Constraints: Some steps of the AET need to be fixed and made available to the public, such as the initial value (IV) and final digest when computing a hash. Other examples include the result of some computation at the end and beginning of the trace.
- Public Inputs: We need to define the structure of information we want to release to the public, including the results of computations and the hash digest.
- Periodic Columns: Round constants for hash functions and flags for the computation need to be defined for using them together with the transition constraints.
- Finite Field: We need to be aware of the size of the finite field, as this defines the maximum value our numbers can take. It is essential to keep this limit in mind when doing computations. Furthermore, different finite fields have different properties (security, computational efficiency).

Public Inputs. A Struct defines the members of the public inputs. This information is sent together with the proof and the description of the AIR. Therefore the Trait `Serializable` must be implemented by the Struct defining the public inputs.

Algebraic Intermediate Representation (AIR). We define a Struct holding data required within the AIR. This includes some boilerplate for the `AirContext`, and the remaining member variables are specific to the STARK we implement.

The Trait `Air` must be implemented for the Struct AIR. This Trait defines the datatype of the public inputs (the Struct we defined previously) and the finite field we use. Furthermore, the transition constraints (including a description of the polynomial degrees), boundary constraints, and periodic columns get defined within the `Air` Trait.

The function `new` (see Listing 4.1) creates a new instance of an Air. The parameters are *information about the trace* (not relevant for programmers), *public inputs*, and *parameters* of the STARK protocol. Necessary information from the public inputs gets stored within the Struct AIR, and the proof parameters get passed further down to the proof system. Engineers must define the **polynomial degrees** of the **transition constraints** in this function.

Listing 4.1: Function Signature `new` of Trait `Air`

```
fn new(trace_info: TraceInfo, pub_inputs: Self::PublicInputs, options:
    ProofOptions) -> Self;
```

Transition constraints get evaluated inside the function `evaluate_transition` (see Listing 4.2). The parameters are the *evaluation frame* (containing the *current* and *next* state of the computation), *periodic values* (concrete assignments of the *periodic columns* for the present transition), and a vector *result*, which stores the evaluation of the constraint evaluation polynomials for the current transition. The vector *result* needs to hold zeroes only for every transition within the AET to be a valid STARK.

Listing 4.2: Function Signature `evaluate_transition` of Trait `Air`

```
fn evaluate_transition<E: FieldElement<BaseField = Self::BaseField>>(
    &self,
    frame: &EvaluationFrame<E>,
    periodic_values: &[E],
    result: &mut [E],
);
```

Boundary constraints get defined in function `get_assertions` (see Listing 4.3). These constraints fix registers (one "cell" in the AET) at specific steps to constants, public inputs, or values derived from public inputs.

Listing 4.3: Function Signature `get_assertions` of Trait `Air`

```
fn get_assertions(&self) -> Vec<Assertion<Self::BaseField>>;
```

Periodic columns are defined in function `get_periodic_column_values` (see Listing 4.4), where all values are pre-computed inside the function, and the result is returned as a `Vector`.

Listing 4.4: Function Signature `get_periodic_column_values` of Trait `Air`

```
fn get_periodic_column_values(&self) -> Vec<Vec<Self::BaseField>>;
```

Algebraic Execution Trace (AET) Instance. A trace instance itself is a table of plain finite field values, holding our private input and computations. Winterfell requires the AET to be in the format of the `TraceTable` Struct. There are various ways of coming up with an instance of a `TraceTable` Struct, and the Winterfell Documentation [23] elaborates on these options.

Proof Creation and Verification. We have now collected and implemented all required parts and are ready to compute a proof and verify that proof. Creating proof requires that we have an AET instance, the associated public inputs, the AIR description, and the STARK protocol parameters. The verification step uses the proof, the public inputs, and the AIR description. The library takes over the remaining parts and exposes functions like prove and verify. The Winterfell documentation [23] provides an excellent example on the landing page for using these interfaces.

4.2.3 Building Blocks

This section shows basic principles for creating simple programs within the STARK protocol. These building blocks are general to STARKs and not specific to the Winterfell library.

Periodic Columns

Many computations use constant values that repeat after some time. An example is computing the hash value, where one permutation consists of multiple rounds, where each round has its unique constants. Hashing data involves invoking many permutations of the hash function, where the round constants get re-used after some time.

This approach is also known outside of the Winterfell library. For example, Buterin V. uses a similar approach to compute rounds of MIMC [1] in his blog post from 2018 [81].

Furthermore, we can disable and enable certain checks periodically by using flags as additional periodic columns. A flag either takes 0 or 1 as a value, and the desired transition constraint polynomial is multiplied by that flag. If the flag is zero, the whole polynomial evaluates to zero, and therefore, the constraint only gets enforced when the flag is one.

A practical example for periodic columns is a nested for-loop in procedural programming. The outer loop iterates over the permutations of computing a hash function. The inner loop iterates over the different rounds within one permutation, where each round requires its unique constants. Additionally, assume that the first round must be computed differently from all other rounds. We can use a periodic column flag to distinguish which round we are currently computing.

Recall that we do not know which step we currently are at when evaluating transition constraints. We only know the current and next state, but not the position in the AET. We would not know which round constants to use without periodic columns, as we do not know which round we are in.

Domain Separation using Flags

Imagine the scenario that we have two separate computations which get executed sequentially. An example would be one loop followed by another loop in procedural programming. We need to enforce the constraints for both loops simultaneously, but we use a flag in the AET to define which loop we are currently executing.

Boundary constraints and transition constraints enforce that the flag has the correct value at each step. For example, when iterating over a list within a loop, we know the length of the list - this would be a public input. As the length is public information, we can create a boundary constraint that flips the flag at that step where the computation of the first loop ends. Transition constraints ensure that the flag either transitions from 1 to 1, from 0 to 0, or from 1 to 0. Therefore, the flag may only transition from 1 to 0 a single time in the whole AET. Two boundary constraints ensure the flag to be 1 at step x and 0 at step $x + 1$. Step x would be the step where the computation of the first loop ends.

Table 4.1: Example AET for computing the sum and product of elements in the set s.

Step i	s	*sum*	*prod*
0	?	**0**	**1**
1	3	3	3
2	7	10	21
3	4	14	84
4	9	**23**	**756**

This approach enables a clear domain separation for performing different computations at different intervals.

Running Sums and Products

Let s be a set of values where we want to compute the sum of its elements. Equation 4.1 shows how to obtain the sum of all elements in the set s.

$$\text{sum} = \sum_{e \in s} e \tag{4.1}$$

We can compute the sum in a STARK using a running sum approach. This approach is shown in Listing 4.5 as pseudocode for procedural programming.

Listing 4.5: Iterative approach to compute the sum of elements inside the set s (procedural programming pseudocode)

```
1 function sum(Set s) -> Integer:
2     Integer result = 0
3     for each e in s:
4         result += e
5     return result
```

Table 4.1 shows the AET of a STARK computing the sum for the example set $s = \{3, 7, 4, 9\}$. The column s holds all values of the set s, whereas the column *sum* computes the sum of all elements in s. We omit the exact definition of boundary and transition constraints. *sum* at step 0 must be 0 (the additive neutral element); this is similar to the initialization step in line 2 of Listing 4.5. The final computation is found at step 4 in the column *sum*, which would be linked to the public inputs (the sum and probably the size of the set s) via a boundary constraint.

The same approach can be used to compute the product of the example set $s = \{3, 7, 4, 9\}$, where the computation is already included in the AET shown in Table 4.1. This computation is identical to computing the sum, except that we use multiplication

Table 4.2: Example AET for a multiset equality check.

Step i	a	b	z
0	?	?	$\mathbf{1}$
1	3	7	$1 \cdot \frac{3+\lambda}{7+\lambda}$
2	4	3	$1 \cdot \frac{3+\lambda}{7+\lambda} \cdot \frac{4+\lambda}{3+\lambda}$
3	7	4	$1 \cdot \frac{3+\lambda}{7+\lambda} \cdot \frac{4+\lambda}{3+\lambda} \cdot \frac{7+\lambda}{4+\lambda} = \mathbf{1}$

instead of addition. Furthermore, we need to use the multiplicative neutral element 1 in step 0 for the column *prod*.

Randomized AIR with Pre-Processing (RAP)

The idea of running products can be extended to perform more powerful computations. As an example, we want to show that the two multisets $a = \{3, 4, 7\}$ and $b = \{7, 3, 4\}$ are multiset-equal (i.e., the set a contains the same elements as set b). We construct two polynomials $A(\lambda)$ and $B(\lambda)$ from our sets to show multiset equality between a and b. Equations 4.2 and 4.3 show these polynomials for sets a and b, respectively [5].

$$A(\lambda) = (3 + \lambda)(4 + \lambda)(7 + \lambda) \tag{4.2}$$

$$B(\lambda) = (7 + \lambda)(3 + \lambda)(4 + \lambda) \tag{4.3}$$

Multiplication is commutative, and we can see that the polynomials $A(\lambda)$ and $B(\lambda)$ are equal. The equality of two polynomials can be checked by evaluating both polynomials at a randomly chosen point (here λ) according to the Schwartz-Zippel lemma, as discussed in Section 2.2. To check multiset equality in a STARK, we use the helper column z for accumulating all values for our polynomials. Note that division is emulated by multiplication with the inverse element in the finite field [5].

$$0 = z' \cdot \frac{a'' + \lambda}{b'' + \lambda} - z'' \tag{4.4}$$

Table 4.2 shows the AET for our STARK, while Equation 4.4 lists the transition constraint, and Equation 4.5 defines the boundary constraints. The definition of the sequence $\{z_i\}$ forms two polynomials, one being the numerator and the other the denominator. After having accumulated all values for all polynomials, then $z_3 = \frac{A(\lambda)}{B(\lambda)}$. If the polynomials are equal, then the fraction equals 1. We need a source of randomness for λ, which we assume we have for now. Section 4.3.8 describes how to draw randomness in our STARKs setting [5].

$$1 = z_0 = z_3 \tag{4.5}$$

As we work with a non-interactive protocol, the only way to obtain randomness is by using the Fiat-Shamir heuristic. The name *randomized AIR with pre-processing* comes from the fact that column z depends on random λ, and the randomness origins from columns a and b [5].

Winterfell does not support native RAPs in the version we are using (v0.3.0) [41]. We recognized this while working on this thesis. Therefore, we need to apply the Fiat-Shamir heuristic manually when designing a STARK, and Section 4.3.8 shows how to realize this. While making the final changes to the thesis, Winterfell version v0.4.0 was released, containing native RAPs support. With this new version, the Fiat-Shamir heuristic does not need to be computed inside the AET anymore, causing a significant speedup.

4.3 Realization and Limitations

This section highlights the approaches used to develop an implementation using the Winterfell library and presents solutions to problems arising in the STARKs space. We tackle the following challenges:

- Hashing the input.
- Computing statistics on the input.
- Restricting statistics computation to a region of influence.

The input is a video being a series of frames. Each frame consists of pixels. For all solutions except for the ROI approach, we interpret the input as a vector of pixels. A pixel is a discrete temperature value measured in degree centigrade. The original video provided by our industry partner measures temperature in tenths of degrees centigrade and uses 16-bit words, where one word corresponds to one pixel. Computing statistics on an ROI requires the concept of frames, where we will use Winterfell's periodic column feature where one cycle processes one frame of the video.

Sections 4.3.1 to 4.3.3 elaborate on the three challenges listed above, while Sections 4.3.4 to 4.3.8 present approaches for the problems occurring in Sections 4.3.1 to 4.3.3. Section 4.3.9 expands on a potential optimization technique.

4.3.1 Hashing the Input

The performance of computing a hash inside a STARK heavily depends on the hash function in use. Computations inside a STARK are limited to a handful of operations described in Section 2.4.1, where more complex operations, such as bitwise operations, need to be built on top of the existing ones, causing a massive overhead. Hash functions

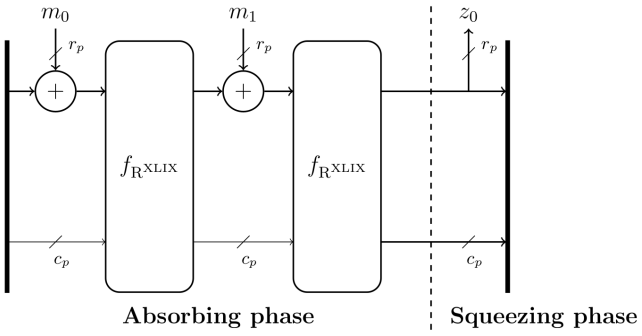

Figure 4.1: Rescue-Prime with two absorbing iterations [75].

like SHA-2 [71] make heavy use of bitwise operations, and arithmetization is expensive. The need for computing hashes in zk-proof systems led to the development of arithmetization-friendly hash functions that rely on operations natively supported by the proof systems [11].

We now introduce the hash function Rescue-Prime [2], which uses a sponge construction (see Figure 4.1), where one permutation consists of N rounds. Figure 4.2 shows one round of the Rescue-XLIX (*Rescue Forty-Nine*) permutation.

The multiplication with the MDS matrix and adding constants are not of interest, as these values are constant and do not influence the degree of our transition constraints. The exciting part is the exponentiation with α and α^{-1}. α is a small value satisfying $\gcd(\alpha, p-1) = 1$, and α^{-1} is huge. Exponentiating a state by a small α yields a small degree for the transition constraints. Doing exponentiation with α^{-1} gives a transition constraint with a very high transition constraint degree which is non-beneficial. Therefore, arithmetization is done for each round using a meet-in-the-middle (MITM) approach,

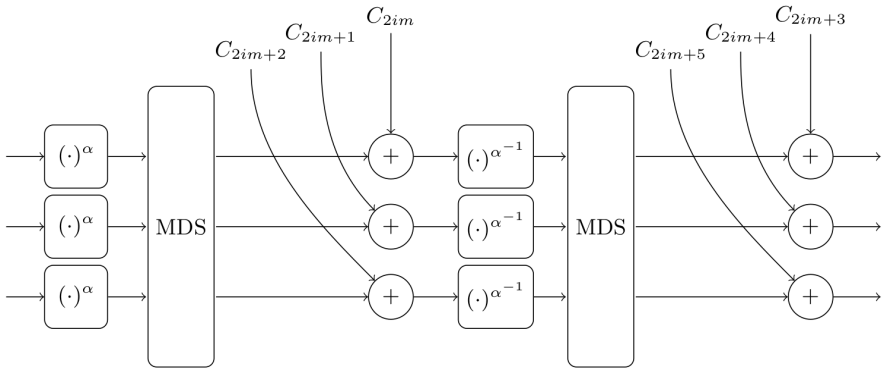

Figure 4.2: Round i of the Rescue-XLIX permutation, with state size $m = 3$ [75].

resulting in an execution trace that requires one state for each round of the Rescue-XLIX permutation [2].

Constructing the execution trace does not require sticking to certain degree bounds as in the case of transition constraints. We compute the whole execution trace in the forward direction. The transition constraint for one round computes forwards from the current state (exponentiation with α) and backwards from the next state (where the inverse of α^{-1} is α). The results get compared and are equal if the execution trace was computed correctly. With this approach, the degree of the transition constraint is α [2].

Periodic columns provide adequate round constants for each round. A flag being part of the periodic columns indicates when to absorb elements into the hash state and when round computation checks must be performed. One periodic cycle corresponds to one Rescue-XLIX permutation. Section 5.2 goes into detail on constructing a STARK comprising Rescue-Prime.

4.3.2 Computing Statistics

The **arithmetic mean (average)** of the input vector is the sum of all elements divided by the number of elements in the vector [86]. Computing a running sum was presented in Section 4.2.3, and the size of the vector is public information being available inside the STARK (as we need to know at which step the boundary constraints need to check for the computed results).

A division in a finite field is emulated by multiplication with the inverse element and only possible when there is no remainder when striving for the semantics known from real/rational numbers. Since the average is often a fraction, we need to describe the average as $\text{avg} = a \cdot b + r$, where avg, a, b, $r \in \mathbb{F}_p$. As the length l of the vector is publicly known, we can simplify the description by providing the sum of all values as output to our STARK computation. The receiver of the proof may then calculate the average with $\text{avg} = \frac{\text{sum}}{l}$. Here we need to be careful not to overflow inside our finite field when adding up many numbers.

Computing the **standard deviation** σ inside a STARK requires the concept of the square root, which involves fractions in many cases as well. Therefore we simplify the approach by computing the sum part of the variance σ^2, the squared standard deviation, according to Equation 4.6 [86] using the running sum approach from Section 4.2.3 inside the STARK. We omit the division and place the numerator inside the public inputs. The average origins from the public inputs, where we define an extra slot for the average rounded to a whole number, where the verifier can check the rounded average is correct outside the STARK protocol. The standard deviation is defined by $\sqrt{\frac{\text{variance sum}}{l-1}}$, where $\text{variance sum} = \sum_{i=1}^{l}(a_i - \text{avg})^2$.

$$\sigma^2 = \frac{\sum_{i=1}^{l}(a_i - \text{avg})^2}{l-1} \tag{4.6}$$

The standard deviation calculation introduces an error due to rounding the average to a whole number. The error becomes insignificant when using a large input vector

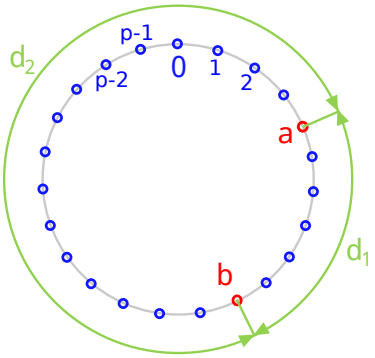

Figure 4.3: Distances d_1 and d_2 between two finite field elements $a, b \in \mathbb{F}_p$ (here $\mathbb{F}_{23}$).

and numbers from a domain being big enough. We observe that the error is negligibly small ($< 10^{-3}$) for our example use case having a vast input set and a domain with a range of approximately $[5000, 18000]$. However, the impact of this rounding error needs to be evaluated for every use case in practice. For the ROI example, we only analyze the "hot" areas of the video, which restricts the domain further. It shall be noted that the error is $< 10^{-2}$ for a range of $[0, 100]$, which equals a temperature range of ten degrees centigrade.

Computing our sequence's **minimum and maximum** value is impossible with native finite field operations since there is no ordering of elements in a finite field. We now briefly outline why this is the case, and Sections 4.3.5 and 4.3.6 describe two possible solutions for this type of computation.

Inequalities and Finite Fields. The general conception of numbers allows us to quickly tell whether $a < b$ for two numbers a, b (for natural numbers, rational numbers, ..). There is also the concept of the distance between two numbers a, b being uniquely defined as $|a - b|$ yielding one solution. We can also argue about natural numbers, where every natural number a has one successor, $a + 1$, being greater than a. There exists an ordering of elements in these infinite fields.

However, for finite field arithmetic, these intuitive rules do not apply. There is no ordering of elements in finite fields as we know them from natural numbers. The distance between two finite field elements is not uniquely defined and two solutions exist (there is also no such thing as an absolute value). Figure 4.3 shall endorse the statements given, where field elements are arranged in a cyclic order (modular arithmetic is sometimes referred to as "clock math" due to its similarity with analog clocks [83], see Section 2.1). Figure 4.3 shows two finite field elements and their distances between them and the elements in the proximity of the additive neutral element 0.

The **median** [86] is another statistic being challenging to calculate within a STARK. Section 4.3.7 describes an approach similar to computing the minimum and maximum values.

4.3.3 Statistics on a Region of Influence

Section 4.3.1 used the concept of a periodic cycle to capture the computation of one whole Rescue-XLIX permutation. We can use the same principle to process one frame at a time.

The video provided by our industry partner has a resolution of 382x288 pixels, so one frame consists of 110016 pixels. Assuming we process one pixel at one step, we can use a periodic cycle of length 131072, the closest power of two greater than 110016. The periodic columns now have two separate cycles, one for computing the permutations and the other for processing the frame. Computing the hash value is performed in any case (also if the pixels are outside the ROI) since we want to hash the whole video.

The periodic columns used for defining the ROI is one flag that either enables or disables the computation of statistics for each pixel. The ROI itself is public information encoded in the STARK's AIR, being available to the prover and the verifier. We can pack any STARK construction into this ROI approach, and Section 5.2.3 shows some implementations of STARKs operating on ROIs.

4.3.4 plookup Check

This section demonstrates how we construct a STARK for a plookup check (see Section 2.5 for a definition of plookup). The vector f contains our private input values and the size $|f|$ of f is public. As an example we want to show that all elements in f are within a certain range $[0, v-1]$, so we define t as $\{0, 1, 2, .., v-1\}$ and $|t| = v$. The range is public information. Furthermore, we need to define random β and γ, but for now we assume that we have some source of randomness to draw β and γ from. Section 4.3.8 describes how to draw randomness in our STARKs setting.

We start constructing our AET by splitting up F into two columns $F = F_f \cdot F_t$, where F_f, F_t, and G are defined as follows:

$$F_f(\beta, \gamma) := (1+\beta)^n \cdot \prod_{i \in [n]} (\gamma + f_i) \tag{4.7}$$

$$F_t(\beta, \gamma) := \prod_{i \in [d-1]} (\gamma(1+\beta) + t_i + \beta t_{i+1}) \tag{4.8}$$

$$G(\beta, \gamma) := \prod_{i \in [n+d-1]} (\gamma(1+\beta) + s_i + \beta s_{i+1}) \tag{4.9}$$

Our helper functions are the multiplicative Selector $S(a, f_x)$ (see Equation 4.10) which returns a when $f_x = 1$, and the multiplicative neutral element 1 for $f_x = 0$. The function T (see Equation 4.11) makes sure that a flag is either transitioning to the same value, or is going from 1 to 0. A transition from 0 to 1 makes $T \neq 0$. Function U (see Equation 4.12) ensures that a value is a flag ($\in \{0, 1\}$).

$$S(a, f_x) = a \cdot f_x + 1 - f_x \tag{4.10}$$

$$T(f_x', f_x'') = (f_x' - f_x'') \cdot (f_x' - f_x'' - 1) \tag{4.11}$$

$$U(f_x) = (f_x - 0) \cdot (f_x - 1) \tag{4.12}$$

We also use the flags f_s, f_f, and f_t as columns for defining the end of each vector. Furthermore, we need a helper column R for comparing F to G, where $R = F_f \cdot F_t - G$.

In step 0 we initialize our variables F_f, F_t, G, f_f, f_t, and f_s as 1. The flags f_f, f_t, and f_s are 1 when values of f, t, and s are available, respectively. s and t provide elements starting at step 0, whereas f starts listing elements at step 1 (we do not use the value of f_0). The transition constraints are defined in Equations 4.13 to 4.23, whereas Equations 4.24 to 4.28 list the boundary constraints.

$$0 = S((1 + \beta) \cdot (\gamma + f''), f_f'') \cdot F_f' - F_f'' \tag{4.13}$$

$$0 = S(\gamma \cdot (1 + \beta) + t' + \beta \cdot t'', f_t'') \cdot F_t' - F_t'' \tag{4.14}$$

$$0 = S(\gamma \cdot (1 + \beta) + s' + \beta \cdot s'', f_s'') \cdot G' - G'' \tag{4.15}$$

$$0 = F_f'' \cdot F_t'' - G'' - R'' \tag{4.16}$$

$$0 = (t'' - t' - 1) \cdot f_t'' \tag{4.17}$$

$$0 = U(f_f'') \tag{4.18}$$

$$0 = U(f_t'') \tag{4.19}$$

$$0 = U(f_s'') \tag{4.20}$$

$$0 = T(f_f', f_f'') \tag{4.21}$$

$$0 = T(f_t', f_t'') \tag{4.22}$$

$$0 = T(f_s', f_s'') \tag{4.23}$$

$$1 = F_{f_0} = F_{t0} = G_0 = f_{f_0} = f_{t0} = f_{s0} \tag{4.24}$$

$$0 = t_0 \tag{4.25}$$

$$1 = f_{f_{|f|}} = f_{t_{|t|-1}} = f_{s_{|f|+|t|-1}} \tag{4.26}$$

$$0 = f_{f_{|f|+1}} = f_{t_{|t|}} = f_{s_{|f|+|t|}} \tag{4.27}$$

$$0 = R_{|f|+|t|-1} \tag{4.28}$$

Table 4.3 shows an example instance for this STARK with $t = \{0, 1, 2, 3, 4, 5, 6\}$ and $f = \{5, 6, 2, 3, 2\}$. β and γ are not explicitly defined and all values depending on β and γ are denoted by δ_i, ζ_i, η_i, and θ_i. The symbol ? means that the value is not relevant and may take any value $\in \mathbb{F}_p$. l denotes the last step and must be at least $|s| - 1 = |f| + |t| - 1$.

Table 4.3: Example AET for performing a plookup check.

Step i	f	t	s	F_f	F_t	G	R	f_f	f_t	f_s
0	?	0	0	1	1	1	?	1	1	1
1	5	1	1	δ_1	ζ_1	η_1	θ_1	1	1	1
2	6	2	2	δ_2	ζ_2	η_2	θ_2	1	1	1
3	2	3	2	δ_3	ζ_3	η_3	θ_3	1	1	1
4	3	4	2	δ_4	ζ_4	η_4	θ_4	1	1	1
5	2	5	3	δ_5	ζ_5	η_5	θ_5	1	1	1
6	?	6	3	δ_5	ζ_6	η_6	θ_6	0	1	1
7	?	?	4	δ_5	ζ_6	η_7	θ_7	0	0	1
8	?	?	5	δ_5	ζ_6	η_8	θ_8	0	0	1
9	?	?	5	δ_5	ζ_6	η_9	θ_9	0	0	1
10	?	?	6	δ_5	ζ_6	η_{10}	θ_{10}	0	0	1
11	?	?	6	δ_5	ζ_6	η_{11}	$\theta_{11} = \mathbf{0}$	0	0	1
12	?	?	?	δ_5	ζ_6	η_{11}	$\theta_{11} = 0$	0	0	0
..	..	..	..	..	..	..	..	..	..	..
l	?	?	?	δ_5	ζ_6	η_{11}	$\theta_{11} = 0$	0	0	0

The STARK performs the multiset equality check $f \subset t$ as described in Section 2.5. This is done by evaluating the polynomials F and G (see Equations 2.9 and 2.10) at randomly chosen points β and γ. If the evaluations of F and G are equal, then w.h.p. the polynomials F and G are equal (according to the Schwartz-Zippel lemma, see Section 2.2).

The present AET uses the design pattern for accumulating a product step by step, as described in Section 4.2.3. Columns F_f, F_t, and G accumulate the products as defined in Equations 4.7, 4.8, and 4.9. Column R compares F to G. Since boundary constraints can only check a column at a certain step for being equal to a specific value, we need to formulate the comparison constraint using the additional column R. After having accumulated all values for the products, we can check R at step $|f| + |t| - 1$ to be zero. Due to the setup of our constraints, we may check column R at any step within $[|f| + |t| - 1, \text{last step}]$.

4.3.5 Minimum and Maximum via Vector Lookups

Defining the minimum and maximum of two finite field elements relies on comparing these elements, which is problematic, as elaborated in Section 4.3. To overcome this issue, we use plookup as described in Section 4.3.4 and extend this STARK.

As our example, we want to find the minimum and maximum of our private input vector a. We use the vector lookup approach described in Section 2.5.3. The sets t_l and t_h result from compressing all possible combinations of valid assignments of Equations 4.29 and 4.30 inside a given range $r = [x, y]$, where l and h stand for low and high, respectively. We use low and high as synonyms for min and max since low and high result in a short notation in the equations. The size of the range $|r|$ is $y - x + 1$, and the size of the tables is therefore $|t_l| = |t_h| = |r|^2$. Compressing field elements requires randomness for α. We assume that we have a source of randomness (as in Section 4.3.4) and Section 4.3.8 describes how to draw randomness in our STARKs setting.

$$t_l = \{a + b \cdot \alpha + \min(a, b) \cdot \alpha^2 : \forall a, b \in [x, y]\} \tag{4.29}$$

$$t_h = \{a + b \cdot \alpha + \max(a, b) \cdot \alpha^2 : \forall a, b \in [x, y]\} \tag{4.30}$$

To compute the minimum and maximum, we use the design pattern of accumulating elements one by one as described in Section 4.2.3. Column a holds our input values, columns min and max hold the accumulated statistics. To perform a lookup check in our tables, we need to compress our input. We do this by defining f_l and f_h as our compressed variants of the comparisons defined in Equations 4.31 and 4.32, which are also used as additional transition constraints. The multiplication with ${f_f}''$ in Equations 4.31 and 4.32 is not necessary for the constraints, but improves readability of the AET in Table 4.4.

$$0 = {f_f}'' \cdot \left(a'' + min' \cdot \alpha + min'' \cdot \alpha^2 - {f_l}''\right) \tag{4.31}$$

$$0 = {f_f}'' \cdot \left(a'' + max' \cdot \alpha + max'' \cdot \alpha^2 - {f_h}''\right) \tag{4.32}$$

We use the STARK from Section 4.3.4 twice, the first one uses $f = f_l$ and $t = t_l$, and the second one uses $f = f_h$ and $t = t_h$. The resulting STARK is shown in Table 4.4 as an example with $a = \{5, 6, 2, 3, 2\}$, $x = 1$, and $y = 7$.

α, β, and γ are not explicitly defined and all values depending on α are denoted with δ_i, ζ_i, η_i, θ_i, ϕ_i, χ_i, ψ_i, ω_i, ι_i, κ_i, μ_i, ν_i, ξ_i, and ρ_i. The details of the plookup check are ommitted, see Table 4.3 and Equations 4.7 to 4.28 for details. The flags f_f, f_t, and f_s are shared between both plookup instances, as the flags are equal. The symbol ? denotes that the value is not relevant and may take any value $\in \mathbb{F}_p$. l denotes the last step and must be at least $|a| + |t| - 1$. The input vector a starts listing elements at step 1. As boundary constraints, we additionally require $min_0 = y$ and $max_0 = x$.

Table 4.4: Example AET for computing the minimum and maximum via vector lookups using plookup.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Step i	a	min	max	f_l	f_h	t_l	t_h	s_l	s_h	F_{f_l}	F_{f_h}	F_{tl}	F_{th}	G_l	G_h	R_l	R_h	f_f	f_t	f_s
0	?	7	1	?	?	η_0	θ_0	ϕ_0	χ_0	1	1	1	1	1	1	?	?	1	1	1
1	5	5	5	δ_1	ζ_1	η_1	θ_1	ϕ_1	χ_1	ψ_1	ω_1	ι_1	κ_1	μ_1	ν_1	ξ_1	ρ_1	1	1	1
2	6	5	6	δ_2	ζ_2	η_2	θ_2	ϕ_2	χ_2	ψ_2	ω_2	ι_2	κ_2	μ_2	ν_2	ξ_2	ρ_2	1	1	1
3	2	2	6	δ_3	ζ_3	η_3	θ_3	ϕ_3	χ_3	ψ_3	ω_3	ι_3	κ_3	μ_3	ν_3	ξ_3	ρ_3	1	1	1
4	3	2	6	δ_4	ζ_4	η_4	θ_4	ϕ_4	χ_4	ψ_4	ω_4	ι_4	κ_4	μ_4	ν_4	ξ_4	ρ_4	1	1	1
5	2	2	6	δ_5	ζ_5	η_5	θ_5	ϕ_5	χ_5	ψ_5	ω_5	ι_5	κ_5	μ_5	ν_5	ξ_5	ρ_5	1	1	1
6	?	?	?	?	?	η_6	θ_6	ϕ_6	χ_6	ψ_5	ω_5	ι_6	κ_6	μ_6	ν_6	ξ_6	ρ_6	0	1	1
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
j=$\lvert t\rvert - 1$	?	?	?	?	?	η_j	θ_j	ϕ_j	χ_j	ψ_5	ω_5	ι_j	κ_j	μ_j	ν_j	ξ_j	ρ_j	0	1	1
$\lvert t\rvert$	?	?	?	?	?	?	?	$\phi_{\lvert t\rvert}$	$\chi_{\lvert t\rvert}$	ψ_5	ω_5	ι_j	κ_j	$\mu_{\lvert t\rvert}$	$\nu_{\lvert t\rvert}$	$\xi_{\lvert t\rvert}$	$\rho_{\lvert t\rvert}$	0	0	1
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
k=$\lvert s\rvert - 1$	?	?	?	?	?	?	?	ϕ_k	χ_k	ψ_5	ω_5	ι_j	κ_j	μ_k	ν_k	ξ_k=**0**	ρ_k=**0**	0	0	1
$\lvert s\rvert$=$\lvert a\rvert$+$\lvert t\rvert$	?	?	?	?	?	?	?	?	?	ψ_5	ω_5	ι_j	κ_j	μ_k	ν_k	ξ_k=0	ρ_k=0	0	0	0
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
l	?	?	?	?	?	?	?	?	?	ψ_5	ω_5	ι_j	κ_j	μ_k	ν_k	ξ_k=0	ρ_k=0	0	0	0

The constraints for t (see Equations 4.17, 4.25, 4.26, and 4.27) cannot be applied to this STARK, since the values for t are defined by Equations 4.29 and 4.30, leading to a different sequence than t in Section 4.3.4. Coming up with constraints for t in the present setting is more expensive compared to the solution in Section 4.3.4. We do not provide a detailed STARK to fix the constraints for t, but we describe a possible solution:

Uniquely define each t by describing how to compute t (Equations 4.29 and 4.30) and then sort the sequence ascending. Each t can now be reconstructed with provided x, y, α, and Equations 4.29 and 4.30. The prover computes a hash of t inside the STARK and uses the hash as a public output of the STARK. The verifier never sees the actual values of t in the STARK but can build t on his own. The hash of the self-built t can now be compared to the public output hash of the STARK. Section 5.2 demonstrates how to hash an input inside a STARK.

4.3.6 Minimum and Maximum via Distance Checks

With this approach, we utilize the problem of having two distances between two finite field elements in our favor (see Section 4.3). Using the plookup approach from Section 4.3.4, we force all involved elements to be $\in [0, v]$ for a finite field of size greater than $2 \cdot v$ elements.

The construction consists of two separate accumulation schemes. One scheme accumulates the minimum, and the other the maximum. To distinguish between the involved variables, we denote variables belonging to the maximum computation with the subscript h and for the minimum with l, where l and h stand for low and high, respectively. We use low and high as synonyms for min and max since low and high result in a short notation in the equations. We extend the STARK from Section 4.3.4 by seven columns (we do not use column f):

- a is the input vector as in Section 4.3.5.
- $f_l, f_h \in \{1, 0\}$. Comparison flags. Each flag defines for two elements which is the bigger one.
- $min, max \in \mathbb{F}_p$. The running minimum and maximum of a as in Section 4.3.5.
- $\omega_l, \omega_h \in \mathbb{F}_p$. The smaller distance between two finite field elements.

The transition constraints listed in Equations 4.33 to 4.38 ensure correctness, alongside with plookup checks for ω_l, ω_h , and a being $\in [0, v]$. All three plookup checks use the same table, and we can merge f and s together. Furthermore, we need the additional boundary constraints $min_0 = v$ and $max_0 = 0$. Equations 4.35 and 4.36 utilize the helper function $U(f_x)$ defined in Equation 4.12.

$$0 = {f_f}'' \cdot ({f_h}'' \cdot a'' + (1 - {f_h}'') \cdot max' - max'') \tag{4.33}$$

$$0 = {f_f}'' \cdot ((1 - {f_l}'') \cdot a'' + {f_l}'' \cdot min' - min'') \tag{4.34}$$

Table 4.5: Example AET for computing the minimum and maximum using distance checks and plookup.

i	a	min	max	f_l	f_h	ω_l	ω_h	t	s	F_{f_A}	F_{f_B}	F_t	G	R	f_f	f_t	f_s
0	?	6	0	?	?	?	?	0	0	?	1	1	1	?	1	1	1
1	5	5	5	0	1	1	5	1	0	δ_1	ζ_1	η_1	θ_1	ϕ_1	1	1	1
2	6	5	6	1	1	1	1	2	1	δ_2	ζ_2	η_2	θ_2	ϕ_2	1	1	1
3	2	2	6	0	0	3	4	3	1	δ_3	ζ_3	η_3	θ_3	ϕ_3	1	1	1
4	3	2	6	1	0	1	3	4	1	δ_4	ζ_4	η_4	θ_4	ϕ_4	1	1	1
5	2	2	6	0|1	0	0	4	5	1	δ_5	ζ_5	η_5	θ_5	ϕ_5	1	1	1
6	?	?	?	?	?	?	?	6	1	δ_6	ζ_5	η_6	θ_6	ϕ_6	0	1	1
7	?	?	?	?	?	?	?	?	2	δ_7	ζ_5	η_6	θ_7	ϕ_7	0	0	1
8	?	?	?	?	?	?	?	?	2	δ_8	ζ_5	η_6	θ_8	ϕ_8	0	0	1
9	?	?	?	?	?	?	?	?	2	δ_9	ζ_5	η_6	θ_9	ϕ_9	0	0	1
10	?	?	?	?	?	?	?	?	3	δ_{10}	ζ_5	η_6	θ_{10}	ϕ_{10}	0	0	1
11	?	?	?	?	?	?	?	?	3	δ_{11}	ζ_5	η_6	θ_{11}	ϕ_{11}	0	0	1
12	?	?	?	?	?	?	?	?	3	δ_{12}	ζ_5	η_6	θ_{12}	ϕ_{12}	0	0	1
13	?	?	?	?	?	?	?	?	3	δ_{13}	ζ_5	η_6	θ_{13}	ϕ_{13}	0	0	1
14	?	?	?	?	?	?	?	?	4	δ_{14}	ζ_5	η_6	θ_{14}	ϕ_{14}	0	0	1
15	?	?	?	?	?	?	?	?	4	δ_{15}	ζ_5	η_6	θ_{15}	ϕ_{15}	0	0	1
16	?	?	?	?	?	?	?	?	4	δ_{16}	ζ_5	η_6	θ_{16}	ϕ_{16}	0	0	1
17	?	?	?	?	?	?	?	?	5	δ_{17}	ζ_5	η_6	θ_{17}	ϕ_{17}	0	0	1
18	?	?	?	?	?	?	?	?	5	δ_{18}	ζ_5	η_6	θ_{18}	ϕ_{18}	0	0	1
19	?	?	?	?	?	?	?	?	5	δ_{19}	ζ_5	η_6	θ_{19}	ϕ_{19}	0	0	1
20	?	?	?	?	?	?	?	?	6	δ_{20}	ζ_5	η_6	θ_{20}	ϕ_{20}	0	0	1
21	?	?	?	?	?	?	?	?	6	δ_{21}	ζ_5	η_6	θ_{21}	ϕ_{21}	0	0	1
22	?	?	?	?	?	?	?	?	?	δ_{22}	ζ_5	η_6	θ_{21}	ϕ_{21}	0	0	0
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
l	?	?	?	?	?	?	?	?	?	δ_l	ζ_5	η_6	θ_{21}	ϕ_{21}	0	0	0

$$0 = f_f'' \cdot U(f_h'') \tag{4.35}$$

$$0 = f_f'' \cdot U(f_l'') \tag{4.36}$$

$$0 = f_f'' \cdot (f_h'' \cdot (a'' - max') + (1 - f_h'') \cdot (max' - a'') - \omega_h'') \tag{4.37}$$

$$0 = f_f'' \cdot (f_l'' \cdot (a'' - min') + (1 - f_l'') \cdot (min' - a'') - \omega_l'') \tag{4.38}$$

These constraints suffice since making wrong statements about the comparison flag results in a huge ω which does not satisfy $\omega \in \{0, v\}$. The comparison flag determines how we compute the distance between two elements, but we force the distance to be the smaller one using the plookup checks. Therefore, the comparison flag needs to be set appropriately, and we can decide which element is bigger than the other.

$$0 = S((1+\beta)^3 \cdot (\gamma + a'') \cdot (\gamma + \omega_h'') \cdot (\gamma + \omega_l''), f_f'') \cdot F_f' - F_f'' \tag{4.39}$$

Equations 4.33 to 4.38 contain a multiplication with the flag f_f, which is unnecessary, but we do it to pin down the steps this constraint affects (and this flag also improves the readability of Table 4.5 since we can use a lot of ? placeholders). The degree of these constraints increases by one and is 3 (remember that we want to keep the degree of our constraints ≤ 3; see Section 4.3). Column F_f absorbs three values in each step as defined in Equation 4.39, resulting in a degree of 5. We can reduce this degree by splitting up F_f into two columns, which results in the transition constraints described by Equations 4.40 and 4.41, all having a degree ≤ 3. F_{f_B} is equal to the old F_f and to be used instead. Equations 4.39 and 4.41 use the helper function $S(a, f_x)$ defined in Equation 4.10.

$$0 = (1+\beta)^3 \cdot (\gamma + a'') \cdot (\gamma + \omega_h'') \cdot (\gamma + \omega_l'') - F_{f_A}'' \tag{4.40}$$

$$0 = S(F_{f_A}'', f_f'') \cdot F_{f_B}' - F_{f_B}'' \tag{4.41}$$

Finally, Table 4.5 shows an example instance of this STARK with $v = 6$, $t = \{0, 1, 2, 3, 4, 5, 6\}$, and $a = \{5, 6, 2, 3, 2\}$. All transition constraints have a degree of at most 3. β and γ are not explicitly defined and all values depending on β and γ are denoted by δ_i, ζ_i, η_i, θ_i, and ϕ_i. The symbol ? means that the value is not relevant and may take any value $\in \mathbb{F}_p$. l denotes the last step where $l \geq |s| = |t| - 1 + 3 \cdot |a|$.

4.3.7 Median via Distance Checks and Multiset Equality Checks

Similar to the distance checks used in Section 4.3.6 for computing the minimum and maximum, we can build a STARK that computes the median of the input values. This solution requires randomness as we are using two subcomponents that rely on randomness. We assume having access to a source of randomness and Section 4.3.8 describes how to derive randomness in the STARKs setting.

We define a helper column b holding the same values as our input column a but sorted ascending. A multiset check as introduced in Section 4.2.3 enforces the elements in the columns to be equal. Another third helper column ω defines the distances between the next and the current values of the sorted input column b, where the current value is subtracted from the next value. This subtraction fixes one of the two possible distances between finite field elements. A plookup check ensures that the input values in column

Table 4.6: Example AET for computing the median using distance checks and plookup.

Step i	a	b	ω	z	t	s	F_f	F_t	G	R	f_f	f_t	f_s
0	?	0	?	$\mathbf{1}$	0	0	1	1	1	?	1	1	1
1	5	2	2	δ_1	1	0	ζ_1	η_1	θ_1	ϕ_1	1	1	1
2	6	2	0	δ_2	2	1	ζ_2	η_2	θ_2	ϕ_2	1	1	1
3	2	3	1	δ_3	3	1	ζ_3	η_3	θ_3	ϕ_3	1	1	1
4	3	5	2	δ_4	4	1	ζ_4	η_4	θ_4	ϕ_4	1	1	1
5	2	6	1	$\delta_5 = \mathbf{1}$	5	2	ζ_5	η_5	θ_5	ϕ_5	1	1	1
6	?	?	?	?	6	2	ζ_5	η_6	θ_6	ϕ_6	0	1	1
7	?	?	?	?	?	2	ζ_5	η_6	θ_7	ϕ_7	0	0	1
8	?	?	?	?	?	2	ζ_5	η_6	θ_8	ϕ_8	0	0	1
9	?	?	?	?	?	2	ζ_5	η_6	θ_9	ϕ_9	0	0	1
10	?	?	?	?	?	3	ζ_5	η_6	θ_{10}	ϕ_{10}	0	0	1
11	?	?	?	?	?	3	ζ_5	η_6	θ_{11}	ϕ_{11}	0	0	1
12	?	?	?	?	?	4	ζ_5	η_6	θ_{12}	ϕ_{12}	0	0	1
13	?	?	?	?	?	5	ζ_5	η_6	θ_{13}	ϕ_{13}	0	0	1
14	?	?	?	?	?	5	ζ_5	η_6	θ_{14}	ϕ_{14}	0	0	1
15	?	?	?	?	?	6	ζ_5	η_6	θ_{15}	ϕ_{15}	0	0	1
16	?	?	?	?	?	6	ζ_5	η_6	θ_{16}	$\phi_{16} = \mathbf{0}$	0	0	1
17	?	?	?	?	?	?	ζ_5	η_6	θ_{16}	$\phi_{16} = 0$	0	0	0
..	..	..	..	..	..	..	..	..	..	..	..	..	..
l	?	?	?	?	?	δ_5	ζ_5	η_6	θ_{16}	$\phi_{16} = 0$	0	0	0

a and the distances ω are within a given range, as in Section 4.3.6. The multiset s is defined as $a \cup \omega \cup t$.

This construction and constraints result in a column b that holds all input values sorted ascending. The length l of the input sequence is public information. Therefore, we can take the element in column b approximately around step $l/2$ and use it as the median.

The median is defined as $\frac{b_{(l/2)-1}+b_{l/2}}{2}$ for an even l and as $b_{(l-1)/2}$ for l being odd [86]. Divisions might lead to a fractional result. Therefore, we define two additional slots in the public inputs that hold two elements describing the median. The resulting median is the addition of both elements divided by two. If l is odd, we place the same value in both slots of the public inputs.

Table 4.6 shows an example AET for computing the median with $t = \{0, 1, 2, 3, 4, 5, 6\}$ and $a = \{5, 6, 2, 3, 2\}$. β, γ, and λ are not explicitly defined and all values depending on β, γ, and λ are denoted by δ_i, ζ_i, η_i, θ_i, and ϕ_i. The symbol ? means that the value is not relevant and may take any value $\in \mathbb{F}_p$. l denotes the last step where $l \geq |s| = |t| + |a| - 1$.

The transition and boundary constraints are the same as for the two subcomponents, plookup (see Section 4.3.4), and multiset equality checks (see Section 4.2.3), applied to the present setting. However, slight modifications are required: Equations 4.43 and 4.44 list the modified transition constraints, and Equation 4.45 provides the adapted boundary constraint. The new constraints for the check of b being monotonically rising are listed in Equations 4.42 and 4.46. Equation 4.44 uses the helper function $S(a, f_x)$ defined in Equation 4.10.

$$0 = {f_f}'' \cdot (b'' - b' - \omega'') \tag{4.42}$$

$$0 = {f_f}'' \cdot \left(z' \cdot \frac{a'' + \lambda}{b'' + \lambda} - z'' \right) \tag{4.43}$$

$$0 = S((1 + \beta)^2 \cdot (\gamma + a'') \cdot (\gamma + \omega''), {f_f}'') \cdot {F_f}' - {F_f}'' \tag{4.44}$$

$$1 = z_0 = z_l \tag{4.45}$$

$$0 = b_0 \tag{4.46}$$

4.3.8 Deriving Pseudo - Randomness

Many constructions we built in this section require randomness. STARKs are non-interactive protocols and use the Fiat-Shamir Heuristic [26], as described in Section 2.4.1. As users of the STARK protocol, we may also utilize this heuristic to derive randomness for our checks.

The main message of Fiat-Shamir [26] (see Section 2.3.4) is that randomness needs to be derived from a pseudo-random function that takes all previously known information as input. The function's output must be unpredictable based on the information known beforehand.

Let us apply Fiat-Shamir to the multiset equality check introduced in Section 4.2.3. The two sets $\{a_i\}_{i \in [n]}$ and $\{b_i\}_{i \in [n]}$ (hereafter referred to as a and b) are considered multiset equal if the polynomials $A(\lambda)$ and $B(\lambda)$ (see Equations 4.47 and 4.48) built from the sets are equal. Using Schwartz-Zippel (see Section 2.2), we evaluate both polynomials at random λ and check if the results are equal. Fiat-Shamir requires the randomness to be computed from a and b since these sets define the polynomials $A(\lambda)$ and $B(\lambda)$. We use a cryptographic hash function to derive a pseudo-random value from the sets a and b according to the theory of random oracle models [8].

Since we might want to hide the sets a and b as we are in a STARK environment, we need to evaluate the hash function inside the AET to guarantee the correctness of applying Fiat-Shamir in this setting. Section 4.3.1 describes how to evaluate a hash function inside

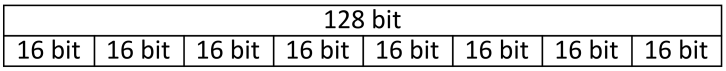

Figure 4.4: Fitting multiple pixels into one field element. 16 bits describe one pixel. We can fit 8 pixels into one 128-bit word.

a STARK. Unfortunately, deriving randomness with this solution introduces significant overhead to the present multiset equality check.

Furthermore, we require randomness in two more areas inside plookup. First, recall the vector lookup technique presented in Section 2.5.3 where a vector of field elements gets compressed into one field element, where the randomness of α needs to be dependent on f and t.

Second, in the general setting of plookup, we evaluate the polynomials F and G (see Section 2.5 and Equations 2.9 and 2.10) at random β and γ. The polynomials are defined over f, t, and s, so the randomness of β and γ needs to be dependent on f, t, and s. Deriving randomness in both cases is done via evaluating a hash function on the dependent-on values within the STARK, causing a significant overhead.

$$A(\lambda) = \prod_{i \in [n]} (a_i + \lambda) \tag{4.47}$$

$$B(\lambda) = \prod_{i \in [n]} (b_i + \lambda) \tag{4.48}$$

However, if we are using a t known to the public due to constraints, we can consider t to be constant and do not need to include t in the hash evaluation for computing pseudo-randomness. This is the case for the example STARK provided in Section 4.3.4.

4.3.9 Fitting multiple Pixels into one Field Element

As mentioned in Section 4.3, we use a list of pixels as input, where one pixel is a 16-bit value. The prime fields available in Winterfell use moduli of 62, 64, and 128 bits, which allows us to fit multiple pixels inside one field element without losing precision. Figure 4.4 demonstrates this idea.

Since we operate on a finite field, we cannot use the whole 128-bit space as shown in Figure 4.4. We need to ensure that the largest possible constructed e is smaller than our field modulus. For the 128-bit field with modulus $2^{128} - 45 \cdot 2^{40} + 1$ each 16-bit value must be within $[0, 2^{16} - 2]$, which results in the largest possible value of e being 0xFFFE..FFFE, which is smaller than our field modulus (0xFFFFF..).

Our example use case operates on the range of approximately 500 to 2000 °C, and the 16-bit representation thereof is the range $[5000, 20000]$. The largest value (20000) is smaller than the highest value we allow, being $2^{16} - 2 = 65534$. Therefore, this range limitation fits our example use case.

This construction allows us to save space in the state of our hash functions. We absorb multiple pixels into each field element of the hash state in the absorption phase, reducing the overall number of absorption phases compared to absorbing one pixel into one field element. This approach reduces the length of the execution trace by the factor of pixels per element.

Let us use the following terminology:

- Uncompressed pixels: One field element holds one pixel.
- Compressed pixels: One field element holds multiple pixels, as shown in Figure 4.4. Note that the compression is lossless.

As we want to compute statistics on the input, we cannot use compressed pixels as input to our STARK. If we use compressed pixels as our input, we need to decompose the field element into multiple values, which involves bitwise operations (e.g. masking using bitwise *and*) to compute statistics on the individual pixels. We already use arithmetization friendly hash functions to avoid bitwise operations, so this approach is not meaningful.

However, we can use uncompressed pixels as input and compress them within the STARK during the absorption phase. Compressing can be done via multiplication (which simulates bit-shifting), therefore being arithmetization friendly. Equations 4.49 to 4.51 demonstrate the compression of four 16-bit values a, b, c, d into one 64-bit value e, where $\cdot << b$ denotes left shift by b bits and is equal to $\cdot 2^b$. This technique is used in the construction of STARK C (see Section 5.2.1), but this construction has security flaws.

$$e = (a << 0) + (b << 16) + (c << 32) + (d << 48) \tag{4.49}$$

$$e = (2^{16})^0 a + (2^{16})^1 b + (2^{16})^2 c + (2^{16})^3 d \tag{4.50}$$

$$e = a + 2^{16} b + 2^{32} c + 2^{48} d \tag{4.51}$$

The compression is only sound when the values are in the range we expect (16-bit, or better said $[0, 2^{16} - 2]$ to make sure e is smaller than the modulus as discussed before). Inside the AET, we compress multiple field elements into one field element of the same size. The prover may come up with values outside the desired range and construct different pixels having the same compressed version (compromising works via finding multiple solutions a, b, c, d for fixed e to Equation 4.51; note that a, b, c, d, e are finite field elements). To avoid this, we need to ensure that our input pixels are within a specific range. plookup (see Section 4.3.4) helps us achieve this, but plookup requires randomness of all involved values, including the uncompressed input pixels. We can derive randomness by hashing the uncompressed input pixels, which we planned to avoid, making this approach meaningless. STARK D (see Section 5.2.1) extends STARK C by the plookup check but omits deriving randomness.

In conclusion, we cannot improve our STARK by compressing pixels into field elements since each approach makes the STARK more complex and reduces efficiency.

Chapter 5

Implementation

The ideas presented in the previous chapters now get crafted into various performant implementations. All files are available at GitHub[1] and files are referenced by file paths using the pattern `/folder/subfolder/source.code`.

5.1 Hash Functions

For the sake of comparison, we use two hash functions in our construction of the STARK:

1. Rescue-Prime
2. Griffin

Rescue-Prime was already introduced in Section 4.3.1, one of the best options available to date [11]. Griffin is also an arithmetization friendly hash function being developed by a team of researchers at the Institute of Applied Information Processing and Communications (IAIK)[2] at TU Graz when writing the present thesis.

Meanwhile, Griffin was released to the public [39]. However, during working on the thesis, Griffin was not public yet, and we used an older and slightly different version than the official one. As Giffin was not public yet, we hide all implementation details of Griffin and only mention that Rescue-Prime is easily interchangeable with Griffin in all constructions.

The library Winterfell provides an example implementation for Rescue-Prime [21], which we use as a base for our implementation. The parameters for Winterfell's Rescue-Prime example are defined using constants in Rust. Since we want to experiment with various parameter choices, we decided to provide a generator script that creates Rust source code for one specific instance of hash functions.

This decision is based on keeping the implementation as simple as possible since we seek a performant solution. We could also have implemented the hash functions in a parametrized and generic way (recall that prime fields in Winterfell are defined via Traits), including the computation of the matrices and constants. We decided against a polished hash implementation since the focus lies on the performance of the STARK and not on creating a user-friendly interface for the hash functions.

[1] `https://github.com/romanmarkusholler/MasterThesis`
[2] `https://www.iaik.tugraz.at/`

5.1.1 Deriving Parameters

We first implement Rescue-Prime by adapting the Rust implementation of the Winterfell example [21]. The code was generalized and uses constants for all parameters. The result is the file `/code/rust/src/rescue/template.rs` which holds the algorithms to compute the hash function while omitting the definition of constants for the parameters. The Linux shell script `/code/sage/derive_rescue.sh` takes four arguments and creates a Rust source file based on the template, including all derived constants for this instance of Rescue-Prime. The arguments for the script are:

- Prime for the definition of the prime field.
- State size (number of elements).
- Capacity (number of elements).
- Security level in bits.

The shell script uses an underlying Sagemath script to compute the MDS matrix and all constants, taken and adapted from [25] which implements the algorithms presented in the Rescue-Prime specification [75]. Various configurations derived by this script are located inside `/code/rust/src/rescue/` where the filename pattern is `p<prime>_m<state size>_c<capacity>_s<security level>.rs`.

Similar to the structure for defining instances for Rescue-Prime, we created another Linux shell script to compute the parameters for Griffin. However, due to Griffin not being published yet, we omit all implementation details, except that Griffin is easily exchangeable with Rescue-Prime in terms of using the hash functions when constructing a STARK.

Meanwhile, Griffin was released to the public [39]. However, the hash function changed, and we use an older version. Therefore we do not provide the details of the old Griffin.

5.1.2 Interfaces

The goal is to make both hash functions easily exchangeable in a STARK construction. Therefore, the interfaces provided to the programmer are as similar as possible.

Both hash functions expose the function defined in Listing 5.1 to retrieve the round constants for the periodic columns. `cycle_length` is the size of one complete periodic column cycle within Winterfell which must be a power of two. `shift` defines an offset when the first round constant is used within a periodic cycle and `Elem` is the datatype of the current prime field.

Listing 5.1: Interface for retrieving round constants

```
1 pub fn get_round_constants_periodic(
2     cycle_length: usize,
3     shift: usize
4 ) -> Vec<Vec<Elem>> { .. }
```

Listing 5.2 shows the signature of the function that takes the `state` of the hash function and applies round number `round` to the mutable state. This function is used for constructing the AET and for helper functions that compute the hash outside a STARK setting.

Listing 5.2: Interface for computing one round of the hash function

```
pub fn apply_round(
    state: &mut [Elem],
    round: usize
) { .. }
```

A STARK consists of public inputs, an execution trace, boundary constraints, and transition constraints, where three different transition constraints are required when computing hash functions:

- Absorbing pixels into the state during the absorption phase.
- Computing one round at a time of the hash function.
- Copying the current state to the next state when the number of rounds does not match the size of the periodic column cycle.

Enforcing constraints on one round of the hash function is specific to the hash function used. Therefore each implementation of hash functions exposes a function with the signature shown in Listing 5.3 to ensure correct computation of one round, which the Winterfell prover calls while constructing the proof.

Listing 5.3: Interface for enforcing transition constraints

```
pub fn enforce_round<E: FieldElement + From<Elem>>(
    result_slice: &mut [E],
    current_slice: &[E],
    next_slice: &[E],
    round_constants: &[E],
    flag: E,
) { .. }
```

The type `FieldElement` is the Trait defining the behavior of the finite field in use. The result of evaluating constraints gets written to the mutable `result_slice`. `current_slice` and `next_slice` hold the current and next state of the hash state, respectively. `round_constants` holds the round constants for the current round to be enforced, originating from the periodic columns. `flag` is either 0 or 1, and constraints get enforced when `flag` is 1.

Moreover, the implementations expose the following helpful constants for using the hash functions:

- `STATE_WIDTH: usize`
- `RATE: usize`
- `NUM_ROUNDS: usize`

5.2 The STARKs

The ideas presented in Section 4.3 now get assembled into some STARK variations. Together with the hash function implementations provided in Section 5.1, we construct various STARKs step by step in a comprehensible fashion.

Throughout this thesis, we use capital letters to enumerate different STARK constructions for simple identification of approaches. We define a STARK and iteratively extend the STARK with new components. The goal is modularization to ease understanding the presented content.

All presented STARKs use variants of Rescue-Prime, as the public knows the hash function. Rescue-Prime may be interchanged with Griffin in every STARK without significant modifications. Recall the interface definitions for the hash functions presented in Section 5.1.2.

Section 5.2.1 describes STARKs that prove knowledge to the preimage of a public hash value. This first step shows how we approach feeding massive datasets into the proof system. Computing statistics alongside the hash value over all pixels extends the first step, described in Section 5.2.2. Moreover, Section 5.2.3 extends computing statistics by only using a specific subset of the pixels using an ROI definition. Finally, Section 5.2.4 aims to optimize space consumption in the AET to improve the overall performance of the STARKs.

5.2.1 Hashing of Input Data

This solution feeds the video into the execution trace and computes the hash value inside the STARK.

STARK A. We start with a basic STARK to demonstrate the general idea to the reader. The utilized variant of Rescue-Prime uses the 128-bit field from the Winterfell library and the implementation is to be found at `/code/rust/src/rescue/p128_m4_c3_s128.rs`. The parameters are a state size of 4, the capacity is 3, the number of rounds is 15, and the minimum security of the hash function is 128 bits. The 128-bit prime field provides security of around 100 bits. The implementation of STARK A is located at `/code/rust/src/stark/stark_a.rs`. Table 5.3 presents the AET of this solution, using example pixel values as input. We omit the values of the hash state and denote them with δ_i, ζ_i, η_i, and θ_i.

Transition constraints and periodic columns. As mentioned in Section 4.2.2, state transition constraints get evaluated between two adjacent states, called the `current` and the `next` state. The first constraints being evaluated operate on the states from steps 0 and 1. Periodic values get used for each constraint evaluation, so the periodic values with index 0 get used with the states from steps 0 and 1. Table 5.1 explains the purpose of the periodic columns for the current example. The type `flag` means that the values are $\in \{0, 1\}$, whereas the type `element` may take any value $e \in \mathbb{F}_p$. Table 5.2 assigns values to the periodic columns.

Description of the AET in Table 5.3. Step i is the index in the AET and describes the values of our variables at a certain point in time i. The state of the AET is defined

Table 5.1: Periodic column definitions for STARK A.

Name	Index	Type	Description
Hash Mask p_H	0	flag	If 1, one round of the permutation gets enforced. Utilizes the round constants p_{C_x}, which are also provided as periodic values.
Statistics Mask p_S	1	flag	If 1, performs absorbing elements into the hash state (absorption phase). Called Statistics Mask because statistics will be computed alongside with absorbing elements in later STARK constructions.
Round Constants p_{C_x}	2 to 9	element	8 round constants ($x \in [0, 7]$) for computing one round of the Rescue-XLIX permutation.

by the variables P and H_x for $x \in [0, 3]$. P holds all pixels from the video, where pixels get placed in the AET at each step $i = 16 \cdot k + 1$, where $k \in [0, j - 1]$ and j is the number of pixels in the video. The H_x hold the state of the Rescue-Prime hash function. At each step $i = 16 \cdot k$ the H_x hold the state before/after a full Rescue-XLIX permutation. For $i = 16 \cdot k + 1$, the Rescue-Prime state has already absorbed the next pixel of the video. The following steps $i = 16 \cdot k + l$ where $l \in [2, 16]$ are involved in performing one round of the Rescue-XLIX permutation each, where the state at step i is the result of applying one round of the Rescue-XLIX permutation to the state $i - 1$ as input.

Periodic columns help us to formulate transition constraints on the AET. For the first state transition between steps 0 and 1, the statistics flag is 1, and all other flags are 0. Therefore, we absorb the pixel value into the state of Rescue-Prime by adding the `next` pixel value to the `current` state of Rescue-Prime, which must be equal to the `next` state of Rescue-Prime.

Table 5.2: Value assignment for the periodic columns for STARK A.

Index	0 p_H	1 p_S	2 p_{C_0}	3 p_{C_1}	.. ..	9 p_{C_7}
0	0	1	e	e	..	e
1..15	1	0	e	e	..	e

Table 5.3: Algebraic Execution Trace for STARK A.

Index	0	1	2	3	4
Step i	P	H_0	H_1	H_2	H_3
0	?	$\delta_0 = 0$	$\zeta_0 = 0$	$\eta_0 = 0$	$\theta_0 = 0$
1	1054	$\delta_1 = \delta_0 + 1054$	$\zeta_1 = \zeta_0$	$\eta_1 = \eta_0$	$\theta_1 = \theta_0$
2	?	δ_2	ζ_2	η_2	θ_2
..	..	..	..	..	..
16	?	δ_{16}	ζ_{16}	η_{16}	θ_{16}
17	1052	$\delta_{17} = \delta_{16} + 1052$	$\zeta_{17} = \zeta_{16}$	$\eta_{17} = \eta_{16}$	$\theta_{17} = \theta_{16}$
..	..	..	..	..	..
$u = 16(j-1)$	?	δ_u	ζ_u	η_u	θ_u
$u + 1$	1089	$\delta_{u+1} = \delta_u + 1089$	$\zeta_{u+1} = \zeta_u$	$\eta_{u+1} = \eta_u$	$\theta_{u+1} = \theta_u$
..	..	..	..	..	..
$v = 16j$	?	δ_v	ζ_v	η_v	θ_v
$v + 1$	?	$\delta_{v+1} = \delta_v + ?$	$\zeta_{v+1} = \zeta_v$	$\eta_{v+1} = \eta_v$	$\theta_{v+1} = \theta_v$
..	..	..	..	..	..
b	..	..	..	..	..

The following 15 transitions (involving steps 1 to 16) make sure that the round function gets correctly applied to the state of the Rescue-Prime hash function. The hash flag is 1, and all other flags are 0. We do not care about the value of the pixel variable (so we denote this by ?). One round of the Rescue-XLIX permutation gets enforced on the Rescue-Prime state, where `next` is the result of applying the Rescue-XLIX permutation to `current` using the round constants provided by the periodic columns. Listing 5.4 shows the implementation of transition constraints for one round of the Rescue-XLIX permutation (recall the MITM approach mentioned in Section 4.3.1 using Figure 4.2 as supportive material). Rescue-Prime parameters influence the present function via slice lengths of the function parameters (`result_slice`, ..) and constants (`MDS`, ..). A description of the interface is provided in Section 5.1.2 and Listing 5.3. Line 32 (`result_slice[i] += flag * (step2[i] - step1[i]);`) defines the i transition constraints where `result_slice[i]` evaluates to zero for each i when the AET was appropriately constructed.

Listing 5.4: Implementation of enforcing transition constraints for one round the Rescue-XLIX permutation of Rescue-Prime.

```
pub fn enforce_round<E: FieldElement + From<Elem>>(
    result_slice: &mut [E],
    current_slice: &[E],
    next_slice: &[E],
    round_constants: &[E],
    flag: E,
) {
    // compute the state that should result from applying the
    // first half of Rescue round to the current state of the
    // computation
    let mut step1 = [E::ZERO; STATE_WIDTH];
    step1.copy_from_slice(current_slice);
    apply_sbox(&mut step1);
    matrix_mul(MDS, &mut step1);
    for i in 0..STATE_WIDTH {
        step1[i] += round_constants[i];
    }

    // compute the state that should result from applying the
    // inverse for the second half for Rescue round to the next
    // step of the computation
    let mut step2 = [E::ZERO; STATE_WIDTH];
    step2.copy_from_slice(next_slice);
    for i in 0..STATE_WIDTH {
        step2[i] -= round_constants[STATE_WIDTH + i];
    }
    matrix_mul(INV_MDS, &mut step2);
    apply_sbox(&mut step2);

    // make sure that the results are equal
    for i in 0..STATE_WIDTH {
        result_slice[i] += flag * (step2[i] - step1[i]);
    }
}
```

As presented in Section 4.2, we use var' to denote the `current` state of the variable var, and var'' to denote the `next` state. This notation is used to describe transition constraints. Periodic columns always get referenced by the variable name (e.g. p_H), as there is exactly one value available when evaluating a transition constraint. In total, STARK A requires eight transition constraints:

- Four transition constraints for the absorption phase, defined in Equations 5.1 to 5.4.
- Four transition constraints for enforcing one round of the computation of one Rescue-XLIX permutation, as defined in Listing 5.4.

$$0 = p_S(H_0{}' + P'' - H_0{}'') \tag{5.1}$$

$$0 = p_S(H_1{}' - H_1{}'') \tag{5.2}$$

$$0 = p_S(H_2' - H_2'') \tag{5.3}$$

$$0 = p_S(H_3' - H_3'') \tag{5.4}$$

Public Inputs. The length j of the pixel sequence, as well as the hash of the pixels, is public information, available to the prover and the verifier. We denote the public inputs with g, where g stands for global. Similar to Rust, we define members of public inputs via the dot membership selection operator. The public inputs are $g.j$ and $g.H$, where $g.H$ is the hash of the input pixels ($g.H$ is an array indexed with x by $g.H[x]$ consists out of r_p elements for Rescue-Prime).

Boundary Constraints. At step 0, the initial step, all H_x for $x \in [0, 3]$ are set to 0, which is mandatory for the variables of the Rescue-Prime state as the initial value comprises zeroes only. There are no boundary constraints for the pixel values, as there exist no constraints between them in any sense. We do not need any boundary constraints throughout the computation since the transition constraints ensure correctness. However, the final value of the hash computation needs to be bound to the publicly known hash, where the result resides at step $16j$. The AET probably does not end at step $16j$, since the length of the AET must be a power of two for the Winterfell library. Step b in the AET shown in Table 5.3 describes the last step of the AET, where the length of the AET $b + 1$ is the next closest power of two $\geq 16j$. The constraints for hashing the input get also enforced after step $16j$, but this does not matter, as we are interested in the result at step $16j$. Equations 5.5 to 5.6 define the boundary constraints. Note that we are able to check the hash state at step $16j$ because j is available as a public input. As presented in Section 4.2, we use var_i to denote the value of the variable var at step i.

$$0 = H_{00} = H_{10} = H_{20} = H_{30} \tag{5.5}$$

$$g.H[0] = H_{016j} \tag{5.6}$$

Padding. Hash functions, such as Rescue-Prime, require padding of the input. This can be achieved using the domain separation approach presented in Section 4.2.3, where different transition constraints get enabled after all pixels have been absorbed. We omit this detail in the present thesis and do not pad any of the implementations presented.

With STARK A, we introduced the principle of feeding the video into the execution trace. Now we focus on reducing space consumption in the AET, which will increase the solution's performance. The main approaches are:

1. Increasing the rate r_p and reducing the capacity c_p allows absorbing more pixels at once.

2. The state size m may be increased to absorb more pixels at once.

3. One finite field element comprises 62, 64, or 128 bits in Winterfell. We can fit multiple pixels into one field element of the hash state.

Table 5.4: Algebraic Execution Trace for STARK B.

Index	0	1	2	3	4	5
Step i	P_0	P_1	H_0	H_1	H_2	H_3
0	?	?	$\delta_0 = 0$	$\zeta_0 = 0$	$\eta_0 = 0$	$\theta_0 = 0$
1	54	65	$\delta_1 = \delta_0 + 54$	$\zeta_1 = \zeta_0 + 65$	$\eta_1 = \eta_0$	$\theta_1 = \theta_0$
..	..	..	..	..	..	..
15	?	?	δ_{15}	ζ_{15}	η_{15}	θ_{15}
16	?	?	$\delta_{16} = \delta_{15}$	$\zeta_{16} = \zeta_{15}$	$\eta_{16} = \eta_{15}$	$\theta_{16} = \theta_{15}$
17	32	34	$\delta_{17} = \delta_{16} + 32$	$\zeta_{17} = \zeta_{16} + 34$	$\eta_{17} = \eta_{16}$	$\theta_{17} = \theta_{16}$
..	..	..	..	..	..	..
$u-1$	?	?	δ_{u-1}	ζ_{u-1}	η_{u-1}	θ_{u-1}
$u = 16(\frac{j}{2} - 1)$	?	?	$\delta_u = \delta_{u-1}$	$\zeta_u = \zeta_{u-1}$	$\eta_u = \eta_{u-1}$	$\theta_u = \theta_{u-1}$
$u+1$	84	86	$\delta_{u+1} = \delta_u + 84$	$\zeta_{u+1} = \zeta_u + 86$	$\eta_{u+1} = \eta_u$	$\theta_{u+1} = \theta_u$
..	..	..	..	..	..	..
$v-1$	?	?	δ_{v-1}	ζ_{v-1}	η_{v-1}	θ_{v-1}
$v = 16\frac{j}{2}$	?	?	$\delta_v = \delta_{v-1}$	$\zeta_v = \zeta_{v-1}$	$\eta_v = \eta_{v-1}$	$\theta_v = \theta_{v-1}$
$v+1$	?	?	$\delta_{v+1} = \delta_v + ?$	$\zeta_{v+1} = \zeta_v + ?$	$\eta_{v+1} = \eta_v$	$\theta_{v+1} = \theta_v$
..	..	..	..	..	..	..
b	..	..	..	..	..	..

Absorbing multiple pixels in one phase means that we need to provide multiple variables for holding the pixels. This approach makes the execution trace wider, but there are savings on the length of the trace. There are various approaches for comparing the performance of STARKs, one of them being $w \cdot T \cdot d_{max}$ where smaller numbers are better (as used in [2]; w is the width of the AET, T is the length of the AET, and d_{max} is the highest degree of all transition constraints). For example, absorbing two elements at once ($r_p = c_p = 2$) makes the AET wider by one element but reduces the trace length by factor two in case of modifying STARK A, therefore shrinking $w \cdot T \cdot d_{max}$.

STARK B. We extend STARK A by absorbing multiple pixels at once into our hash state by increasing the rate of the hash function. The parameters for Rescue-Prime remain the same as for STARK A, except for $r_p = 2$ and $c_p = 2$. Modifying the rate influences other parameters as well (see Section 5.1). For STARK B, we use the Rescue-Prime implementation found at `/code/rust/src/rescue/p128_m4_c2_s128.rs` and the parameters are a state size of 4, the capacity is 2, the number of rounds is 14, and the minimum security of the hash function is 128 bits. The implementation of STARK B

Table 5.5: Periodic column definitions for STARK B.

Name	Index	Type	Description
Hash Mask p_H	0	flag	If 1, one round of the permutation gets enforced. Utilizes the round constants p_{C_x}, which are also provided as periodic values.
Copy Mask p_{cp}	1	flag	If 1, the state of the Rescue-Prime hash function gets copied to the next state.
Statistics Mask p_S	2	flag	If 1, performs absorbing elements into the hash state (absorption phase). Called Statistics Mask because statistics will be computed alongside with absorbing elements in later STARK constructions.
Round Constants p_{C_x}	3 to 10	element	8 round constants ($x \in [0, 7]$) for computing one round of the Rescue-XLIX permutation.

is located at `/code/rust/src/stark/stark_b.rs`. Note that the number of rounds changed from 15 to 14 for STARK B. The AET grows by one variable and now has the width $w = 6$. Table 5.4 shows the AET for STARK B, using example pixel values as input. We omit the values of the hash state and denote them with δ_i, ζ_i, η_i, and θ_i.

For this construction, we assume the length of the input vector P to be a multiple of 2.

The switch from 15 to 14 rounds forces us to perform additional constructions for STARK B. We need to create a copy constraint between steps $16k + 15$ and $16k + 16$ for $k \in [0, (j/2) - 1]$ as we compute one round less of the Rescue-XLIX permutation compared to STARK A. We introduce a copy flag that enforces a copy constraint at the desired steps. The modified periodic column definitions for STARK B are listed in Table 5.5, and the assignment of values to those periodic columns is shown in Table 5.6.

The public inputs are the same as for STARK A ($g.j$, $g.H_0$, $g.H_1$, $g.H_2$, and $g.H_3$). The boundary constraints have slightly changed and are defined in Equations 5.7 to 5.9 for STARK B. The transition constraints also have changed and are defined in Equations 5.10 to 5.17 and Listing 5.4. The length $b + 1$ of the AET is now shorter, and b is the next closest power of two $\geq 16\frac{j}{2}$.

$$0 = H_{00} = H_{10} = H_{20} = H_{30} \tag{5.7}$$

$$g.H[0] = H_{0_{16\frac{j}{2}}} \tag{5.8}$$

$$g.H[1] = H_{1_{16\frac{j}{2}}} \tag{5.9}$$

$$0 = p_S(H_0' + P_0'' - H_0'') \tag{5.10}$$

$$0 = p_S(H_1' + P_1'' - H_1'') \tag{5.11}$$

$$0 = p_S(H_2' - H_2'') \tag{5.12}$$

$$0 = p_S(H_3' - H_3'') \tag{5.13}$$

$$0 = p_{cp}(H_0' - H_0'') \tag{5.14}$$

$$0 = p_{cp}(H_1' - H_1'') \tag{5.15}$$

$$0 = p_{cp}(H_2' - H_2'') \tag{5.16}$$

$$0 = p_{cp}(H_3' - H_3'') \tag{5.17}$$

By increasing the rate r_p to two instead of one, the AET length of STARK B has halved compared to STARK A. The width of the AET w gets increased by one. We can increase the state size and rate of the hash function further to gain more savings on the length of our AET.

STARK C. To better utilize available space in our hash state, we extend STARK B by absorbing multiple pixels into one field element of our hash state, as described in Section 4.3.9. We use the same implementation of Rescue-Prime as for STARK B (`/code/rust/src/rescue/p128_m4_c2_s128.rs`), and the implementation for STARK C is located at `/code/rust/src/stark/stark_c.rs`.

For this construction, we assume the length of the input vector P to be a multiple of 16.

Table 5.6: Value assignment for the periodic columns for STARK B.

Index	0	1	2	3	4	..	10
	p_H	p_{cp}	p_S	p_{C_0}	p_{C_1}	..	p_{C_7}
0	0	0	1	e	e	..	e
1..14	1	0	0	e	e	..	e
15	0	1	0	e	e	..	e

We first construct the AET for compressing pixels into the hash state only, and we add the plookup check later in the implementation of STARK D, which extends STARK C. In STARK C, we use 16 columns for providing the input pixels, where the first 8 pixels get absorbed into H_0 and the second eight pixels get absorbed into H_1. The periodic column definitions are the same as for STARK B (see Tables 5.5 and 5.6). Table 5.7 shows the AET of STARK C, using example pixel values as input. We omit the values of the hash state and denote them with δ_i, ζ_i, η_i, and θ_i. The boundary constraints are defined in Equations 5.18 to 5.20, and the transition constraints in Equations 5.21 to 5.28. The state width of the AET is 20, and the length of the AET is $b + 1$ where b is the next closest power of two $\geq j$.

Table 5.7: Algebraic Execution Trace for STARK C.

Index	0 .. 15	16	17	18	19
Step i	P_0 .. P_{15}	H_0	H_1	H_2	H_3
0	..	$\delta_0 = 0$	$\zeta_0 = 0$	$\eta_0 = 0$	$\theta_0 = 0$
1	..	$\delta_1 = \delta_0 + \sum_{x=0}^{7} 2^{16x} P_{x1}$	$\zeta_1 = \zeta_0 + \sum_{x=8}^{15} 2^{16x} P_{x1}$	$\eta_1 = \eta_0$	$\theta_1 = \theta_0$
..	..	..	..	..	..
15	..	δ_{15}	ζ_{15}	η_{15}	θ_{15}
16	..	$\delta_{16} = \delta_{15}$	$\zeta_{16} = \zeta_{15}$	$\eta_{16} = \eta_{15}$	$\theta_{16} = \theta_{15}$
17	..	$\delta_{17} = \delta_{16} + \sum_{x=0}^{7} 2^{16x} P_{x17}$	$\zeta_{17} = \zeta_{16} + \sum_{x=8}^{15} 2^{16x} P_{x17}$	$\eta_{17} = \eta_{16}$	$\theta_{17} = \theta_{16}$
..	..	..	..	..	..
$j-17$	..	δ_{j-17}	ζ_{j-17}	η_{j-17}	θ_{j-17}
$j-16$	..	$\delta_{j-16} = \delta_{j-17}$	$\zeta_{j-16} = \zeta_{j-17}$	$\eta_{j-16} = \eta_{j-17}$	$\theta_{j-16} = \theta_{j-17}$
$j-15$	..	$\delta_{j-15} = \delta_{j-16} + \sum_{x=0}^{7} 2^{16x} P_{xj-15}$	$\zeta_{j-15} = \zeta_{j-16} + \sum_{x=8}^{15} 2^{16x} P_{xj-15}$	$\eta_{j-15} = \eta_{j-16}$	$\theta_{j-15} = \theta_{j-16}$
..	..	..	..	..	..
$j-1$	..	δ_{j-1}	ζ_{j-1}	η_{j-1}	θ_{j-1}
j	..	$\delta_j = \delta_{j-1}$	$\zeta_j = \zeta_{j-1}$	$\eta_j = \eta_{j-1}$	$\theta_j = \theta_{j-1}$
$j+1$	..	$\delta_{j+1} = \delta_j + ?$	$\zeta_{j+1} = \zeta_j + ?$	$\eta_{j+1} = \eta_j$	$\theta_{j+1} = \theta_j$
..	..	..	..	..	..
b	..	..	..	..	..

$$0 = H_{00} = H_{10} = H_{20} = H_{30} \tag{5.18}$$

$$g.H[0] = H_{0j} \tag{5.19}$$

$$g.H[1] = H_{1j} \tag{5.20}$$

$$0 = p_S(H_0{}' - H_0{}'' + \sum_{x=0}^{7} 2^{16x} P_x{}'') \tag{5.21}$$

$$0 = p_S(H_1{}' - H_1{}'' + \sum_{x=8}^{15} 2^{16x} P_x{}'') \tag{5.22}$$

$$0 = p_S(H_2{}' - H_2{}'') \tag{5.23}$$

$$0 = p_S(H_3{}' - H_3{}'') \tag{5.24}$$

$$0 = p_{cp}(H_0{}' - H_0{}'') \tag{5.25}$$

$$0 = p_{cp}(H_1{}' - H_1{}'') \tag{5.26}$$

$$0 = p_{cp}(H_2{}' - H_2{}'') \tag{5.27}$$

$$0 = p_{cp}(H_3{}' - H_3{}'') \tag{5.28}$$

STARK D. We extend STARK C by adding a plookup check to the input pixels. The domain of a pixel is 16 bits, and we allow every pixel to be in the range $[0, 2^{16} - 2]$, which allows every 16-bit value but 0xFFFF (as described in Section 4.3.9). The decision of excluding 0xFFFF (65535) does not influence our industry example use case since the maximum meaningful temperature is less than 2000 °C (16-bit representation: 0x4E20 = 20000).

For this construction, we assume the length of the input vector P to be a multiple of 16.

The implementation for STARK D is located at `/code/rust/src/stark/stark_d.rs`, and we use the same implementation of Rescue-Prime as for STARK B and STARK C, located at `/code/rust/src/rescue/p128_m4_c2_s128.rs`.

Table 5.8 shows the AET of STARK D, using example pixel values as input. We omit the values of the hash state and denote them with δ_i, ζ_i, η_i, and θ_i. The columns with indices 20 to 28 belong to plookup, as defined in Section 4.3.4. Column f from the definition of plookup are columns P_0 to P_{15} in STARK D. Since f is spread out over 16 columns, we need to modify the plookup structure. All columns for plookup remain the same, except for F_f and f. Furthermore, we need to define random β and γ, but for now we assume that we have some source of randomness to draw β and γ from. Section 4.3.8 describes how to draw randomness in a STARKs setting. All values dependent on β and γ are denoted by ϕ_i, χ_i, ψ_i, and ω_i.

Table 5.8: Algebraic Execution Trace for STARK D.

Index	0	..	15	16	..	19	20	21	22	23	24	25	26	27	28
Step i	P_0	..	P_{15}	H_0	..	H_3	t	s	F_f	F_t	G	R	f_f	f_t	f_s
0	?	..	?	0	..	0	0	0	1	1	1	?	1	1	1
1	57	..	68	δ_1	..	θ_1	1	1	ϕ_1	χ_1	ψ_1	ω_1	1	1	1
2	57	..	68	δ_2	..	θ_2	2	2	ϕ_2	χ_2	ψ_2	ω_2	1	1	1
3	57	..	68	δ_3	..	θ_3	3	3	ϕ_3	χ_3	ψ_3	ω_3	1	1	1
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
14	57	..	68	δ_{14}	..	θ_{14}	14	14	ϕ_{14}	χ_{14}	ψ_{14}	ω_{14}	1	1	1
15	57	..	68	δ_{15}	..	θ_{15}	15	15	ϕ_{15}	χ_{15}	ψ_{15}	ω_{15}	1	1	1
16	57	..	68	δ_{16}	..	θ_{16}	16	16	ϕ_{16}	χ_{16}	ψ_{16}	ω_{16}	1	1	1
17	78	..	72	δ_{17}	..	θ_{17}	17	17	ϕ_{17}	χ_{17}	ψ_{17}	ω_{17}	1	1	1
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..

We introduce additional periodic columns, which help us aggregate all pixels into F_f. Table 5.9 shows the periodic columns for STARK D, which extends the periodic columns of STARK B by the Cycle Mask $p_{X_x}, x \in [0, 15]$, a 16x16 identity matrix. The constraint for F_f (as defined in Equation 4.13) gets replaced by the constraint described in Equation 5.29, where Equation 5.29 uses the helper function $S(a, f_x)$ defined in Equation 4.10.

$$0 = -{F_f}'' + \sum_{x=0}^{15} p_{X_x} \cdot S((1+\beta) \cdot (\gamma + {P_x}''), {f_f}'') \cdot {F_f}' \tag{5.29}$$

Since we feed one pixel per step into F_f, we need copy constraints for the pixels within a cycle. Equation 5.30 serves as 16 additional transition constraints for $x \in [0, 15]$ which ensure that the pixels do not change within one cycle.

$$\forall x \in [0, 15] : 0 = \text{Pixel Copy Mask} \cdot ({P_x}'' - {P_x}') \tag{5.30}$$

We would need another flag as a periodic column definition for this copy constraint. However, we can construct this flag out of the existing flags Hask Mask and Copy Mask. It turns out that we could construct any mask using the periodic columns defining the identity matrix without increasing the degree of our constraints, allowing versatile constructions of masks. Equations 5.31 to 5.35 show some examples of crafting masks from other masks for STARK D.

Table 5.9: Value assignment for the periodic columns for STARK D.

Index	0	1	2	3	..	10	11	12	..	25	26
	p_H	p_{cp}	p_S	p_{C_0}	..	p_{C_7}	p_{X_0}	p_{X_1}	..	$p_{X_{14}}$	$p_{X_{15}}$
0	0	0	1	e	..	e	1	0	..	0	0
1	1	0	0	e	..	e	0	1	..	0	0
2	1	0	0	e	..	e	0	0	..	0	0
..	..	..	..	..	..	..	..	..	..	..	..
13	1	0	0	e	..	e	0	0	..	0	0
14	1	0	0	e	..	e	0	0	..	1	0
15	0	1	0	e	..	e	0	0	..	0	1

$$\text{Hash Mask} = 1 - \text{Statistics Mask} - \text{Copy Mask} \tag{5.31}$$

$$\text{Hash Mask} = \sum_{i=1}^{14} X_i \tag{5.32}$$

$$\text{Copy Mask} = X_{15} \tag{5.33}$$

$$\text{Statistics Mask} = X_0 \tag{5.34}$$

$$\text{Pixel Copy Mask} = \sum_{i=1}^{15} X_i \tag{5.35}$$

The remaining boundary and transition constraints are equal to those used in the STARK C and plookup constructions. Both constructions co-exist inside one execution trace and are linked via the modified transition constraints defined in Equations 5.29 and 5.30. The length of the AET of plookup dominates over the AET of STARK C, so the overall length is the next largest power of two greater than or equal to $|s|$.

Randomness. We require randomness for β and γ for correctly performing the plookup check. Section 4.3.8 describes how to derive randomness inside a STARKs setting, and the solution is to compute a hash inside the STARK of all values that define our polynomials $F(\beta, \gamma)$ and $G(\beta, \gamma)$. As already mentioned in Section 4.3.8, it is required to compute a hash of P_x for $x \in [0, 15]$ and s. We do not need to include t, as t is public information and constrained by our transition constraints of plookup.

To correctly derive randomness, we need to add more columns to our AET of STARK D that compute the hash of the required values. It turns out that this additional construction is similar to the construction of STARK B and includes extra overhead for hashing s. Since the additional construction already computes a hash of our input, it is evident that

STARK B consumes less space than STARK D (including deriving randomness). The design of STARK B is more efficient in terms of space, as STARK D is STARK B plus overhead.

Conclusion. We presented constructions for hashing the video, and we showed that the compression technique presented in Section 4.3.9 makes the construction less efficient. It turns out that STARK B is the best choice within the options presented. STARK B performs better when using a wider hash state size m and a larger rate r_p, as described in the definition of STARK B. Furthermore, a larger hash state reduces the number of rounds within one permutation and reduces space consumption in the AET.

Let us now elaborate on the possibilities of saving space in the AET using the parameter derivation scripts introduced in Section 5.1 for Rescue-Prime and Griffin. For 128-bit security of the hash functions, Rescue-Prime requires 8 rounds with a state size greater than 8 and a capacity of 1. Griffin requires at least 9 rounds for each viable scenario (state size < 100). When reducing the security of the hash functions to 100 bits, Griffin also requires 8 rounds for state sizes greater than 8. Meanwhile, Griffin was released to the public [39]. However, the hash function changed, and we use an older version. The new version requires more rounds than the old version for the same level of security.

A permutation using 8 rounds is very meaningful in saving space in the AET, as 8 is a power of two, and the periodic cycle may be reduced to 8 instead of 16, effectively shrinking the length of the AET by factor two. The next viable option is to reduce the number of rounds to 4, the following minor power of two. However, such a low number of rounds is not achievable with meaningful parameters for the hash functions.

5.2.2 Hashing and Statistics

We extend the STARK from Section 5.2.1 by computing statistics on the input alongside the hash value as public information of our proof. The statistics we cover are the average (arithmetic mean), the minimum, the maximum, the standard deviation, and the median. Section 4.3 elaborates on the problems arising in the STARKs setting for computing these statistics and demonstrates possible solutions, which we use in the present section.

STARK E. We constructed STARK D so that, on average, one pixel gets processed in one step. Therefore we can combine the solutions for computing statistics with STARK D without significant modifications. All STARKs operate on the same input and enforce their constraints in parallel without interfering with each other. However, STARK D uses the inefficient approach of compressing pixels. Therefore we use the approach of STARK B instead, where we ensure that, on average, one pixel gets processed in one step.

For this construction, we assume the length of the input vector P to be a multiple of 8.

We use `/code/rust/src/rescue/p128_m9_c1_s128.rs` for hashing the pixels in the construction of STARK E. The number of rounds is 8, the state size is 9, the capacity is 1, and the minimum security of the hash function is 128 bits. The implementation of STARK E is located at `/code/rust/src/stark/stark_e.rs`.

Table 5.10 shows the assignment of periodic columns for STARK E, where we only use the identity matrix and round constants as periodic columns, as we can construct all

Table 5.10: Value assignment for the periodic columns for STARK E.

Index	0	..	17	18	19	..	24	25
	p_{C_0}	..	$p_{C_{17}}$	p_{X_0}	p_{X_1}	..	p_{X_6}	p_{X_7}
0	e	..	e	1	0	..	0	0
1	e	..	e	0	1	..	0	0
2	e	..	e	0	0	..	0	0
..	..	..	..	..	..	..	..	..
5	e	..	e	0	0	..	0	0
6	e	..	e	0	0	..	1	0
7	e	..	e	0	0	..	0	1

possible flag combinations from the identity matrix. As the state size is 9, we have 18 columns defining the round constants for Rescue-Prime.

The AET of STARK E is shown in Table 5.11 and uses a STARK B construction for hashing the pixels. Columns P_0 to P_7 hold the input pixels, and 8 pixels are absorbed in each absorption phase at steps $x = 8k + 1$ where $k \in [0, (j-1)/8]$ and j is the number of pixels in the video. Similar to STARK D, we need pixel copy constraints for pixels P_0 to P_7 within one cycle, as we use one pixel at one step each for computing statistics. Equation 5.36 describes the pixel copy constraints for STARK E, utilizing the pixel copy flag p_{cp} defined in Equation 5.38 (note that the number of rounds matches the cycle length - we do not need a copy flag for the hash state for STARK E; p_{cp} has a different meaning in STARK D).

Table 5.11: Algebraic Execution Trace for STARK E.

Index Step i	0 .. 16 STARK B	17 sum	18 var	19 min	20 max	21 .. 28 $\omega_{l0..7}$	29 .. 36 $\omega_{h0..7}$	37 f_l	38 f_h	39 .. 46 $med_{0..7}$	47 .. 54 $\omega_{m0..7}$	55 z	56 .. 113 $\mathfrak{R}$
0	..	0	0	$v = 65534$	0	?	?	?	?	0	?	1	..
1	..	27	36	27	27	65507	27	0	1	13	13	δ_1	..
2	..	45	261	18	27	9	9	0	0	14	1	δ_2	..
3	..	76	265	18	31	13	4	1	1	18	4	δ_3	..
4	..	90	626	14	31	4	17	0	0	27	9	δ_4	..
5	..	123	626	14	33	19	2	1	1	30	3	δ_5	..
6	..	153	635	14	33	16	3	1	0	31	1	δ_6	..
7	..	166	1035	13	33	1	20	0	0	33	2	δ_7	..
8	..	204	1060	13	38	25	5	1	1	38	5	δ_8	..
9	..	?	?	?	?	?	?	?	?	?	?	?	..
..	..	..	..	..	..	..	..	..	..	..	..	..	..
b	..	..	..	..	..	..	..	..	..	..	..	..	..

Computing permutation rounds is now split into computing the first round (including the absorption phase) and all other rounds. Listing 5.5 defines the transition constraints for computing the first round, and Listing 5.4 defines the transition constraints for computing the remaining rounds. Note that Listing 5.5 is Listing 5.4, with the addition of lines 3 and 11-13 (and a different function name). The flag p_{H_f} for enforcing the first round is defined in Equation 5.39, and the flag p_{H_r} for enforcing the remaining rounds is defined in Equation 5.40. The parameter `pixels` holds the pixels P_0'' to P_7''.

Listing 5.5: Implementation of enforcing transition constraints for the first round the Rescue-XLIX permutation of Rescue-Prime including absorption of pixels.

```
1  pub fn enforce_first_round<E: FieldElement + From<Elem>>(
2      result_slice: &mut [E],
3      pixels: &[E],
4      current_slice: &[E],
5      next_slice: &[E],
6      round_constants: &[E],
7      flag: E,
8  ) {
9      let mut step1 = [E::ZERO; STATE_WIDTH];
10     step1.copy_from_slice(current_slice);
11     for i in 0..RATE {
12         step1[i] += pixels[i];
13     }
14     apply_sbox(&mut step1);
15     ..
16     ..
17 }
```

Table 5.11 shows the AET of STARK E for the example input sequence {27, 18, 31, 14, 33, 30, 13, 38} with size $j = 8$. The columns with indices 0 to 16 are omitted since these columns hold STARK B. Details of the inner workings of STARK B are described in Table 5.4 and Section 5.2.1. This time, we use a modified STARK B construction with absorbing 8 elements instead of two in one absorption phase. This implies that we need 8 columns holding the input. We have a hash state size of 9, where the rate is 8, and one element is reserved as capacity. Therefore the size of STARK B in Table 5.11 is 17 elements, 8 for the input pixels and 9 for the hash state.

Statistics. Columns 17 to 55 describe the structure for computing statistics, as introduced in Section 4.3.2. We use the approach with distance checks to compute the minimum and maximum (see Section 4.3.6) since it requires the same plookup structure for computing the median and would also nicely combine with the pixel compression approach from STARKs C and D. The structure remains the same, except for spreading some columns over multiple columns: As we require randomness for certain checks, we design affected columns for deriving randomness using the same structure as we use it for hashing pixels. The structure allows us to process one element in one step on average, and the identity matrix p_X lets us select the proper value at each step. We also require copy constraints, such that the values remain the same within one cycle. In short, the layout of the columns ω_l, ω_h, med, ω_m, and s is the same as for the input pixels.

Table 5.12: Remaining columns $\mathfrak{R}$ of the AET for STARK E.

Index	56 .. 63	64	65	66	67	68	69 .. 104	105 .. 113
Step i	$s_{0..7}$	F	G	R	f_f	f_s	$\mathfrak{H}_{f_{0..35}}$	$\mathfrak{H}_{s0..8}$
0	0	F_t	1	?	1	1	..	..
1	1	δ_1	ζ_1	η_1	1	1	..	..
2	1	δ_2	ζ_2	η_2	1	1	..	..
3	1	δ_3	ζ_3	η_3	1	1	..	..
4	1	δ_4	ζ_4	η_4	1	1	..	..
5	2	δ_5	ζ_5	η_5	1	1	..	..
6	2	δ_6	ζ_6	η_6	1	1	..	..
7	2	δ_7	ζ_7	η_7	1	1	..	..
8	3	δ_8	ζ_8	η_8	1	1	..	..
9	3	$\delta_9 = \delta_8$	ζ_9	η_9	0	1	..	..
..	..	..	..	..	..	..	..	..
$4j + \|t\| - 1$	$t_{\|t\|-1}$	δ_8	$\zeta_{4j+\|t\|-1}$	$\eta_{4j+\|t\|-1} = \mathbf{0}$	0	1	..	..
$4j + \|t\|$	?	δ_8	$\zeta_{4j+\|t\|-1}$	$\eta_{4j+\|t\|-1}$	0	0	..	..
..	..	..	..	..	..	..	..	..
b	..	..	..	..	..	..	..	..

The AET for STARK E has grown to a significant size. Therefore we denote the columns for computing the plookup check and the randomness by $\mathfrak{R}$, and $\mathfrak{R}$ is defined in Table 5.12.

Pseudo-Randomness. As discussed in Section 4.3.8, the pseudo-randomness needs to depend on all values defining the structure we want to do random sampling on, the polynomials F, G, and the two polynomials defined in z. The values defining these polynomials are the input pixels P, ω_l, ω_h, med, ω_m, and s. Using the Fiat-Shamir heuristic introduced in Section 4.3.8, we derive pseudo-randomness by hashing all these values.

The input pixels P are already hashed within the STARK B construction (columns 0 to 16). For each of the remaining values (ω_l, ω_h, med, ω_m, and s) we also build a STARK B construction identical to hashing the input pixels P. The columns $\mathfrak{H}_f$ (columns 69 to 104) hold the hash state of 4 instances of our STARK B construction, one of each hashing ω_l, ω_h, med, and ω_m. The result of this hash function is located at step j, the length of

the input pixels. $\mathfrak{H}_s$ (columns 105 to 113) holds the hash state of the values of s, and the hash result resides at step $4j + |t|$.

We use boundary constraints to place the results of these hash values into the public inputs as $g.H_P$, $g.H_{\omega_l}$, $g.H_{\omega_h}$, $g.H_{med}$, $g.H_{\omega_m}$, and $g.H_s$. One hash comprises $r_p = 8$ elements (the hash always has the size of the rate).

For the plookup check, β and γ are derived from $g.H_P$, $g.H_{\omega_l}$, $g.H_{\omega_h}$, $g.H_{\omega_m}$, and $g.H_s$ by hashing using `/code/rust/src/rescue/p128_m4_c2_s128.rs`, where β is the resulting hash at index 0 and γ at index 1.

The computation of the median (see Section 4.3.7) requires random λ, which is derived from $g.H_P$ and $g.H_{med}$ by hashing using `/code/rust/src/rescue/p128_m4_c3_s128.rs` where the resulting hash is λ.

Range Check (plookup). We want to perform a plookup check with t being the full 16-bit domain, sorted ascending ($\{0, 1, 2, .., 2^{16} - 1\}$). f is the multiset comprising the input pixels P, ω_l, ω_h, and ω_m. s is the multiset union of f and t. The check gets performed as described in Section 4.3.4 with slight simplifications. t itself is public information, so we do not need to place t in the AET - also, F_t is not required. F_{f_0} requires to be 1 by the boundary constraints. For STARK E, we set F_{f_0} to the result of computing F_t. The randomness is public information (derived by the hashes), so F_t is also public and precomputed. Since F_f now holds the whole polynomial F, we use F as column name instead of F_f. The transition constraints for computing F remain the same as for the AET of plookup, except we need to select the proper values in each step using the identity matrix. Furthermore, we absorb all values of F at once within one step, which increases the degree of this transition constraint. We keep this degree high for now (for simplicity of the AET structure) and fix the problem later by using more columns that split the multiplication and reduce the degree.

Constraints. All constraints remain the same for each separate STARK construction unless stated otherwise in this section. As mentioned before, we need copy constraints for P, ω_l, ω_h, med, ω_m, and s as defined in Equations 5.36 to 5.37. Moreover, the transition constraints for applying rounds of hash functions now use the functions `enforce_round` and `enforce_first_round` (see Listings 5.4 and 5.5) utilizing the flags from Equations 5.39 and 5.40. The main change of transition constraints is selecting the correct pixel using the identity matrix at each step. One more change simplifies the accumulation of values for F of plookup.

$$\forall x \in \{P, \omega_l, \omega_h, med, \omega_m\} : \ \forall y \in [0, 7] : \ 0 = p_{cp}({x_y}' - {x_y}'') \tag{5.36}$$

$$\forall y \in [0, 7] : \ 0 = p_{cp}({s_y}' - {s_y}'') \tag{5.37}$$

$$p_{cp} = \sum_{x=1}^{x=7} p_{X_x} \tag{5.38}$$

$$p_{H_f} = p_{X_0} \tag{5.39}$$

$$p_{H_r} = \sum_{x=1}^{x=7} p_{X_x} = p_{cp} \tag{5.40}$$

The transition constraints for computing the statistics are adapted according to Equations 5.41 to 5.48 utilizing the identity matrix.

$$0 = {f_f}'' \left(sum' - sum'' + \sum_{x=0}^{7} p_{X\,x} {P_x}'' \right) \tag{5.41}$$

$$0 = {f_f}'' \left(var' - var'' + \sum_{x=0}^{7} p_{X\,x} ({P_x}'' - g.avg\_rounded)^2 \right) \tag{5.42}$$

$$0 = {f_f}'' \left(-max'' + (1 - {f_h}'')max' + {f_h}'' \sum_{x=0}^{7} p_{X\,x} {P_x}'' \right) \tag{5.43}$$

$$0 = {f_f}'' \left(-min'' + {f_l}''min' + (1 - {f_l}'') \sum_{x=0}^{7} p_{X\,x} {P_x}'' \right) \tag{5.44}$$

$$0 = {f_f}'' \left({f_h}'' \left(-max' + \sum_{x=0}^{7} p_{X\,x} {P_x}'' \right) + (1 - {f_h}'') \left(max' - \sum_{x=0}^{7} p_{X\,x} {P_x}'' \right) - \sum_{x=0}^{7} p_{X\,x} {\omega_{h}}_x'' \right) \tag{5.45}$$

$$0 = {f_f}'' \left({f_l}'' \left(-min' + \sum_{x=0}^{7} p_{X\,x} {P_x}'' \right) + (1 - {f_l}'') \left(min' - \sum_{x=0}^{7} p_{X\,x} {P_x}'' \right) - \sum_{x=0}^{7} p_{X\,x} {\omega_{l}}_x'' \right) \tag{5.46}$$

$$0 = {f_f}'' \left(\sum_{x=0}^{7} p_{X\,x} {med_x}'' - \sum_{x=0}^{7} p_{X\,x} {med_{(x-1) \bmod 8}}' - \sum_{x=0}^{7} p_{X\,x} {\omega_{m}}_x'' \right) \tag{5.47}$$

$$0 = {f_f}'' \left(z' \frac{\lambda + \sum_{x=0}^{7} p_{X\,x} {P_x}''}{\lambda + \sum_{x=0}^{7} p_{X\,x} {med_x}''} - z'' \right) \tag{5.48}$$

The boundary constraints for step 0 remain the same, except for F_{f_0}, which is now set to F_t instead of 1. The boundary constraints for the resulting statistics are defined in

Equations 5.52 to 5.54, and their meaning is discussed in Section 4.3.2. The boundary constraints for the hash states are defined in Equations 5.49 to 5.51.

$$\forall x \in \{H_P, H_{\omega_l}, H_{\omega_h}, H_{med}, H_{\omega_m}, H_s\} : \ \forall y \in [0,8] : \ 0 = x_{y0} \tag{5.49}$$

$$\forall x \in \{H_P, H_{\omega_l}, H_{\omega_h}, H_{med}, H_{\omega_m}\} : \ \forall y \in [0,7] : \ g.x[y] = x_{y_j} \tag{5.50}$$

$$\forall y \in [0,7] : \ g.H_s[y] = H_{sy_{4j+|t|}} \tag{5.51}$$

$$\forall x \in \{sum, var, min, max\} : \ g.x = x_j \tag{5.52}$$

$$g.med\_low = med_{j/2} \tag{5.53}$$

$$g.med\_high = med_{j/2+1} \tag{5.54}$$

As defined in Equations 5.55 to 5.57, transition constraints for plookup are modified for the accumulation process of F and G using the identity matrix. The modified boundary constraints for plookup are defined in Equations 5.58 to 5.63. Note that the first element of s (placed at s_{70}, see Equation 5.59) does not get hashed, but the first element is always the first element of t for correct instances of plookup. In our case, the first element of t is 0. We could argue that s_{70} influences the hash value, as we set the IV of the hash function to $s_{70} = 0$, which is implicitly the case as the IV comprises zeroes only. In conclusion, all elements of s influence the computation of the hash; therefore, Fiat-Shamir gets applied correctly to the present plookup construction. Equations 5.55 and 5.56 use the helper function $S(a, f_x)$ defined in Equation 4.10.

$$0 = -F'' + F' \prod_{e \in f} S\left((1+\beta)\left(\gamma + \sum_{x=0}^{7} p_{X\,x} e_x{}''\right), f_f{}''\right) \text{ for } f = \{P, \omega_l, \omega_h, \omega_m\} \tag{5.55}$$

$$0 = S\left(\gamma(1+\beta) + \sum_{x=0}^{7} p_{X\,x} s_{(x-1) \bmod 8}{}' + \beta \sum_{x=0}^{7} p_{X\,x} s_x{}'', f_s{}''\right) G' - G'' \tag{5.56}$$

$$0 = F'' - G'' - R'' \tag{5.57}$$

$$F_t = F_0 \tag{5.58}$$

$$0 = s_{70} \tag{5.59}$$

$$1 = G_0 = f_{f\,0} = f_{s0} \tag{5.60}$$

$$1 = f_{f\,j} = f_{s4j+|t|-1} \tag{5.61}$$

$$0 = f_{f\,j+1} = f_{s4j+|t|} \tag{5.62}$$

$$0 = R_{4j+|t|-1} \tag{5.63}$$

Table 5.13: Algebraic Execution Trace for STARK F.

Index	0 .. 7	8 .. 16	17	18
Step i	P_0 .. P_7	H_0 .. H_8	sum	var
0	?	0	0	0
1	27	..	27	36
2	18	..	45	261
3	31	..	76	265
4	14	..	90	626
5	33	..	123	626
6	30	..	153	635
7	13	..	166	1035
8	38	..	204	1060
..	..	..	..	..
b = 15	..	..	..	..

Public Inputs. This paragraph serves as a summary of public inputs for STARK E. As members of the public inputs, we have the length j of the pixel input vector as $g.j$; all resulting hashes $g.H_P$, $g.H_{\omega_l}$, $g.H_{\omega_h}$, $g.H_{med}$, $g.H_{\omega_m}$, and $g.H_s$; and all resulting statistics $g.sum$, $g.avg\_rounded$, $g.var$, $g.min$, $g.max$, $g.med\_low$, and $g.med\_high$.

The verifier needs to check outside the STARK protocol in $\mathbb{R}$ that $g.avg\_rounded = \left\lfloor \frac{g.sum}{g.j} \right\rceil$ (recall that $g.avg\_rounded$ is used to compute the variance in the STARK) and computes the standard deviation $\sigma = \sqrt{g.var}$ and the median as $\frac{g.med\_low + g.med\_high}{2}$.

STARK F. The construction of STARK E showed us that the overhead for computing the minimum, maximum, and median is significant. Therefore we construct a lightweight STARK that only computes the hash of the input, the average, and the standard deviation as STARK F. plookup and randomness is not required for STARK F, which dramatically boosts performance. In short, STARK F is STARK E being cut off after column 18. The implementation of STARK F is located at `/code/rust/src/stark/stark_f.rs`.

For this construction, we assume the length of the input vector P to be a multiple of 8. The AET of STARK F is shown in Table 5.13 using the example input sequence from STARK E {27, 18, 31, 14, 33, 30, 13, 38} with length $j = 8$. The periodic columns are equal to those of STARK E, and all constraints are the same as in STARK E (copy constraints of pixels within one cycle, hash state round enforcement, computation of the sum, and variance).

5.2.3 Hashing and Statistics on an ROI

Computing statistics on an ROI of the video is done by placing one frame within one periodic cycle, as discussed in Section 4.3.3. As the periodic cycle must have a length of a power of two, we will waste some space within the cycle, as the video resolution is not a power of two. The video provided by our industry partner has a resolution of 382x288 pixels, so one frame consists of 110016 pixels. We process one pixel at one step, using a periodic cycle of length 131072, the closest power of two greater than 110016.

STARK G. For simplicity, we extend STARK F to show the principle of wrapping STARKs into the concept of frames. This concept may then be used to bring the ROI approach to STARK E, but we omit this step. The implementation of STARK G is located at `/code/rust/src/stark/stark_g.rs`. For this construction, we assume the length of the input vector P to be a multiple of 110016, as we want to fit frames into the AET.

The AET of STARK G comprises the same columns as the AET of STARK F (see Table 5.13), but we use different transition and boundary constraints. Furthermore, we also extend the periodic columns by two flags, where p_{f_h} defines when pixels need to be absorbed into the hash state, and p_{f_s} defines the ROI. Pixels are placed in the AET for the first 382x288 = 110016 steps within each cycle of 131072 steps. The value of the pixels in the remaining steps of each cycle does not matter, as these values are not absorbed into the hash state, because the hash flag p_{f_h} is 0 for these steps.

The periodic columns for STARK G are shown in Table 5.14. The periodic columns for the identity matrix and round constants repeat after 8 steps - we have periodic values inside our periodic values. Recall that periodic columns are public information; therefore, the ROI is public, and the video dimensions are probably reconstructible using the information from the ROI definition.

The transition constraints are modified variants of those of STARK F, where either new values are processed or the previous value is copied. Equations 5.64 and 5.65 list the transition constraints for the statistics. For enforcing rounds of the hash function, the functions `enforce_round` and `enforce_first_round` (see Listings 5.4 and 5.5) are used where the flags are $p_{H_f} p_{f_h}$ and $p_{H_r} p_{f_h}$, respectively (see Equations 5.39, 5.40, and Table 5.14 for definitions of the flags). The copy transition constraints for the hash state are defined in Equation 5.66. The boundary constraints at step 0 remain untouched, while the boundary constraints for linking the result with the public inputs need to be fetched from step 131072 $\left(\frac{j}{110016} - 1\right) + 110015$, where j is the length of the pixel input vector P.

Table 5.14: Value assignment for the periodic columns for STARK G.

Index	0	..	17	18	19	..	24	25	26	27
	p_{C_0}	..	$p_{C_{17}}$	p_{X_0}	p_{X_1}	..	p_{X_6}	p_{X_7}	p_{f_h}	p_{f_s}
0	$p_{C_{00}}$	..	$p_{C_{170}}$	1	0	..	0	0	1	?
1	$p_{C_{01}}$	..	$p_{C_{171}}$	0	1	..	0	0	1	?
2	$p_{C_{02}}$	..	$p_{C_{172}}$	0	0	..	0	0	1	?
..	..	..	..	..	..	..	..	..	..	..
5	$p_{C_{05}}$	..	$p_{C_{175}}$	0	0	..	0	0	1	?
6	$p_{C_{06}}$	..	$p_{C_{176}}$	0	0	..	1	0	1	?
7	$p_{C_{07}}$	..	$p_{C_{177}}$	0	0	..	0	1	1	?
8	$p_{C_{00}}$	..	$p_{C_{170}}$	1	0	..	0	0	1	?
..	..	..	..	..	..	..	..	..	..	..
110015	$p_{C_{07}}$	..	$p_{C_{177}}$	0	0	..	0	1	1	?
110016	$p_{C_{00}}$	..	$p_{C_{170}}$	1	0	..	0	0	0	0
..	..	..	..	..	..	..	..	..	..	..
131071	$p_{C_{07}}$	..	$p_{C_{177}}$	0	0	..	0	1	0	0

$$0 = p_{f_s}\left(sum' - sum'' + \sum_{x=0}^{7} p_{X\,x} {P_x}''\right) + (1 - p_{f_s})(sum' - sum'') \tag{5.64}$$

$$\begin{aligned} 0 =& p_{f_s}\left(var' - var'' + \sum_{x=0}^{7} p_{X\,x}({P_x}'' - g.avg\_rounded)^2\right) \\ &+ (1 - p_{f_s})(var' - var'') \end{aligned} \tag{5.65}$$

$$\forall x \in [0, 8] : \; 0 = (1 - p_{f_h})(H'_x - H''_x) \tag{5.66}$$

5.2.4 Optimization of the AET Design

The main parameters influencing the performance of a STARK are the length T of the AET, the width w of the AET, and the maximum degree d_{max} of all transition constraints. Various comparison metrics exist for the performance of STARKs, where we want to focus on minimizing $w \cdot T \cdot d_{max}$ [2].

There are potential performance optimizations for all STARK constructions that use plookup, as the columns for s, F, G, and R are noticeably longer than the other columns. We can use multiple columns for defining s, F, and G, which reduces the length of the AET significantly (e.g. for STARK E: when $j > |t|$ and s is defined over 5 columns, then the length of the AET reduces to j - this is a reduction of length by factor 5 and the width increases by $\sim 63\%$), so the size ($w \cdot T$) of the optimized AET is 1/3 of the initial AET, while also reducing d_{max} to 3. Since we use multiple columns for s, we also need more columns to compute the hash $\mathfrak{H}_s$ of s.

When $d = 3$ (128-bit prime field), we can reduce the degree of the transition constraint for columns F and G (see Equations 5.55 and 5.56) by splitting up the columns into multiple ones to reduce the degree to ≤ 3.

STARK E is very interesting for performing optimization, and the $\mathfrak{R}$ part of the AET of the optimized STARK E (referred to as STARK E (opt)) is shown in Table 5.15. The base part of the AET of STARK E (opt) remains the same (see Table 5.11). The implementation of STARK E (opt) is located at `/code/rust/src/stark/stark_e_opt.rs`. This optimization lessens the metric $w \cdot T \cdot d_{max}$, which should lead to a more performant STARK in theory.

For STARK E (opt) we assume the length j to be a multiple of 40 and $j > |t|$, and t allows values in the range [0,65520], such that every cycle (8 steps) is either filled up or empty. Recall that we place the first element of s in step 0 and all consecutive steps hold 40 values of s, so $|s| - 1$ needs to be a multiple of 40.

STARK F also opens room for optimization by making the trace wider and shorter, therefore slightly shirinking $w \cdot T \cdot d_{max}$. The AET of the optimized STARK F (referred to as STARK F (opt)) is shown in Table 5.16. The implementations of STARK F (opt) are located at `/code/rust/src/stark/stark_f_opt_m<value of m>.rs`.

For STARK F (opt), we place m blocks of 8 columns holding input pixels next to each other. The rate of the hash function is $8m$, and the state size is $8m + 1$. This design allows multiple constructions of STARK F (opt), and the results get compared in Chapter 6. The columns *sum* and *var* accumulate new values additively, so the transition constraint degrees do not increase by widening the AET. For STARK F (opt), we assume the length of the input vector P to be a multiple of $8m$. The length of the AET $b + 1$ is the next largest power of two $\geq \frac{j}{m}$, where j is the length of the input pixel vector.

The idea for STARK F (opt) sparks the motivation to optimize STARK G (the ROI approach based on STARK F). As a first intuitive approach, we aim to improve the quota of "unused cells" within one periodic cycle (steps that are connected via copy constraints for the hash state; we refer to the quota as $\frac{\text{hash round enforcements}}{\text{periodic cycle length}}$ where values closer to 1 are better; the quota of our example with $m = 1$ is $\frac{110016}{131072}$). However, it turns out that the quota cannot be improved with the approach of STARK F (opt) and the trace width limit of 256 elements imposed by the Winterfell library. Viable options for m that keep the quota constant are {1, 2, 4, 8} and all other values for m reduce the quota. Applying the optimization approach of STARK F (opt) to STARK G brings the equal benefit as for STARK F (opt), but only for $m \in \{1, 2, 4, 8\}$.

Table 5.15: Remaining columns $\mathfrak{R}$ of the AET for STARK E (opt).

Index Step i	56..95 $s_{0..39}$	96..98 $F_{0..2}$	99..101 $G_{0..2}$	102 R	103 f_f	104 f_s	105..140 $\mathfrak{H}_{f_{0..35}}$	141..185 $\mathfrak{H}_{s0..44}$
0	0	F_t	1	?	1	1	..	..
1	..	δ_1	ζ_1	η_1	1	1	..	..
2	..	δ_2	ζ_2	η_2	1	1	..	..
..	..	..	..	..	..	..	..	..
$\frac{4j+\lvert t\rvert-1}{5}$	..	$\delta_{\frac{4j+\lvert t\rvert-1}{5}}$	$\zeta_{\frac{4j+\lvert t\rvert-1}{5}}$	$\eta_{\frac{4j+\lvert t\rvert-1}{5}}$	1	1	..	..
$\frac{4j+\lvert t\rvert-1}{5}+1$	?	$\delta_{\frac{4j+\lvert t\rvert-1}{5}+1}$	$\zeta_{\frac{4j+\lvert t\rvert-1}{5}}$	$\eta_{\frac{4j+\lvert t\rvert-1}{5}+1}$	1	0	..	..
..	?	..	..	..	..	..	..	..
j	?	δ_j	$\zeta_{\frac{4j+\lvert t\rvert-1}{5}}$	$\eta_j = \mathbf{0}$	1	0	..	..
j + 1	?	δ_j	$\zeta_{\frac{4j+\lvert t\rvert-1}{5}}$	η_j	0	0	..	..
..	?	..	..	..	..	..	..	..
b	?	..	..	..	..	..	..	..

Table 5.16: Algebraic Execution Trace for STARK F (opt) and STARK G (opt).

Index Step i	0 .. $8m-1$ P_0 .. P_{8m-1}	$8m$ .. $16m$ H_0 .. H_{8m}	$16m+1$ sum	$16m+2$ var
0	?	0	0	0
1	..	..	..	..
2	..	..	..	..
..	..	..	..	..
8	..	..	..	..
..	..	..	..	..
b	..	..	..	..

STARK G (opt) requires m columns in the periodic columns for defining the hash flag p_{f_h} and statistics flag p_{f_s} each since we process m pixels at each step on average. We omit to implement STARK G (opt), as the performance improvement is expected to be the same as for STARK F but slightly worse since the number of periodic columns increases.

The details for the constraints of STARK E (opt) and STARK F (opt) are to be found in the source code of the constructions, the AETs shown in Tables 5.15 and 5.16 shall provide the reader with an idea of the optimization approach. Chapter 6 compares the different solutions in terms of performance.

5.2.5 Overview of STARK Variants

Table 5.17 shows a summary of all STARKs we constructed in the present chapter. n denotes the input length (total number of pixels), and $|t|$ denotes the size of the lookup table t required for the plookup check in the respective STARK variants. The maximum transition constraint degree is 7 for the 64-bit field, as $\alpha = 7$ for that field.

STARKs C and D were designed in the 128-bit field and cannot be ported 1:1 to other fields, as we consume the full 128-bit range of the field elements. We could modify these STARKs to absorb 4 field elements instead of 8, allowing the 62- and 64-bit fields to be used. However, the 62- and 64-bit variants cannot be compared directly to the 128-bit variant, as the AET is twice as long. STARKs C and D exist for demonstration purposes only and are useless without native RAPs support.

From STARK E onwards, the AET width differs for the Rescue-Prime and Griffin variants. Rescue-Prime allows any integer for the size of the hash state, whereas Griffin is constrained on the possible sizes of the hash state. STARKs E, F, and G use a hash state size of 9 for Rescue-Prime, and the closest allowed larger hash state size for Griffin is 12. This is the only difference between these STARKs besides the hash function in use.

Table 5.17: All STARK Variants: Overview and Properties.

STARK Variant	Algebraic Execution Trace			Supported Finite Fields (bit)	Computations
	Length	Width	max. Transition Constraint Degree		
A	$16n$	5	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash
B	$8n$	6	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash
C	n	20	3	128	Inp. Hash (insecure)
D	$n + \|t\|$	29	3	128	Inp. Hash (insecure)
E	$4n + \|t\|$	114 (Resc.) 132 (Griff.)	6 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash, Stats (all pix.): Sum, Std. Dev., Min, Max, Median
E (opt)	n	186 (Resc.) 216 (Griff.)	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash, Stats (all pix.): Sum, Std. Dev., Min, Max, Median
F	n	19 (Resc.) 22 (Griff.)	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash, Stats (all pix.): Sum, Std. Dev.
F (opt m2)	$\frac{n}{2}$	35 (Resc.) 38 (Griff.)	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash, Stats (all pix.): Sum, Std. Dev.
F (opt m4)	$\frac{n}{4}$	67 (Resc.) 70 (Griff.)	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash, Stats (all pix.): Sum, Std. Dev.
F (opt m8)	$\frac{n}{8}$	131 (Resc.) 134 (Griff.)	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash, Stats (all pix.): Sum, Std. Dev.
G	$\frac{131072}{110016}n$	19 (Resc.) 22 (Griff.)	3 (128/62-bit) 7 (64-bit)	128, 64, 62	Inp. Hash, Stats (ROI): Sum, Std. Dev.

Chapter 6

Results

This chapter focuses on the performance analysis of the STARKs presented in Chapter 5. A detailed table containing various performance evaluation results is located at `/code/performance_evaluation.xlsx`. All results and numbers presented in this chapter are derived from this table. Appendices A and B present the performance data based on an excerpt of the data contained in the spreadsheet file in more detail. We use two machines for assessing the performance of our STARKs:

- Desktop PC (8 logical cores, Intel Core i7-4790 CPU @ 3.60GHz × 8, 16 GB DDR3 RAM)
- Cluster (88 logical cores, 2 × Intel Xeon CPU E5-2699 v4 @ 2.20GHz, 512GB DDR4 RAM)

We analyze the STARKs based on full frames (382x288 = 110016 pixels), referred to as *Full Size*. Furthermore, we start with smaller input sizes where we use frame sizes of 65280 pixels (*Half Size*) and 32640 pixels (*Quarter Size*). This definition of different frame sizes impacts the input length, the size of t for plookup checks, and the length of the periodic cycle for STARK G, where we use Rust Features[1] to compile each version based on a features flag.

All STARKs were designed using Rescue-Prime and the 128-bit prime field and were ported to Griffin and the other fields afterwards. Any STARK uses the initial configuration (Rescue-Prime and 128-bit field) if not stated otherwise. The finite fields with 128-bit, 64-bit, and 62-bit moduli are abbreviated with f128, f64, and f62.

Section 6.1 gives an overview and compares the STARKs with each other. The three finite fields available in Winterfell get compared in Section 6.2 using STARK F. The performance of Griffin and Rescue-Prime gets analyzed in Section 6.3. Concurrent computation may be used to create a proof, and the impact thereof is shown in Section 6.4. The proof size and verifier complexity get analyzed in Section 6.5. Section 6.6 discusses the influence of the input length on the performance, and Section 6.7 highlights the results of the STARK F (opt mX) experiment. Finally, Section 6.8 provides numbers for the performance of processing the whole video within STARK F.

[1] `https://doc.rust-lang.org/cargo/reference/features.html`

6.1 Overview and STARKs Comparison

First, we want to provide a general overview of how the STARKs perform relative to each other. Based on the construction of our STARKs (see Table 5.17), we have the following expectations:

- STARK B should be twice as efficient as STARK A since we halved the length of the AET.
- STARK C should be significantly faster than STARK B, as we use the compression approach to reduce the AET's length further.
- STARK D should be more expensive than STARK C, as we extend STARK C by a plookup check.
- The performance of STARK E is expected to be the worst, as the length of the AET is enormous and the maximum degree of transition constraints exceeds three. Fiat-Shamir is used to provide pseudo randomness, being very expensive. We expect that STARK E (opt) improves on STARK E. However, it remains an open question of how much the optimization improves upon STARK E.
- STARK F should be the most efficient approach, as we optimized the space consumption inside the AET, and we only compute statistics that do not require randomness. Recall that STARKs C and D are insecure, but might outperform STARK F.
- STARKs F (opt mX) are an experiment. Trace width times trace length is approximately the same for all optimization levels. The question is if the performance will also be approximately the same.
- STARK G is expected to be slightly more expensive than STARK F, as we use an ROI for computing statistics.

Figure 6.1 shows the comparison of all STARKs in terms of runtime and space consumption. As a first step, the prover needs to build the AET. The second step is to compute the proof based on the AET and AIR. The total time required to compute the proof is the time to build the AET plus the proving time. These two steps need to be performed sequentially.

Building the AET cannot be done in parallel, as the hash functions in use allow little to no parallelism. On the other hand, the proving step benefits from concurrency. Since computing the hash is expected to be the most expensive step when building the AET, we also provide a comparison of plain hashing time versus trace construction time.

Figure 6.1 shows us that most of the trace construction time is spent on computing the hash. Plain hashing time consumes only a fraction of trace building time for STARK E, and the reason is the computation of hashes for Fiat-Shamir. By plain hashing time we refer to hashing the input only.

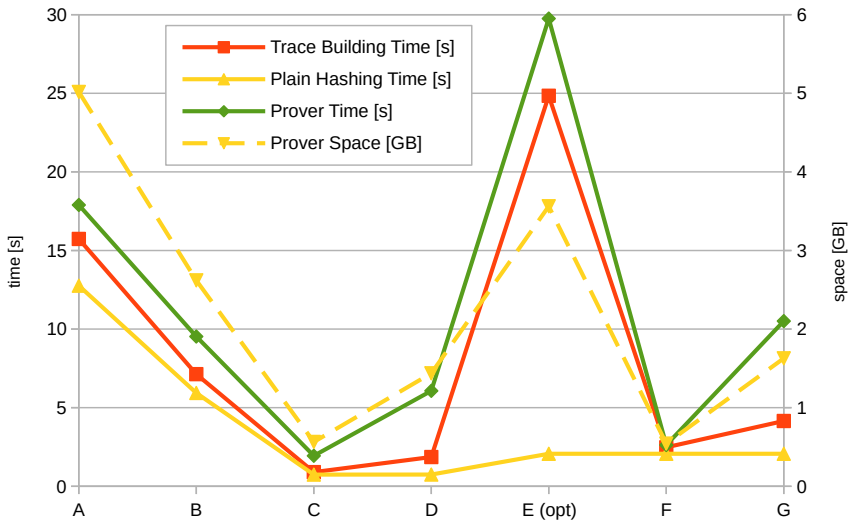

Figure 6.1: All STARKs comparison: Full Size, 1 Frame (128-bit field, Desktop PC, 8 cores).

The optimized variant of STARK E yields drastic performance improvements and shows the importance of a dedicated AET design. STARK E (opt) outperforms STARK E by a factor of 5 to 10 for the construction of the AET, prover time, and space complexity (see Tables B.1 and B.2).

Section 6.7 elaborates on the outcome of the STARK F (opt mX) experiment. Performance numbers are omitted for STARK F (opt mX) in Figure 6.1.

6.2 Finite Fields Comparison

The library Winterfell offers implementations of three prime fields, and the authors describe these as follows: [24]

- The 128-bit prime field was not designed with any significant thoughts to performance.
- The 62-bit prime field offers efficient arithmetic operations and is the most performant prime field available within Winterfell.
- The 64-bit prime field is approximately 15 % slower than the 62-bit prime field.

Rescue-Prime and Griffin use S-Boxes that raise elements to the power of α, where α needs to fulfill specific criteria based on the prime field in use, and α is a parameter of both hash functions. We strive for a small α, as α defines the transition constraint degree for computing rounds of the hash functions. The 128-bit and 62-bit prime fields allow $\alpha = 3$, whereas the 64-bit prime field forces us to use $\alpha = 7$. Since $\alpha = 7$ for the 64-bit

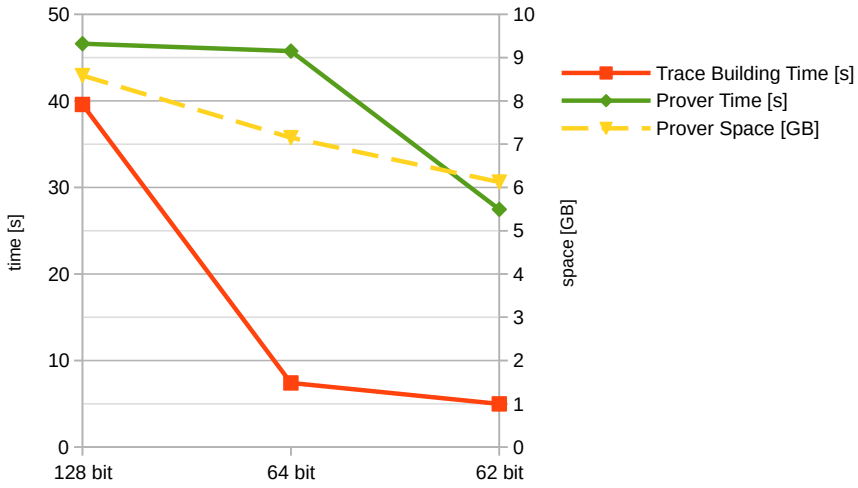

Figure 6.2: Comparing available fields in Winterfell based on STARK F: 16 Frames (Desktop PC, 8 cores).

prime field, we expect the 64-bit prime field to perform the worst. The 62-bit prime field is expected to be the best available field, as $\alpha = 3$ and arithmetic operations are efficient.

Figure 6.2 compares the performance of all three available fields using the example STARK F. **The 62-bit prime field is the best choice.** Surprisingly, the 64-bit prime field outperforms the 128-bit prime field, even though $\alpha = 7$.

6.3 Rescue-Prime vs. Griffin

Meanwhile, Griffin was released to the public [39]. However, the hash function changed, and we use an older version. The new version is expected to be slightly more expensive, as the number of rounds has increased.

Arithmetization friendly hash functions suffer from very inefficient plain hash performance. Rescue-Prime is the best available hash function in the STARKs setting today and is recommended by StarkWare Industries [11]. The inefficiency of Rescue-Prime origins from the high number of exponentiation operations (exponentiation with α^{-1}, a huge exponent) of the S-Boxes, and Griffin tries to improve upon this issue.

Figure 6.3 compares the performance of STARK F (opt m4) based on the hash function in use. We can observe **remarkably better plain hash performance** for the **Griffin** variant. However, the **prover performance** is **slightly worse** for the **Griffin** variant in terms of space consumption. Strong concurrent execution (Cluster: 88 cores) of the proving step outperforms trace building time for the Rescue-Prime example.

It has to be noted that Rescue-Prime uses a hash state of width 9, whereas Griffin uses a hash state of width 12, the closest allowed hash state width $\geqslant 9$. A larger hash state implies a wider AET, and therefore the performance is expected to be worse. Griffin

has fewer degrees of freedom in the size of the hash state, which is a drawback in the construction of STARK F. Furthermore, the first round of the Griffin permutation is different from all other rounds, implying that an additional transition constraint is required in the STARK. Even when the hash states of Griffin and Rescue-Prime are equally wide, Griffin performs slightly worse than Rescue-Prime (see STARKs A and B in the comprehensive Figures A.4 to A.7 in Appendix A; STARKs A and B have a hash state width of four for both Griffin and Rescue-Prime).

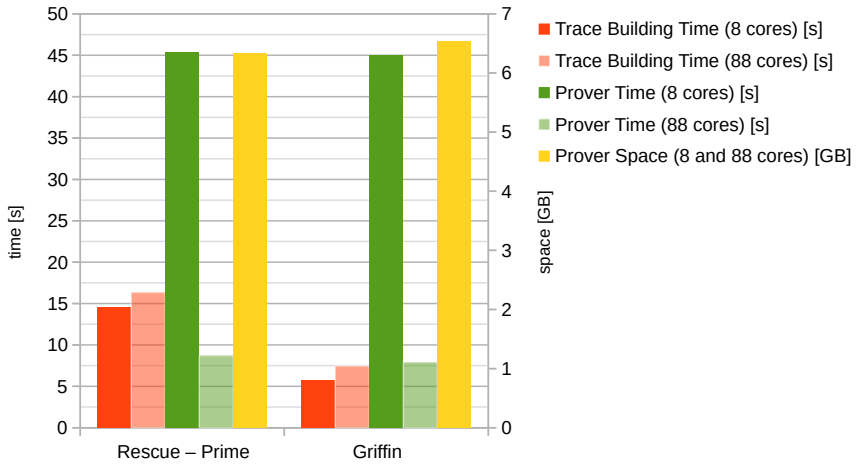

Figure 6.3: Performance comparison of Rescue-Prime and Griffin based on STARK F (opt m4) (62-bit field): 32 Frames (8 cores: Desktop PC, 88 cores: Cluster).

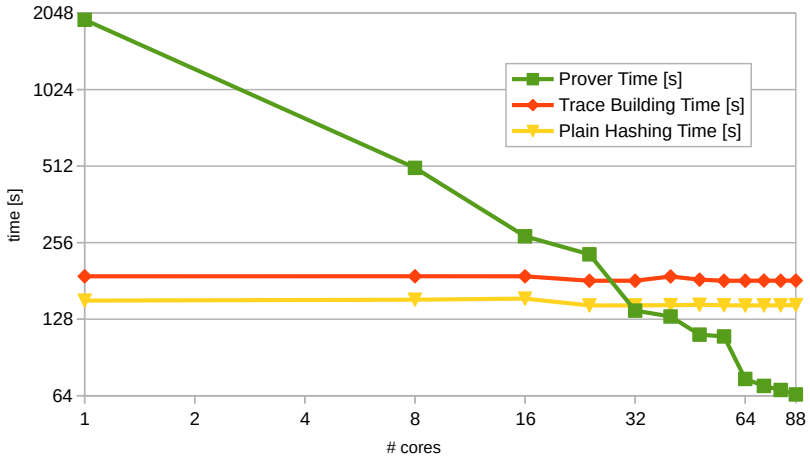

Figure 6.4: Dependency of prover time on the number of cores based on STARK F (opt m8): 256 Frames (62-bit field, Cluster).

6.4 Concurrent Proof Generation

Winterfell allows concurrent trace building, as well as concurrent proof generation. Since Rescue-Prime and Griffin cannot be computed concurrently, we are bound to use concurrent proof generation only. Concurrent proof generation is about 3.7 times faster compared to non-concurrent proof generation on the Desktop PC (8 cores). Figure 6.4 shows the speedup for different numbers of cores. **Doubling the number of cores halves the proving time** up to a certain extent, as long as the length of the AET is large enough.

In the example used in Figure 6.4, the prover time becomes faster than computing the plain hash when using more than 32 cores. Numbers for plain hash computation vary from STARK to STARK, but prover time eventually becomes faster than plain hash computation for any STARK when using enough cores.

The experiment with variable number of cores (see Figure 6.4) was conducted only once. We observe that trace construction time varies by up to seven seconds, even though the time should be constant. Therefore it is also likely that the prover time fluctuates, and the average prover time probably has a stronger linear correlation than visible in Figure 6.4.

6.5 Proof Size and Verifier Complexity

Figure 6.5 shows that the **proof size grows logarithmically with the input size**. In our experiments, the verifier time seems to be constant for STARK F with 7-8 ms on the Cluster. Verifier time should also grow logarithmically with the size of the input, and we can observe this behavior for STARK G, as the large periodic cycles negatively impact verifier performance. STARK G (62-bit field) verifies in 148 ms on the Desktop PC for 16 frames.

6.6 Influence of the Input Length

The impact of the input length is of significant interest, as we aim to fit large datasets into the proof system. The length of the AET and the cycle length of the periodic columns need to be a power of two, which is a requirement of the Winterfell library. Therefore, we analyzed the performance using input lengths of numbers close to powers of two, where we always use whole frames of the video (the input length is a multiple of 110016).

Figure 6.5 shows the correlation of prover time and trace building time for variable input lengths. Prover time, space consumption, and plain hash computation **grow linearly with the input length**. Space consumption is not explicitly plotted in Figure 6.5; the relation may be checked in the raw analysis file located at `/code/performance_evaluation.xlsx` and by using all other provided figures in Appendix A and the tables holding plain evaluation data in Appendix B.

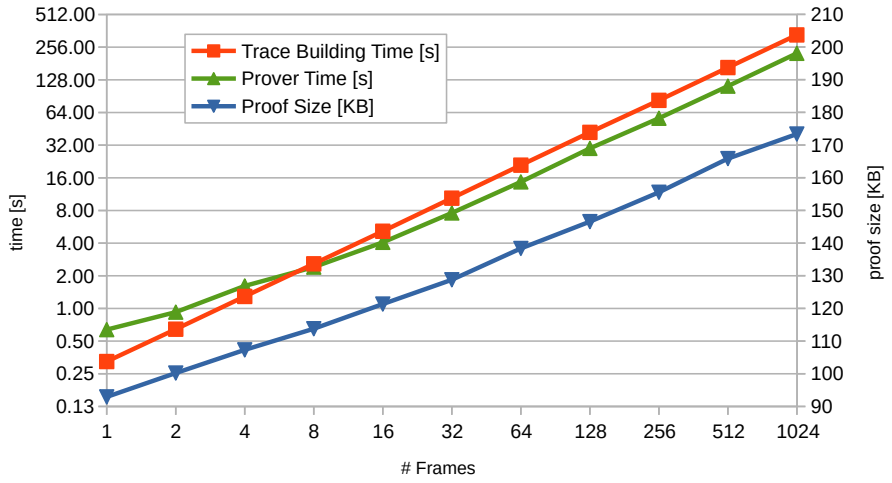

Figure 6.5: Impact of input length on time complexity and proof size for STARK F (opt m8) (Griffin) (62-bit field, Cluster, 88 cores).

6.7 AET Design: Width and Length

Figure 6.6 shows that a **wider trace reduces memory consumption and prover time**, resulting in better overall performance. This effect becomes visible when using large enough input sizes, and we were unable to reproduce this behavior on the Desktop PC (see Figures A.1 and A.2), as insufficient RAM is available to operate on large inputs.

The winner is STARK F (opt m8) in the 62-bit field for Rescue-Prime and Griffin, where the prover space is the smallest among all candidates (see Figure A.3 and Table B.3) for large inputs, arguably the most urgent metric for today's machines. Also, STARK F (opt m8) provides one of the fastest prover times available. Griffin dominates over Rescue-Prime for plain hash performance, so STARK F (opt m8) (Griffin) f62 is the best choice for our problem.

6.8 Performance of the Full Video

As Winterfell does not yet support randomized AIRs with pre-processing (see Chapter 7), we will not provide numbers for the inefficient Fiat-Shamir approach. We also did not do optimization on the ROI variant (STARK G). Therefore we focus on hashing the input and calculating statistics on all pixels (STARK F).

The video provided by the industry partner has a resolution of 382x288 pixels, a framerate of approx. 12 frames per second, a total number of 14794 frames, and the playback time of the video is 18 minutes 28 seconds. The STARK needs to process 1.63 billion pixels.

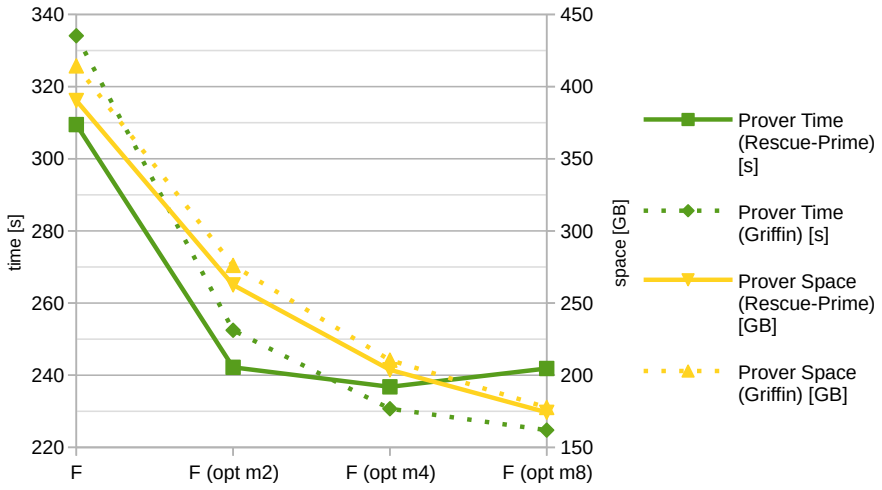

Figure 6.6: Prover time and RAM consumption for different optimization levels of STARK F for Rescue-Prime and Griffin: 1023 Frames (62-bit field, Cluster, 88 cores)

The Cluster at the institute allows us to run STARK F (opt m8) f62 with up to 2048 frames, whereas the following length extension (4096 frames) would exceed the maximum available RAM of 512 GB. Since we know that the prover time, space consumption, and plain hash computation have linear asymptotics, we can extrapolate the metrics for computing the STARK on the whole video.

Section 6.7 states that STARK F (opt m8) (Griffin) f62 is the best available candidate. We expect that feeding the whole video into this STARK would consume the following resources (assuming we use the hardware from the Cluster with more RAM available):

- 3104 MB raw input size (video)
- 2845 GB RAM for the prover
- 64 min plain hashing time
- 89 min trace building time
- 60 min proving time
- 260 KB proof size
- 8 ms verification time

These numbers shall provide a rough estimation of the orders of magnitude for this problem. Recall that RAPs will boost performance for the STARK F scenario as well. Chapter 7 discusses further directions of improvements and future work.

Computing statistics on a region of influence (ROI) (defined via dis-/enabling each pixel) blows up the space and time complexity by a factor of two to five. As discussed in

Chapter 7, we can change the ROI definition to gain better time and space complexity. An optimization example would be to define the ROI on more than one pixel.

Chapter 7

Discussion and Future Work

We showed various approaches to provide solutions to our example problem. There is no definitive answer for the best performing solution, as an essential feature is not available yet within Winterfell: Randomized AIRs with pre-processing. This feature is being worked on while finalizing the present thesis, and the suggestions provided in this chapter allow building a performant STARK as follow-up work.

Section 7.1 discusses all findings, their meaning, and possible improvements to develop a performant STARK. Working with zk-proof systems is laborious as of today, and Section 7.2 expands on improving upon this by creating Compilers that relieve the engineers by eliminating tedious work. Section 7.3 discusses the meaning and impact on the industry.

7.1 Towards a Performant STARK

Native Randomized AIRs with Pre-Processing. The most significant improvement is the support of native RAPs for Winterfell. Grinaway P. asked for implementing RAPs on 20th August 2021 [41] and received positive resonance. Only thoughts and ideas were exchanged in August 2021. In March 2022, researchers from Toposware[1] brought a breath of fresh air to this issue as they started to work on RAPs support independently [37]. Meanwhile, native RAPs support is available for Winterfell in version v0.4.0 (we use v0.3.0) [41, 24].

Native RAPs allow using the Fiat-Shamir heuristic without the need for manually calculating a hash inside the AET. When appropriately implemented, the native RAPs only have a minimal negative impact on overall STARK performance, as the authors of Cairo claim [34].

The use of native RAPs results in a smaller AET, therefore being more efficient. This allows the hash state compression approach (STARKs C/D) to perform better than the uncompressed variants. The result is a smaller AET, therefore being more efficient. Furthermore, the computation of the minimum, maximum, and median become more efficient as the manual hash computation for Fiat-Shamir gets abolished. STARK E may then be based on STARK C instead of STARK B, where we expect a massive performance boost. The plookup checks for the compression approach and the distance checks may be combined into one single plookup check for saving more space in the AET.

[1] `https://toposware.com/`

Improving the ROI approach. STARK G allows defining the ROI on the granularity of one pixel. The ROI gets defined via Periodic Columns, and a large cycle length negatively impacts the performance of the STARK. Defining the ROI on more than one pixel reduces the periodic cycle length and speeds up the STARK. The pixels might need to be reordered before feeding them into the AET, as neighboring pixels in the image need to be neighbors in the trace. The reordering of pixels also influences the definition of how to hash the video.

The design of STARK F (opt mX) nicely combines with an ROI defined over multiple pixels, as X pixels get processed in each step in the AET (and one value of the periodic columns is available for each transition).

Wider and Shorter AETs. The results for STARK F (opt mX) have shown that a broader but shorter trace results in significant savings in space consumption and slight savings on time consumption. A smaller trace might be easier to understand for engineers, but the performance benefits outplay that argument.

Hash Functions and Hash State Alignment. Griffin offers faster plain hash performance compared to Rescue-Prime. As we deal with massive datasets, this attribute is essential for overall performance. The proving step benefits from concurrency, and the proving time can be reduced by using more cores. However, the trace building step (including plain hashing) does not benefit from concurrency. Plain hashing time eventually becomes the bottleneck. Therefore the choice of the hash function in use is crucial.

Prover time is faster than trace construction time for selected examples of our implementations, e.g., STARK F (opt m4) using Rescue-Prime with 1024 frames on the Cluster (88 cores): 8 minutes 40 seconds trace building time and 3 minutes and 57 seconds prover time. The same STARK consumes 3 minutes and 58 seconds for trace building when using Griffin instead of Rescue-Prime.

When using broader hash states (like for STARK F: state width = 12), Griffin requires more rounds than Rescue-Prime to provide the same level of security, making the AET larger and more complex. STARK F provides 128 bits of security when built with Rescue-Prime, and only 105 bits of security when using Griffin. While working on this thesis, Griffin underwent some modifications (as the hash function is under development), requiring more rounds for the same security. When sticking to eight rounds, the security of the modified version of Griffin would be 76 bits (in the 62-bit field for STARK F). The AET needs to be enlarged to achieve adequate security using Griffin. Meanwhile, Griffin was released to the public [39].

The hash state may be stored as a mix of horizontal and vertical alignment to reduce the length of the AET and periodic cycles. Transition constraints may be enforced within one state, as for some variables in the STARK E (opt) construction. An example for this design idea is the storage of the set s in the AET of STARK E (opt).

Tweaking Protocol Parameters. The library Winterfell allows modifying the parameters of the STARK protocol. As the proofs provided by STARKs are succinct, we can modify the parameters so that the verifier needs to perform more computations,

reducing the workload of the prover. Tampering with parameters is future work; we did not spend time on this topic.

Using SSDs for Swapping. The presented constructions are extremely space-demanding. As every machine reaches its limits of RAM capacity at some point, swapping should be considered a viable option. Mukherjee S. showed in his master's thesis that swapping using a fast SSD yields excellent performance. A fast CPU (AMD Ryzen 7 2700X @ 3.7 GHz) using 32 GB RAM and 32 GB swap space outperformed a slower CPU (Intel Xeon Cascade Lake @ 2.5 GHz) with 128 GB RAM for a single-threaded computation consuming approx. 50 GB space [55].

Two CPUs. Building our trace cannot be run in parallel and consumes only a fraction of the space complexity needed for the proving step. Therefore, we can build the trace on a machine having a powerful CPU with a high clock speed and a comparably "small" RAM. CPUs with many cores have a lower clock speed, which is not a problem for concurrent proof generation.

The optimized STARK F constructions consume approximately $\frac{1}{10}$ of prover space for building the trace. This approach would require 3 TB RAM for the prover machine and 300 GB RAM for the trace building machine for the numbers provided in Section 6.8. The powerful CPU of the trace building machine results in a shorter trace building time. Transferring the built trace will cause a massive overhead. Therefore this solution probably only makes sense for a machine holding both CPUs using shared RAM.

Distributed Proof Generation. Another goal and unimplemented feature for Winterfell is distributed proof generation. Multiple machines shall participate in computing one proof. The library's authors talk cryptically about this topic - not much is known about the benefits yet [24, 46]. We hope that this feature allows distributing the required total RAM space over multiple machines, so that each machine only requires to provide a fraction of the total RAM space needed.

7.2 High Level Languages

When creating applications for today's machines, developers may use the Assembly Language or higher languages, such as C, to express their logic. Let us discuss calling a function on the x86 architecture, where the programmer needs to create the new stack frame for holding all the local variables of the function being called (base pointer, return address, ..). This repetitive procedure needs to be done for each function call. The high-level language C takes over creating the stack frame and lets the programmer focus on the relevant aspects - the application logic. We may even use names for variables in C instead of indices in various lists (current stack frame or virtual memory space).

Designing a STARK feels similar to implementing a function call in Assembly Language. The AET's dimensions must be defined at multiple locations when working with the Winterfell library. Variables are columns of the AET, and we need to use indices to interact with them. Modifying a STARK involves updating all indices when adding/deleting variables (columns in the AET). Transition constraints also get defined using indices in a list, similar to variables.

Other examples are basic building blocks, as introduced in Section 4.2.3. The concept of running sums/products is always the same, yet the developer needs to define all constraints individually for each instance of this building block. The analog example in the Assembly/C setting is for/while-loops. Another repetitive building block is a lookup check to restrict the allowed domain for specific variables (the plookup check, introduced in Sections 4.2.3 and 4.3.4). As transition constraints get evaluated between all state transitions, we introduced another building block to perform domain separation for certain constraints using flags in the AET (see Section 4.2.3).

All these components need to be implemented by the programmer of the STARK. Errors occur quickly, and the programmer needs to have a clear concept of all components to be implemented. Mistakes are crucial, as the design of the STARK directly impacts the security of the application.

We observe this issue among colleagues at the institute who work with other proof systems besides STARKs and the Winterfell library. The impression is the same, and we wish for high-level programming languages in the zk-proof system area. Imagine a simple compiler that constructs components, such as running sums/products and multiset checks for the developer in a secure way. Another example would be using variable names and constraint names instead of accessing and defining them via indices. The current situation provides excellent potential for improvements for the libraries' users.

7.3 Meaning for the Industry

Security Concerns. The most important message is that these cryptographic protocols undergo heavy development and are not adequately audited regarding security, as this is a highly active research area. Other cryptographic tools, such as encryption and signatures, are well-established and can be trusted up to a high degree. The authors of Winterfell summarize this aspect on the main GitHub page for describing the library: [24]

WARNING: This is a research project. It has not been audited and may contain bugs and security flaws. This implementation is NOT ready for production use.

Another warning must be issued for the STARKs constructed in the present thesis. Even if the STARKs protocol implemented in Winterfell is secure, the AETs and constraints might be insecure (consider the problem with missing compilers in Section 7.2). Attackers can try to find faulty AETs that satisfy all constraints to create a fake proof. As an example, recall the insecure STARK C, where the prover defines randomness.

This problem is real, and one example is Zcash [17, 9] which aims to provide privacy for transactions. The zero-knowledge proofs used had a flaw, and attackers could create money without getting debunked, as this attack happened in zero-knowledge [20]. The developers of Zcash fixed the vulnerability, and this attack cannot be carried out today [74]. This example shows the negative side of zk-proofs, as the attack's impact remains unknown. An unrelated but similar issue exists for Intel's SGX [19] enclaves which try to provide privacy for executing code. However, if attackers manage to execute their attack inside an enclave, their actions can hardly be backtracked [66].

The platform Ethereum experienced a major incident in 2016. The problem was an erroneous Smart Contract that some attackers exploited. This problem was caused by the platform's users who created the Smart Contract and not by the platform itself. As the affected Smart Contract was a prominent application, the Ethereum Foundation decided to act to undo these attacks. A hard fork was initiated to create two blockchains. The result is two platforms (Ethereum and Ethereum Classic), where Ethereum Classic contains the exploited Smart Contract, and Ethereum had the faulty Smart Contract removed [4, 47, 70].

An engineer implementing a STARK must also consider cryptographic security properties. Implementing STARKs using today's libraries is dangerous when aiming for commercial products. Programmers who had no prior contact with zk-proofs are not suitable for performing the implementation task. Cryptographers with experience in zk-proofs are the appropriate choice, and even experienced experts make crucial mistakes, as highlighted in the previous paragraphs. This is an important takeaway - mistakes in constructing a STARK are not just bugs. These mistakes make the whole construction worthless.

Computational Limitations. We have observed that branching and relative operators are expensive to express within zk-proof systems. Floating-point numbers must be approximated by discrete values. Computations are efficient when using data being close to each other - the further apart two values are, the more expensive the STARK is. An example is the frame size (video resolution) for our problem. Increasing the frame size has a negative impact on the performance.

The best available format for massive datasets is continuous, like the video in our example problem statement. Existing algorithms must be translated into the language of the protocol used, a non-trivial task.

Practical Feasibility. We have shown that solutions to the example problem can be realized on today's machines, even though they are very resource-intensive. The development of a suitable STARK for any problem is a time-consuming task. Constructing a STARK for any problem needs to be done thoroughly by zk-proof experts.

One crucial aspect is educating all involved business partners about the meaning and impact of zk-proof systems. Encryption and digital signatures are well-known cryptographic tools, whereas zk-proofs are a new and mysterious topic. As soon as the properties and benefits are well-understood, various parties might be ready to pay the extra price of the resource-intensiveness for computing proofs.

The main benefit is found in the problem statement, as discussed in Chapter 1. The provider keeps its secrets while releasing certain information the consumer is demanding. Additionally, this information exchange gets assigned with a commitment of time, as introduced in Chapter 3 — the producer cannot tamper with his statement afterward. Both parties are convinced that the facts claimed by the provider are correct, enabling an entirely new level of trust when doing business.

The industry needs to assess whether setting up such proof systems and computing proofs pays off with today's proof systems. We provide insights and estimates on the

performance, and the practical relevance has to be evaluated and discussed by the industry.

Chapter 8

Conclusion

Our goal is to feed massive datasets into zk-proof systems with the motivation arising from the supply chain setting. Consumers want to know what they are buying, while providers want to keep secrets for competitive reasons. Various industry areas struggle with this transparency vs. privacy problem, one of them being the metals industry. Large amounts of data are recorded during the production phase, which may be used to derive qualitative statements about the finished product. We bind the output of the computation to the input by computing the input's hash value alongside the computation inside the zk-proof system (Chapter 3 expands on the proof system design).

Using a video of an additive manufacturing process as an example input, we showed possible approaches for computing statistics alongside the hash value inside zk-proof systems. We used the comprehensive Rust library Winterfell [24] which implements the STARK protocol [10]. We came up with various solutions, allowing us to provide recommendations for building a performant STARK within the scope of enormous input sizes. This work does not provide "the one and only" solution, as we identified many optimization/design paths calling for future work.

We realized that a vital feature is not yet available within Winterfell: Native randomized AIRs with pre-processing (RAPs). However, this feature is being worked on [41] while writing this thesis and was released close before finalizing the present thesis. All presented solutions may be re-implemented to gain performance benefits. Meanwhile, we implemented non-native RAPs by deriving randomness via the Fiat-Shamir heuristic, resulting in a massive overhead. Computing the minimum, maximum, and median relies on RAPs, while computing the standard deviation and the average does not. Furthermore, native RAPs would allow fitting multiple pixels into each element of the hash state, which speeds up the hash computation within the STARK. We expect a significant performance boost for our solutions when native RAPs are available in Winterfell.

Trace building time (including plain hash computation) eventually becomes the bottleneck, as creating proofs is faster for a high number of cores, and today's arithmetization-friendly hash functions are not parallelizable. We compared two hash functions: Rescue-Prime and Griffin, where Griffin is being developed at the institute[1] and was made available to the public when finalizing this thesis [39]. Griffin offers faster plain hash computation than Rescue-Prime and is the better choice, as plain hash computation is the bottleneck. This observation shows that Griffin fulfills the researcher's goals in a

[1] Institute of Applied Information Processing and Communications: `https://www.iaik.tugraz.at/`

practical setting, as they aim to "fix" Rescue-Prime by improving upon the plain hashing performance. We cut down trace building time by more than a factor of two by using Griffin instead of Rescue-Prime for our best implementation.

Another mentionable result is that wider but shorter algebraic execution traces (AETs) perform better than longer and slimmer ones. This behavior was observed for STARKs that solve the same problem where the product of trace width and trace length is (close to) equal (e.g., a STARK with width 4 and length 8 performs better than a STARK with width 2 and length 16). We halved the space (RAM) consumption and reduced the prover time by one-third by switching to a wider and shorter AET.

The main findings are that the current state of the art allows creating proofs on large datasets, like videos. Continuous data formats are well suited as inputs to zk-proof systems. Complex data formats and sophisticated computations strongly negatively impact performance. Designing a clean and straightforward problem is more important than optimizing a STARK of a comparably (unnecessary) complex problem that could be simplified.

Our example video comprises a video resolution of 382x288 pixels, 12 frames per second, and a total playback time of 18 minutes and 30 seconds. Using our institute's machine with 88 cores and 512 GB available RAM (see Chapter 6), we estimate (assuming no RAM limitations) a RAM consumption of 2845 GB, 64 minutes of plain hashing time, 89 minutes of trace building time, 60 minutes of proving time, 260 KB proof size, and 8 milliseconds of verification time. The total time for the prover is 149 minutes (trace building + proving time).

One goal is to come up with real-time proof creation (trace building time + prover time), where real-time refers to being at least as fast as the video playback duration. The prover time may be optimized by increasing the number of cores, and eventually, prover time will be faster than the video playback duration. However, trace building time exceeds the video playback duration, highlighting the importance of performant plain hashing operations. Until such hash functions are discovered, we need to optimize zk-proofs and utilize special hardware for computing the hash. Griffin already significantly improves on plain hashing time, and we expect that proper optimizations will make plain hashing time faster than the video playback duration when using Griffin.

Various approaches for pushing these numbers further down get discussed in Chapter 7. The numbers presented shall provide an idea of the overall magnitude of time and space complexity, and we already know we can do better than the numbers presented above. We leave the creation of a performant STARK as future work, as we provided insights on various approaches, their problems, and recommendations for working with STARKs.

The practical relevance of these estimates needs to be assessed by the industry. Utilizing zk-proofs provides excellent benefits at the cost of a tedious technical setup phase (securely designing the STARK) and expensive proof creation for each product. Further work needs to be done to optimize the design of the proof system and make all surrounding components fit well together. Also, the upcoming features of Winterfell are auspicious and must be considered to improve the solutions further. It is still a long way to go until we end up with a satisfying product — but this path looks more than promising.

Bibliography

[1] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity*. Cryptology ePrint Archive, Report 2016/492. https://ia.cr/2016/492. 2016.

[2] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. *Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols*. Cryptology ePrint Archive, Report 2019/426. https://ia.cr/2019/426. 2019.

[3] Ariel Gabizon. *Multiset checks in PLONK and Plookup*. 2020. URL: https://hackmd.io/@arielg/ByFgSDA7D (visited on 01/05/2022).

[4] Iyke Aru. *Expert: The DAO Was Exploited, Not Hacked, Ethereum Should Do Nothing*. 2016. URL: https://cointelegraph.com/news/expert-the-dao-was-exploited-not-hacked-ethereum-should-do-nothing (visited on 04/24/2022).

[5] Aztec Network. *From AIRs to RAPs - how PLONK-style arithmetization works*. 2021. URL: https://hackmd.io/@aztec-network/plonk-arithmetiization-air (visited on 01/05/2022).

[6] Aztec Network. *Scalable Privacy on Ethereum*. 2022. URL: https://aztec.network/ (visited on 01/05/2022).

[7] Mario Barbara, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lueftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. *Reinforced Concrete: Fast Hash Function for Zero Knowledge Proofs and Verifiable Computation*. Cryptology ePrint Archive, Report 2021/1038. https://ia.cr/2021/1038. 2021.

[8] Mihir Bellare and Phillip Rogaway. "Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols". In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. CCS '93. Fairfax, Virginia, USA: Association for Computing Machinery, 1993, pp. 62–73. ISBN: 0897916298. DOI: 10.1145/168588.168596. URL: https://doi.org/10.1145/168588.168596.

[9] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[10] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Report 2018/046. https://ia.cr/2018/046. 2018.

[11] Eli Ben-Sasson, Lior Goldberg, and David Levit. *STARK Friendly Hash – Survey and Recommendation*. Cryptology ePrint Archive, Report 2020/948. https://ia.cr/2020/948. 2020.

[12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Report 2011/443. https://ia.cr/2011/443. 2011.

[13] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. *Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting*. Cryptology ePrint Archive, Report 2016/263. https://ia.cr/2016/263. 2016.

[14] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[15] Falak Chandna. *What Is A Hash Function In Cryptography?* 2021. URL: https://www.opengrowth.com/article/what-is-a-hash-function-in-cryptography (visited on 04/13/2022).

[16] European Commission. *EU Digital COVID Certificate*. 2022. URL: https://ec.europa.eu/info/live-work-travel-eu/coronavirus-response/safe-covid-19-vaccines-europeans/eu-digital-covid-certificate_en#what-is-the-eu-digital-covid-certificate (visited on 04/13/2022).

[17] Electric Coin Company. *Zcash*. 2022. URL: https://z.cash/ (visited on 04/24/2022).

[18] Patrick Corn and Jimin Khim. *Schwartz-Zippel Lemma*. 2022. URL: https://brilliant.org/wiki/schwartz-zippel-lemma/ (visited on 04/11/2022).

[19] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. https://ia.cr/2016/086. 2016.

[20] National Vulnerability Database. *CVE-2019-7167 Detail*. 2019. URL: https://nvd.nist.gov/vuln/detail/cve-2019-7167 (visited on 04/24/2022).

[21] Facebook. *Rescue-Prime implementation in the Winterfell library*. 2022. URL: https://github.com/novifinancial/Winterfell/blob/main/examples/src/rescue/rescue.rs (visited on 01/06/2022).

[22] Facebook. *Rust crate: Winterfell*. 2022. URL: https://crates.io/crates/Winterfell/0.3.0 (visited on 01/05/2022).

[23] Facebook. *Rust documentation: Crate Winterfell*. 2022. URL: https://docs.rs/Winterfell/0.3.0/Winterfell/ (visited on 01/05/2022).

[24] Facebook. *Winterfell*. 2022. URL: https://github.com/novifinancial/Winterfell (visited on 01/05/2022).

[25] Ferdinand Sauer and Siemen Dhooghe and Alan Szepieniec. *Marvellous (instance generator)*. 2019. URL: https://github.com/KULeuven-COSIC/Marvellous (visited on 01/17/2022).

[26] Amos Fiat and Adi Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194. ISBN: 978-3-540-47721-1.

[27] firmenabc.at. *Dipl.Ing. Heinz Basalka*. 2022. URL: https://www.firmenabc.at/dipl-ing-heinz-basalka_PTJH (visited on 05/05/2022).

[28] firmeninfo.at. *Technologies4you GmbH*. 2022. URL: https://www.firmeninfo.at/firma/technologies4you-gmbh/143407143 (visited on 05/05/2022).

[29] FlippyFlink. *Private key signing.svg*. 2019. URL: https://en.wikipedia.org/wiki/File:Private_key_signing.svg (visited on 04/13/2022).

[30] Rust Foundation. *Rust*. 2021. URL: https://www.rust-lang.org/ (visited on 04/04/2022).

[31] Ariel Gabizon and Zachary J. Williamson. *plookup: A simplified polynomial protocol for lookup tables*. Cryptology ePrint Archive, Report 2020/315. https://ia.cr/2020/315. 2020.

[32] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. https://ia.cr/2019/953. 2019.

[33] RIDDLE&CODE GmbH. *A company providing solutions for blockchain interfaces and digital identities*. 2022. URL: https://www.riddleandcode.com/ (visited on 04/14/2022).

[34] Lior Goldberg, Shahar Papini, and Michael Riabzev. *Cairo – a Turing-complete STARK-friendly CPU architecture*. Cryptology ePrint Archive, Report 2021/1063. https://ia.cr/2021/1063. 2021.

[35] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof Systems". In: *SIAM Journal on Computing* 18.1 (1989), pp. 186–208. DOI: 10.1137/0218012. URL: https://doi.org/10.1137/0218012.

[36] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)". In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*. Ed. by Robert Sedgewick. ACM, 1985, pp. 291–304. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178.

[37] Alonso González and Robin Salen. *Winterfell fork: PR: RAPs Part 1: Interactive Trace Generation*. 2022. URL: https://github.com/Toposware/Winterfell/pull/29/files (visited on 03/29/2022).

[38] David Göthberg. *Public key encryption.svg*. 2006. URL: https://en.wikipedia.org/wiki/File:Public_key_encryption.svg (visited on 04/13/2022).

[39] Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. *A New Feistel Approach Meets Fluid-SPN: Griffin for Zero-Knowledge Applications*. Cryptology ePrint Archive, Report 2022/403. https://ia.cr/2022/403. 2022.

[40] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems*. Cryptology ePrint Archive, Report 2019/458. https://ia.cr/2019/458. 2019.

[41] Patrick Grinaway and Irakliy Khaburzaniya. *Winterfell library: issue 54: Supporting RAPs*. 2021. URL: https://github.com/novifinancial/Winterfell/issues/54 (visited on 03/07/2022).

[42] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive, Report 2016/260. https://ia.cr/2016/260. 2016.

[43] Guillaume Drevon. *Practical Applications of Zero-Knowledge Proofs*. 2019. URL: https://sikoba.com/docs/zklux1/ZKLux1_Drevon_PracticalApps.pdf (visited on 01/05/2022).

[44] Prof. Nick Harvey. *Lecture Notes: CPSC 536N: Randomized Algorithms (University of British Columbia)*. 2011. URL: https://www.cs.ubc.ca/~nickhar/W12/Lecture9Notes.pdf (visited on 04/11/2022).

[45] https://crypto.stackexchange.com/. *what is the difference between proofs and arguments of knowledge?* 2016. URL: https://crypto.stackexchange.com/questions/34757/what-is-the-difference-between-proofs-and-arguments-of-knowledge (visited on 04/13/2022).

[46] Irakliy Khaburzaniya, Kostas Chalkias, Kevin Lewi, and Harjasleen Malvai. *Open sourcing Winterfell: A STARK prover and verifier*. 2021. URL: https://engineering.fb.com/2021/08/04/open-source/Winterfell/ (visited on 03/30/2022).

[47] Oleksii Konashevych. *Takeaways: 5 years after The DAO crisis and Ethereum hard fork*. 2021. URL: https://cointelegraph.com/news/takeaways-5-years-after-the-dao-crisis-and-ethereum-hard-fork (visited on 04/24/2022).

[48] Maksym Petkus. *Talk at Consensus 2018: Zero-Knowledge Supply Chain Blockchain, MediLedger, Responsible Gold*. 2018. URL: https://www.youtube.com/watch?v=mNfeO0F_zLg (visited on 01/05/2022).

[49] Encyclopedia of Mathematics. *Algebraic equation*. 2020. URL: http://encyclopediaofmath.org/index.php?title=Algebraic_equation&oldid=45061 (visited on 04/11/2022).

[50] Encyclopedia of Mathematics. *Galois field*. 2014. URL: http://encyclopediaofmath.org/index.php?title=Galois_field&oldid=34238 (visited on 04/11/2022).

[51] Encyclopedia of Mathematics. *Polynomial*. 2015. URL: http://encyclopediaofmath.org/index.php?title=Polynomial&oldid=36519 (visited on 04/11/2022).

[52] Asher Mattison and Brennan Coogan. *Zero-Knowledge Proofs: STARKs vs SNARKs*. 2021. URL: https://consensys.net/blog/blockchain-explained/zero-knowledge-proofs-starks-vs-snarks/ (visited on 04/04/2022).

[53] Jose Meseguer-Valdenebro, Eusebio Martínez-Conesa, and Antonio Portolés. "Numerical-experimental validation of the welding thermal cycle carried out with the MIG welding process on a 6063-T5 aluminium tubular profile". In: *Thermal Science* 23 (Feb. 2019), pp. 3639–3650. DOI: 10.2298/TSCI181215030M.

[54] Metastate Team. *On PLONK and plookup*. 2020. URL: https://research.metastate.dev/on-plonk-and-plookup/ (visited on 01/05/2022).

[55] Shibam Mukherjee. *Master's Thesis: Privacy Preserving Recommender Systems*. Institute of Applied Information Processing and Communications, Graz University of Technology. 2022.

[56] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 04/12/2022).

[57] Nicholas Fett and Kevin Marek and Maxime Suard. *ZK Supply - ZKPs for the Supply Chain Industry*. 2019. URL: https://devpost.com/software/zk-blow-zero-knowledge-proofs-for-the-cocaine-industry (visited on 01/05/2022).

[58] NTT DATA Australia. *Technology Trend "The impact of Blockchain on supply chain management"*. 2019. URL: https://www.nttdata.com/au/en/foresight/2019/september/technology-trend-the-impact-of-blockchain-on-supply-chain-management (visited on 01/05/2022).

[59] offshift.io. *Bulletproofs, zkSNARKs, and zkSTARKs Walk into a Blockchain*. 2021. URL: https://offshift.io/public/blog/2021-11-24-bulletproofs-zksnarks-zkstarks/ (visited on 04/12/2022).

[60] Dick Olsson. *Two balls and the colour-blind friend*. 2018. URL: https://dickolsson.com/two-balls-and-the-colour-blind-friend/ (visited on 04/13/2022).

[61] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. *Pinocchio: Nearly Practical Verifiable Computation*. Cryptology ePrint Archive, Report 2013/279. https://ia.cr/2013/279. 2013.

[62] Ping An Technology (Shenzhen) Co., Ltd. *Patent: Zero-Knowledge Proof-Based Supply Chain Data Management Method and Apparatus*. 2020. URL: https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2020224092&tab=PCTBIBLIO (visited on 01/05/2022).

[63] Anup Rao. *Lecture Notes: Lecture 15: The Schwartz-Zippel Lemma and the Determinant (University of Washington)*. 2018. URL: https://homes.cs.washington.edu/~anuprao/pubs/431/lecture15.pdf (visited on 04/11/2022).

[64] S1Seven. *Material Identity*. 2022. URL: https://www.s1seven.com/ (visited on 05/05/2022).

[65] Shubham Sahai, Nitin Singh, and Pankaj Dayama. "Enabling Privacy and Traceability in Supply Chains using Blockchain and Zero Knowledge Proofs". In: *2020 IEEE International Conference on Blockchain (Blockchain)*. 2020, pp. 134–143. DOI: 10.1109/Blockchain50366.2020.00024.

[66] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. 2017. URL: https://arxiv.org/abs/1702.08719.

[67] skuchain. *Currency Agnostic Blockchain For Global Trade*. 2022. URL: https://www.skuchain.com/ (visited on 01/05/2022).

[68] Nigel P. Smart. *Cryptography Made Simple*. 1st. Springer Publishing Company, Incorporated, 2015. ISBN: 978-3-319-21935-6. DOI: 10.1007/978-3-319-21936-3.

[69] Leopold Th. Spanring. *top agrar Österreich: Volksbegehren zur Herkunftskennzeichnung eingeleitet*. 2021. URL: https://www.topagrar.at/management-und-politik/news/volksbegehren-zur-herkunftskennzeichnung-gestartet-12643462.html (visited on 03/31/2022).

[70] Cryptopedia Staff. *What Was The DAO?* 2022. URL: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao#section-the-response-to-the-dao-hack (visited on 04/24/2022).

[71] National Institute of Standards and Technology (NIST). *SHA-2 Standard*. 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (visited on 02/15/2022).

[72] StarkWare. *ethSTARK Documentation*. Cryptology ePrint Archive, Report 2021/582. https://ia.cr/2021/582. 2021.

[73] Susanne Somerville. *Supply Chains Don't Want Total Data Transparency. Here's What Will Work Instead*. 2018. URL: https://blog.chronicled.com/supply-chains-dont-want-total-data-transparency-here-s-what-will-work-instead-43e21c4106e (visited on 01/05/2022).

[74] Josh Swihart, Benjamin Winston, and Sean Bowe. *Zcash Counterfeiting Vulnerability Successfully Remediated*. 2019. URL: https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/ (visited on 04/24/2022).

[75] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. *Rescue-Prime: a Standard Specification (SoK)*. Cryptology ePrint Archive, Report 2020/1143. https://ia.cr/2020/1143. 2020.

[76] Karl Tröger. *Blockchain in Deutschland - Was kommt nach dem Hype?* 2021. URL: https://www.psi.de/de/blog/psi-blog/post/blockchain-in-deutschland-was-kommt-nach-dem-hype/ (visited on 04/14/2022).

[77] Takio Uesugi, Yoshinobu Shijo, and Masayuki Murata. *Short Paper: Design and Evaluation of Privacy-preserved Supply Chain System based on Public Blockchain*. 2020. arXiv: 2004.07606 [cs.CR].

[78] Beuth Verlag. *DIN EN 10373:2021-09: Determination of the physical and mechanical properties of steels using models*. 2021. URL: https://dx.doi.org/10.31030/3237422 (visited on 03/31/2022).

[79] Vitalik Buterin. *Exploring Elliptic Curve Pairings*. 2017. URL: https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627 (visited on 01/05/2022).

[80] Vitalik Buterin. *Quadratic Arithmetic Programs: from Zero to Hero*. 2016. URL: https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649 (visited on 01/05/2022).

[81] Vitalik Buterin. *STARKs, Part 3: Into the Weeds*. 2018. URL: https://vitalik.ca/general/2018/07/21/starks_part_3.html (visited on 01/05/2022).

[82] Vitalik Buterin. *STARKs, Part I: Proofs with Polynomials*. 2017. URL: https://vitalik.ca/general/2017/11/09/starks_part_1.html (visited on 01/05/2022).

[83] Vitalik Buterin. *STARKs, Part II: Thank Goodness It's FRI-day*. 2017. URL: https://vitalik.ca/general/2017/11/22/starks_part_2.html (visited on 01/05/2022).

[84] Vitalik Buterin. *Zk-SNARKs: Under the Hood*. 2017. URL: https://medium.com/@VitalikButerin/zk-snarks-under-the-hood-b33151a013f6 (visited on 01/05/2022).

[85] Florian Wacker. *Master's Thesis: Use of Blockchain Technology for Tracing CO2 Emission along the Steel Supply Chain*. Vienna University of Economics and Business. 2020.

[86] Dennis Wackerly, William Mendenhall, and Richard L. Scheaffer. *Mathematical Statistics with Applications*. eng. Cengage Learning, 2014. ISBN: 9781111798789.

[87] Wikipedia. *Modular arithmetic*. 2022. URL: https://en.wikipedia.org/wiki/Modular_arithmetic (visited on 04/11/2022).

[88] Wikipedia. *Notary*. 2018. URL: https://en.wikipedia.org/wiki/Notary (visited on 02/15/2022).

[89] Wikipedia. *Polynom*. 2009. URL: https://de.wikipedia.org/wiki/Polynom (visited on 04/11/2022).

Appendices

Appendix A
Performance Charts

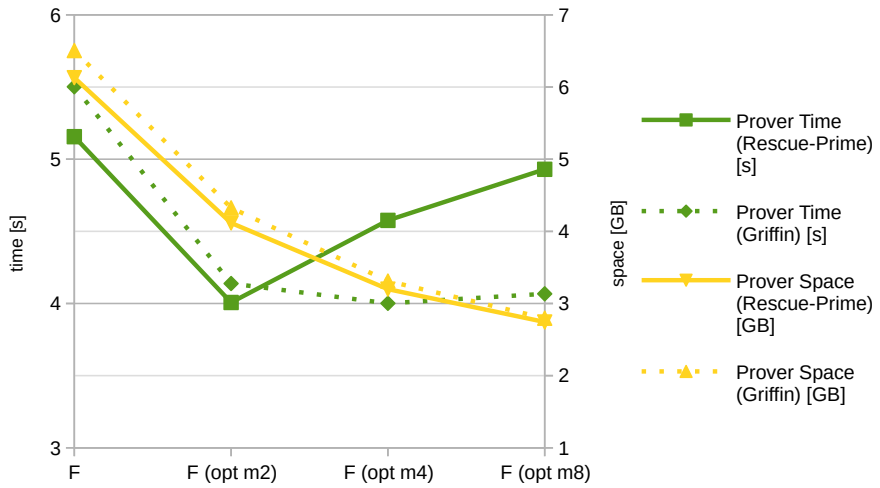

Figure A.1: Prover time and RAM consumption for different optimization levels of STARK F for Rescue-Prime and Griffin: 16 Frames (62-bit field, Cluster, 88 cores).

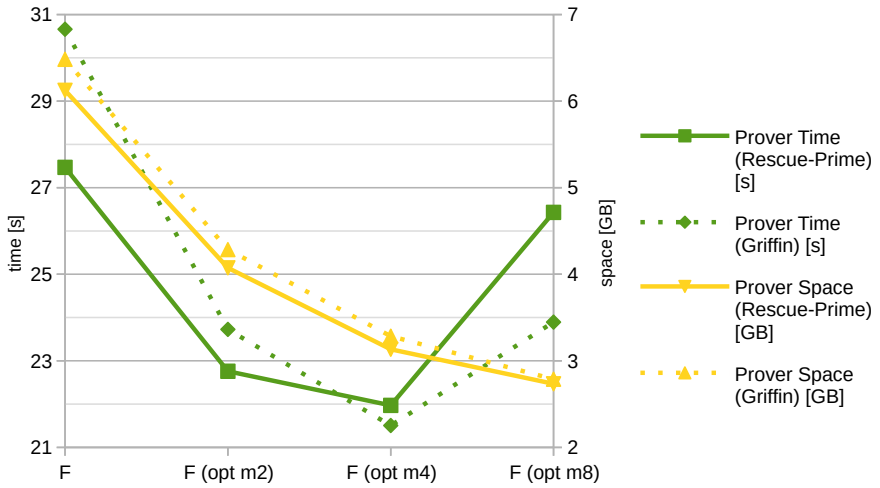

Figure A.2: Prover time and RAM consumption for different optimization levels of STARK F for Rescue-Prime and Griffin: 16 Frames (62-bit field, Desktop PC, 8 cores).

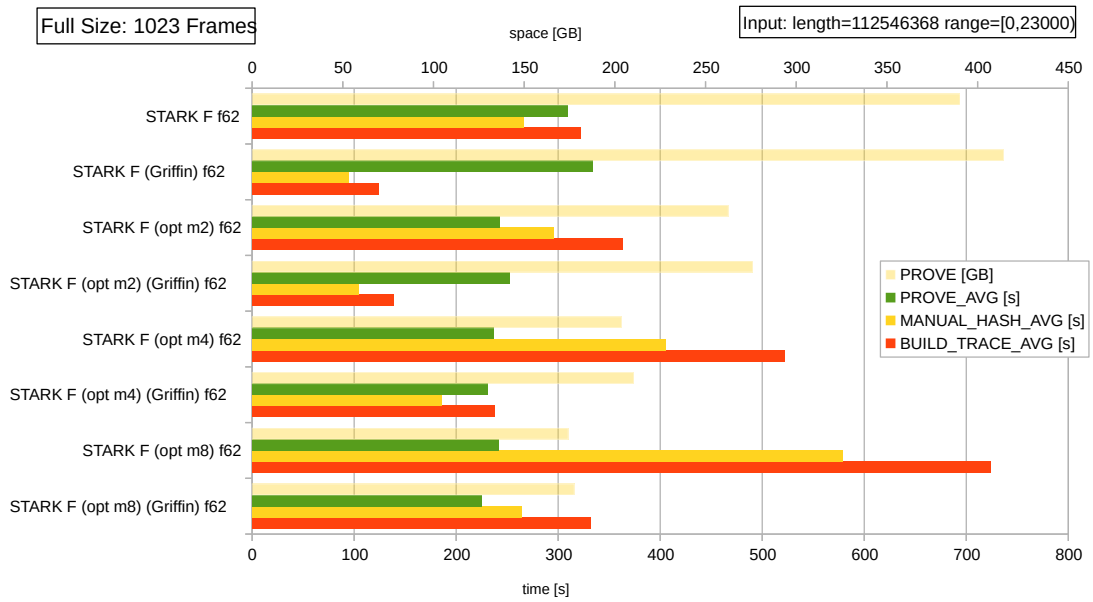

Figure A.3: Prover time, trace building time, and RAM consumption for different optimization levels of STARK F for Rescue-Prime and Griffin (62-bit field, Cluster, 88 cores).

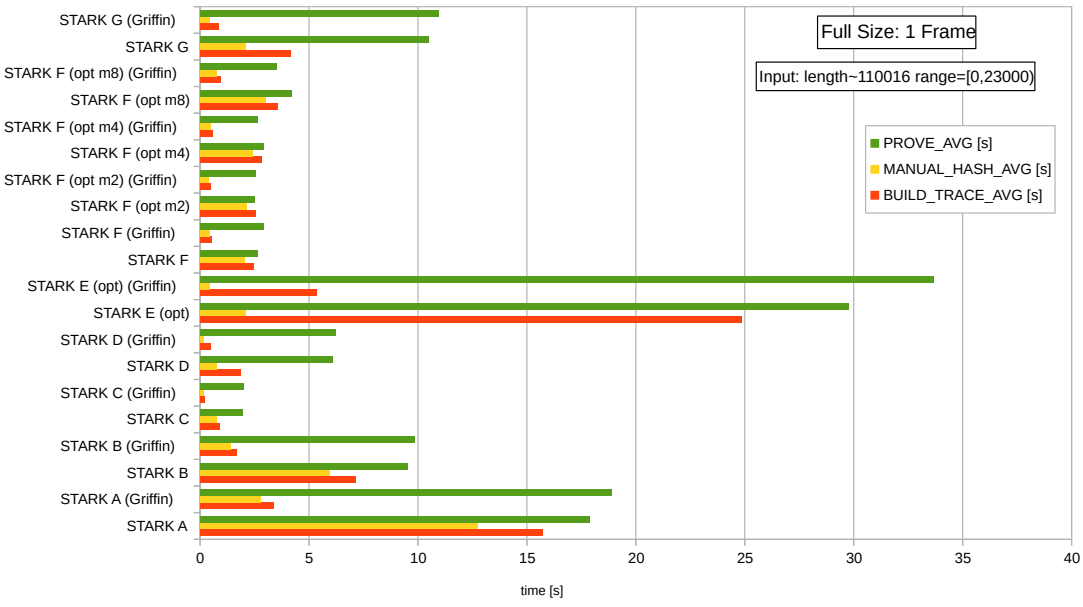

Figure A.4: Extended all STARKs time complexity comparison: Full Size, 1 Frame (128-bit field, Desktop PC, 8 cores).

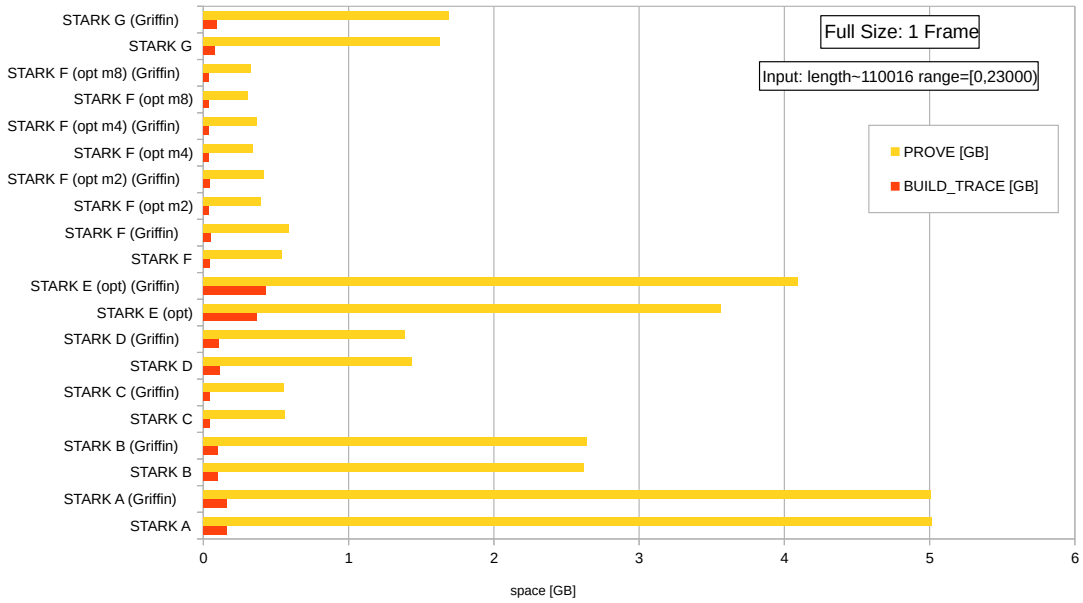

Figure A.5: Extended all STARKs space complexity comparison: Full Size, 1 Frame (128-bit field, Desktop PC, 8 cores).

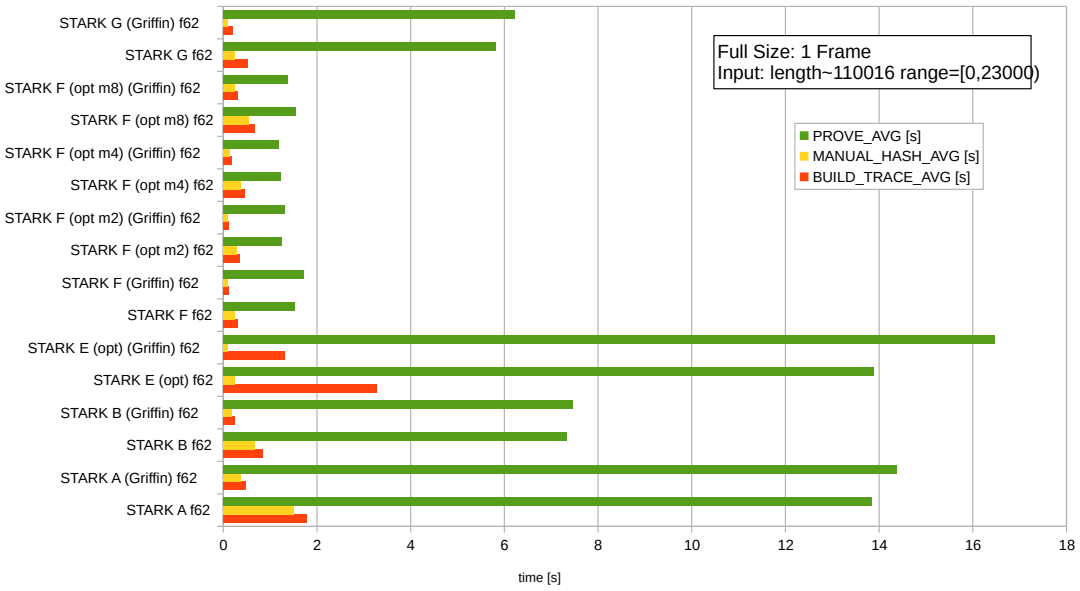

Figure A.6: Extended all STARKs time complexity comparison: Full Size, 1 Frame (62-bit field, Desktop PC, 8 cores).

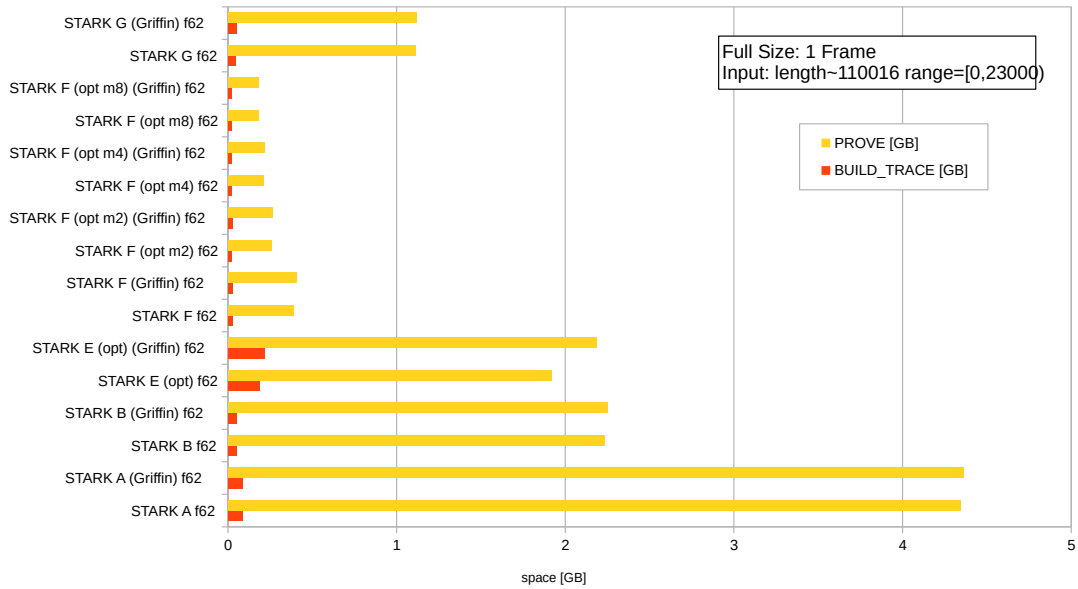

Figure A.7: Extended all STARKs space complexity comparison: Full Size, 1 Frame (62-bit field, Desktop PC, 8 cores).

Appendix B

Performance Tables

Table B.1: Performance numbers for all STARKs: Quarter Size (128-bit field, Desktop PC).

Variant	Hash	Trace Building Time [s]	Plain Hashing Time [s]	Prover Time [s]	Verifier Time [s]	Trace Building RAM [GB]	Prover RAM [GB]
A	Resc.	3.94	3.82	4.30	0.001	0.044	1.20
	Griff.	0.84	0.82	4.54	0.001	0.044	1.26
B	Resc.	1.79	1.77	2.27	0.001	0.029	0.64
	Griff.	0.43	0.41	2.33	0.001	0.029	0.67
C	Resc.	0.23	0.22	0.44	0.001	0.014	0.12
	Griff.	0.06	0.05	0.45	0.001	0.008	0.11
D	Resc.	0.47	0.22	1.39	0.001	0.033	0.33
	Griff.	0.13	0.05	1.41	0.001	0.017	0.35
E	Resc.	30.05	0.61	78.17	0.013	0.451	9.18
	Griff.	6.46	0.12	86.83	0.014	0.522	10.37
E (opt)	Resc.	6.26	0.62	7.17	0.020	0.097	0.88
	Griff.	1.35	0.12	8.10	0.021	0.112	1.03
F	Resc.	0.62	0.61	0.61	0.001	0.015	0.14
	Griff.	0.13	0.12	0.68	0.001	0.016	0.13
F (opt m2)	Resc.	0.64	0.63	0.60	0.001	0.014	0.10
	Griff.	0.12	0.11	0.60	0.002	0.015	0.11
F (opt m4)	Resc.	0.71	0.71	0.71	0.002	0.013	0.09
	Griff.	0.15	0.14	0.66	0.002	0.014	0.09
F (opt m8)	Resc.	0.90	0.89	1.05	0.004	0.013	0.08
	Griff.	0.23	0.22	0.88	0.004	0.013	0.08
G	Resc.	1.24	0.62	2.45	0.060	0.024	0.41
	Griff.	0.26	0.12	2.60	0.048	0.026	0.43

Table B.2: Performance numbers for all STARKs: Quarter Size (62-bit field, Desktop PC).

Variant	Hash	Trace Building Time [s]	Plain Hashing Time [s]	Prover Time [s]	Verifier Time [s]	Trace Building RAM [GB]	Prover RAM [GB]
A	Resc.	0.45	0.45	3.34	0.001	0.024	1.10
	Griff.	0.12	0.11	3.53	0.001	0.024	1.08
B	Resc.	0.21	0.20	1.75	0.001	0.017	0.55
	Griff.	0.06	0.06	1.80	0.001	0.016	0.55
E	Resc.	4.02	0.08	39.70	0.009	0.229	5.38
	Griff.	1.58	0.03	44.72	0.008	0.264	5.97
E (opt)	Resc.	0.84	0.08	3.36	0.010	0.052	0.47
	Griff.	0.33	0.03	3.88	0.012	0.059	0.55
F	Resc.	0.08	0.08	0.35	0.001	0.010	0.10
	Griff.	0.03	0.03	0.39	0.001	0.011	0.11
F (opt m2)	Resc.	0.09	0.09	0.30	0.002	0.010	0.10
	Griff.	0.03	0.03	0.31	0.002	0.011	0.11
F (opt m4)	Resc.	0.12	0.11	0.30	0.003	0.009	0.05
	Griff.	0.05	0.04	0.30	0.003	0.009	0.05
F (opt m8)	Resc.	0.16	0.16	0.39	0.006	0.009	0.05
	Griff.	0.07	0.07	0.35	0.006	0.009	0.05
G	Resc.	0.16	0.08	1.38	0.037	0.014	0.26
	Griff.	0.06	0.03	1.46	0.029	0.016	0.28

Table B.3: Performance numbers for all STARK F variations: Full Size, 1023 Frames (62-bit field, Cluster).

Variant	Hash	Trace Building Time [s]	Plain Hashing Time [s]	Prover Time [s]	Verifier Time [s]	Trace Building RAM [GB]	Prover RAM [GB]
F	Resc.	322.0	265.9	309.5	0.003	?	390.36
	Griff.	124.0	95.0	334.1	0.003	?	414.36
F (opt m2)	Resc.	363.6	295.8	242.2	0.003	?	262.60
	Griff.	138.7	104.3	252.5	0.004	?	276.10
F (opt m4)	Resc.	521.5	405.2	236.8	0.005	?	203.73
	Griff.	238.1	185.5	230.7	0.005	?	210.48
F (opt m8)	Resc.	723.5	578.7	241.9	0.008	?	174.29
	Griff.	331.9	264.0	224.8	0.008	?	177.66