Air-FRI: Acceleration of the FRI Protocol on the GPU for Low-Degree Polynomial Testing in zk-SNARK Applications

by

Tanmayi Jandhyala

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical & Computer Engineering

Waterloo, Ontario, Canada, 2024

© Tanmayi Jandhyala 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

By employing secure, interoperable, and streamlined data-sharing capabilities, companies can leverage the blockchain technology to enhance data accessibility and collaboration while maintaining the security of their data. Notably, blockchain possesses potential to also support privacy-preservation by integrating zero-knowledge proofs into its design. Yet, the efficiency and scalability of the underlying cryptographic protocols that enable this remains a challenge. There evidently exists a gap between theoretical advances in cryptographic algorithms and their practical, performance-efficient implementations in software.

Zero-Knowledge Succinct Non-Interactive ARguments of Knowledge (zk-SNARKs) offer a promising approach to large-scale privacy assurance. However, with significantly large constraints and complex proofs, realizing zk-SNARK-based protocols in practice can be challenging. This thesis addresses this gap by enhancing the efficiency of the Fast Reed-Solomon Interactive Oracle Proof of Proximity (FRI) protocol- a core component in post-quantum zk-SNARKs that reduces computational complexity in systems with substantial mathematical instances. Existing schemes that implement the FRI protocol entail significant computational times due to large proof sizes. Through an optimized, GPU-based implementation of FRI, this work aims to accelerate performance, essential for privacy-preserving, decentralized, and post-quantum secure systems.

This thesis presents Air-FRI, a novel GPU-enabled software implementation of the FRI protocol, which includes a parallelized computation of Reed-Solomon codewords, precomputation of time-intensive finite field operations, non-interactiveness, and the application of a unified Merkle tree commitment to authenticate the entire proof. Together, the implemented optimizations yield a software-oriented solution that significantly reduces prover and verifier times while minimizing proof size, addressing both scalability and efficiency challenges in zk-SNARK-based algorithms. Performance evaluations across security levels confirm the implementation's practicality and high throughput, establishing it as a valuable contribution within the research space.

Our results show a stellar 93.3% improvement on an average in the speed of the protocol on the GPU as compared to its implementation on a CPU for the same parameters, including the execution time of all of its sub-phases: the commit phase, the query phase, the co-linearity and authentication checks to ensure correctness of the proof.

This work expands on the use of FRI in zk-SNARK algorithms to strengthen the privacy in decentralized systems. It establishes a foundation for advancing privacy-preserving post-quantum cryptography, supporting the development of secure, transparent, and privacy-centered digital infrastructure for the future.

Acknowledgements

I would like to thank my wonderful supervisor, Professor Guang Gong, for the invaluable mentorship and support that you have provided me with in the past year. Every day, I am inspired to learn to ride the little waves in the ocean of impact that you have created in the space of Cryptography and Security Engineering. Your belief in me has led me to explore opportunities that I had never dreamed of pursuing. Above all, I am deeply grateful for your empathy, which has been a source of strength and encouragement, reminding me to persevere and grow even in the most challenging moments.

I am also very thankful to Professor Anwar Hasan and Professor Hiren Patel for taking the time out of your incredibly busy schedules to review my thesis and provide me with your valuable remarks.

I also extend my sincere thanks to all the inspiring folks at the lab for your support and advice along the way.

A very special and heartfelt thank you to my parents, whom I owe everything to. Thank you for teaching me that education is the true foundation for life. Sending me across the world to pursue my dreams must have taken courage and selflessness, and your belief has kept me strong and true to my roots. To my beloved grandparents—my late grandfather, whose strength still guides me, and my grandmother, who inspires me—thank you for instilling resilience and love. To the rest of my family, your love and support have been my anchor through every challenge, and I couldn't have come this far without you by my side.

I am incredibly grateful to my dear friends back home, scattered across, and the very special ones that I have made here. Your constant love and support has brought me so much encouragement and joy in living life and in pursuing research- something that I had not pictured myself doing and enjoying. To you, I say: *To my rock and my light, the purpose of my doing, is for you to see it happen.*

I would also like to extend my deep gratitude to the Cybersecurity and Privacy Institute at the University of Waterloo for their GPU resources that were of incredible use in my thesis work.

Part of the work in this thesis is supported by MITACS and BTQ project grant, whom I sincerely thank. I also thank the University of Waterloo for giving me a transformative experience in the past 2 years.

Dedication

This thesis is dedicated to my loving parents.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Overview 1

1.2 Contributions 4

1.3 Thesis Outline 5

2 Background and Related Works 6

2.1 Some Context on Zero Knowledge Proofs 6

2.2 Practical zk-SNARKs 8

2.3 Merkle Trees and Hashing 11

2.4 The FRI-IOP Protocol 12

2.4.1 Reed-Solomon Codes and the RS Proximity Problem 12
2.4.2 RS Proximity Testing 13
2.4.3 The Fast RS IOPP (FRI) 14
2.5 Implementing the FRI Protocol 15
2.5.1 Notations [Gon23] 15
2.5.2 Phases of the FRI protocol [Bad+24; Gon24a] 16
2.5.3 A Toy Example Showcasing Bases and Codeword Computations in Each Round 18
2.6 Software Implementations of FRI 23
2.6.1 Existing Software for FRI 23
2.6.2 Programming the FRI Protocol in C (on the CPU) and CUDA-C (on the GPU) 24

3 Optimizations in Air-FRI to Improve Overall Performance of the Protocol 26

3.1 Pre-computing the Inverses of Domain Bases for Next Codeword (Line) Computation 27
3.2 Repeated Hashing of Sampled Element for Each Round to Make Protocol Non-Interactive 28
3.3 Parallel Computation of Codeword Elements on the GPU 29
3.4 Implementing an Integrated FRI Merkle Tree 30
3.4.1 RS Tree Structure 30
3.4.2 Merkle Tree Commitment of RS Codewords 31
3.4.3 Verification and Authentication Paths 33

4 Implementation Methodology of Air-FRI 36

4.1 Architectural Overview 36
4.1.1 Components 36
4.1.2 Implementation Assumptions 36
4.1.3 Use of Open Source Libraries 38

4.1.4 Extending the Basis Elements . . . 39
4.1.5 Borrowing Preon's [Che+23] Parameters . . . 40
4.2 Proof Management . . . 40
4.2.1 Merkle Tree Commitment and Opening . . . 41
4.2.2 Handling of Authentication Paths . . . 42
4.2.3 Optimization Considerations . . . 42
4.2.4 Memory Management on the GPU . . . 43
4.2.5 Proof Stream and Verification . . . 45

5 Performance of Air-FRI **47**
5.1 Test Plan . . . 47
5.1.1 Unit Tests . . . 47
5.1.2 Performance Test Design . . . 50
5.1.3 Test Environment . . . 51
5.1.4 Test Metrics . . . 51
5.2 Results . . . 52
5.2.1 Performance of the Code Run with L3 Parameters . . . 52
5.2.2 Performance on L5 Parameters . . . 53
5.3 Performance of Current FRI Implementations . . . 55

6 Conclusions and Future Work **58**
6.1 Summary of Thesis . . . 58
6.2 Limitations . . . 59
6.3 Future Work . . . 60
6.3.1 Further Reduction of the Proof Size . . . 60
6.3.2 Integrating with a Faster Implementation of the FFT Algorithm . . . 60
6.3.3 Implementing Zero-Knowledge with this FRI Protocol . . . 60
6.3.4 Encryption of Serialized Data . . . 61

6.3.5 Comparison of Energy Consumption between CPU and GPU Implementations of the Protocol 61
6.3.6 Memory Profiling of the GPU Code 61

References **62**

List of Figures

3.1 Computation of the next codeword element $f_{k+1}(L_{k+1}[i])$ from two previous codeword elements. 30

3.2 Parallel Computations of Codeword Elements and Merkle Tree Hashes . . 31

3.3 Integrated FRI Merkle Tree 32

3.4 Example Authentication Path (in purple) for codeword f_{04} (in green) . . . 34

3.5 Example Verifier's computations (in yellow) leading up to the root 35

4.1 Software Components of Our Implementation 37

5.1 FRI Degree Testing Time on the GPU with L3 Parameters 54

5.2 CPU vs GPU Execution Times for FRI Degree Testing for L3 Parameters . 54

List of Tables

2.1 Values of $q_i(x)$ and basis elements for each domain. 20

2.2 Elements of $\mathcal{L}_i$ for each iteration. 20

2.3 Computed values of $f_1(x)$ based on f_0 evaluations over domain L_0. 22

5.1 Parameters for L3 FRI Protocol 52

5.2 Performance Comparison of the FRI protocol on CPU vs GPU for L3 Parameters 53

5.3 Minimum, Maximum, and Average Times for Prover and Verifier for degree 2^{12} over 10 iterations 53

5.4 Speedup Factor (%) of GPU over CPU for Various Degrees for L3 Parameters 55

5.5 Parameters for L5 FRI Protocol 55

5.6 Minimum, Maximum, and Average Times for Prover and Verifier for degree 2^{16} over 10 iterations on the GPU 56

5.7 Performance Comparison of Existing FRI Implementations on the CPU . . 56

5.8 Merkle Tree Test Summary for Libiop Implementations 56

5.9 Preon's Prover and Verifier Times for Testing of Degree 2^{12} over 50 Iterations on the CPU 57

Chapter 1

Introduction

1.1 Overview

In a society that is as digitally-driven as ours, data privacy and trust are complexly intertwined. Individuals expect that their personal information, including sensitive data such as that of medical, biometric, or financial, will be securely stored and is accessible only to authorized entities that users consent to. In reality, this assumption is often exploited by institutions to obscure misconduct, conceal errors, or mislead the public [Ben+18d], or it could be so that technological vulnerabilities in 'trusted' third parties do most of the damage. Such abuses not only harm citizens but also erode the very foundation of trust that centralized systems rely on. The blockchain technology [NB08], popularly records transaction data in a decentralised, distributed, persistent fashion. Its application in various financial services [PP15; FT15], smart contracts [Kos+16], public services [ANV13], and Internet of Things [ZW15] shifts its responsibility from being a 'cool' technology to one that needs strong implementation of infrastructural and cryptographic primitives within to make it usable. The field of systems security and technologies that enhance privacy is challenged with the fact that virtually all databases operate in adversarial environments [ZXL19]. A blockchain-based distributed public ledger, where all participants possess identical data, crucial for applications like online banking, also presents significant privacy concerns. When the lens is diverted to this, it becomes clear that privacy-preservation is not a natural part of traditional blockchain systems, pushing businesses willing to employ such an infrastructure into addressing it if they need to gain the trust of their users.

An immediate object of discussion would be of the privacy issues with de-anonymization of data published on a blockchain. Every transaction recorded on a public ledger is vis-

ible to all participants. Once a transaction is publicly verified, it is stored in an open database. Network-based de-anonymization attacks in the past [BKP14; KKM14; BT19] were successfully able to link transactions with real-world identifiers (such as IP addresses). Further, when users interact with the blockchain, they often query specific transactions, check account balances, or request information about particular blocks. Each of these actions can reveal which parts of the blockchain a user is interested in. Adversaries observing these interactions can infer the user's activities or link them to specific transactions. For example, if a user regularly checks the balance of a particular wallet, it could indicate ownership or a financial relationship with that wallet [ATM21].

Researchers in the field were quickly able to come up with solutions to address these privacy challenges, which include promising cryptographic solutions. Although proven to not fully prevent user linkability [Kap+18], ZCash [Hop+16](first inroduced as ZeroCash [Ben+14a]) was amongst the first companies to employ ***Zero-knowledge Proofs*** in their blockchain architecture to preserve user anonymity. Zero-knowledge Proofs, first introduced by [GMR89] enable a party to prove possession of information to another party without revealing the contents of the information itself. Initially, these proofs were interactive [Bab85; GMW86], but it was soon realised to have a non-interactive ways of proof systems to be written in order for them to be useful for adoption into any kind of modern cryptographic systems (e.g., in digital signatures and public key cryptography). Non-interactive proofs would allow for a verifier in the proof system to validate a single proof submitted by a prover at a later time [BFM88; BG90].

Non-interactive Proofs of knowledge (NIZK) would then be loosely extended into Succinct Non-interactive Arguments (SNARGs) [Mic94], which would consist of very short proofs even for large computations to be adopted for use cases such as verifiable computation. Succinct Non-Interactive Arguments of Knowledge (SNARKs) would be about the same, but also about the arguments of knowledge of information that the proof is generated for [Gro10; Gro16]. Eventually, zk-SNARKs were found to be the most ideal proof system candidates for cryptographic algorithms requiring very short proofs, efficient verification, proving possession of knowledge of a large computation in a non-interactive way, while also not revealing the information itself. These class of proof systems show great potential to be widely adopted into blockchain technology, and also more suitable to be implemented within digital signature schemes.

Adopting zk-SNARKs into digital signature schemes involves working with a large amount of computations that need to be verified efficiently. The specifics of such a computation will be provided in the upcoming chapter of this thesis. As part of the effort to improve the scalability of zero-knowledge proofs, a protocol that authors Ben-Sasson et al. in [Ben+18a] termed ***Fast Reed-Solomon Interactive Oracle Proofs of Proximity***,

or **FRI** for short, was presented that focuses on the underlying algebraic structure used for verifying computations. This protocol is particularly useful in polynomial commitment schemes [Bon+20], where large polynomials need to be efficiently checked for low degree. It also supports the need for a transparent setup as is reflected in the blockchain technology infrastructure. FRI is an Interactive Oracle Proof [BCS16a] that checks whether a given function is of low degree over a finite field, with extremely low communication complexity and verification cost.

The FRI protocol was previously implemented by works such as Aurora [Ben+19], STARK [Ben+18c], Fractal [COS20], Spartan [Set19], and PREON [Che+23] (which calls Aurora's FRI), amongst others. This work aims to optimize the sub-protocols and efficiently implement the FRI protocol for polynomial commitments for use by algorithms that wish to implement zk-SNARKs, typically as part of digital signature schemes. It validates the following thesis statement:

It is possible to optimize and implement an accelerated version of the FRI protocol for enabling Low-Degree Testing in zk-SNARK protocols in a non-interactive way using the computational powers of a GPU and such a protocol can be efficiently verified with just a single specially-constructed Merkle Tree.

The work done in this thesis is primarily to implement FRI in Polaris [FG22], but stands relevant to any zk-SNARK algorithm that has FRI as one of its core components. In this thesis, we adopt the use of Reed-Solomon codewords, which treat messages as polynomials over a finite field, within the FRI protocol. These polynomials are then evaluated at distinct points within the field, and the resulting evaluations become the codewords. Adoption of RS-codewords to implement FRI is shown in works such as [Ben+18a; Che+23].

Our implementation is primarily software-oriented, wherein we first optimized noted aspects of the protocol that can benefit from such optimizations, and coded them initially in C. Then, we identified certain aspects of the code that could be computed in parallel using the powerful computational tools of the graphics processing unit (GPU), programmed in NVIDIA's CUDA-C. Our results show that with effective software implementation on the GPU and a few optimizations to the inner workings of the algorithm, the protocol is sped-up by an average of 94.4% as compared to the CPU implementation of it with no optimizations. We were specifically able to save on significant performance time through the construction of a specialized Merkle Tree that makes the verification faster while maintaining the authenticity of the protocol.

1.2 Contributions

This thesis implements optimizations on the FRI protocol programmed to run on the GPU for the first time, effectively improving on the low degree testing of polynomials, useful for plugging into zk-SNARK schemes. The FRI-IOP algorithm utilizes Reed-Solomon Codewords for committing to a polynomial and then verifying that the committed polynomial has a low-enough degree. Primarily, we identify and implement the following optimizations to the FRI in our code:

1. *Precomputation of initial basis elements.*
   The FRI protocol is conducted through a number of rounds in which the polynomial to be tested against for its degree is represented as Reed-Solomon codewords. These codewords at each layer involve inverse computation of field elements. Due to the size of each codeword at each round, and due to the fact that there are multiple rounds in the protocol, this computation results in significant overhead in performance during its implementation. Our first contribution is to move this inverse computation to pre-computation. Essentially, the inverse part of the codeword element is identified as the initial basis element of the evaluation domain for each codeword. These values are pre-computed and the data is stored to be retrived while computing the codeword elements for each round of the protocol in our code.

2. *Repeated hashing of a random element by the prover for every round.*
   In the process of codeword computation, a field element is uniformly-randomly by the verifier for each round for the computation of the next codeword. This involves a lot of back-and-forth between the prover and the verifier with the prover having to wait for such a sampled lement from the verifier to carry on with the protocol. Inspired from Fractal's [COS20] FRI implementation, we make the protocol non-interactive by having the prover sample the random element only once before the first round of the FRI protocol, and repeatedly hashing the element for future rounds, and implement this in our code. This would also mean that the verifier, during the later phase of the protocol, would only have to be given the sampled element only once, and can use it to non-interactively verify the protocol at its own time.

3. *Parallelization of FRI's Commit Phase.*
   As a major contribution, this thesis, for the first time to our knowledge, implements the FRI protocol's commit phase on the GPU to benefit from the fact that all indices in a codeword round can be computed in parallel. We make use of NVIDIA's CUDA-C Programming Language compiled and run on a 40-Amp GPU compute machine

to achieve this. Each thread is assigned to each index's next codeword element computation at each round and threads are synchronized at the end to retain the hierarchy of indices in each round.

4. *Implementation and parallelization of a specialized Merkle tree for efficient commitment and verification in the FRI Protocol.*
   The other major and novel contribution of this thesis is the implementation of an *integrated, verifiable* Merkle tree, which efficiently commits all the codewords in one go. This algorithm is referenced from unpublished notes authored by Gong [Gon24b], and implemented in software as a part of this thesis. Effectively, the prover would have to commit to its codewords just in this one specially-constructed Merkle tree, and the verifier would be able to verify the authenticity of all the codewords that the prover commits to, without the prover having to construct a Merkle tree to commit to every codeword. This implementation significantly reduced not just the prover size, but also the prover time and improved the overall performance of the algorithm.

All of these contributions put together is termed *Air-FRI*, where 'Air' indicates the fast performance factor of the FRI Protocol, and doubles to mean 'Acceleration'. These contributions are explained in-detail in the future chapters.

1.3 Thesis Outline

The rest of the thesis has the following structure. Chapter 2 explains existing literature in digital signature schemes using zk-SNARKs that contain FRI as one of their core components. This chapter also presents the FRI protocol in detail, introduces RS codewords with relevance to the protocol and illustrates the FRI-LDT scheme. Chapter 3, Chapter 4, and Chapter 5 explain the thesis contributions in detail. Chapter 3 presents each optimization comprehensively. Chapter 4 presents implementation details of FRI in Software. Chapter 5 discusses the results of our implementations and compares performance to unoptimized versions of the protocol. Chapter 6 presents closing remarks, limitations, and future work.

Chapter 2

Background and Related Works

Often, centralised management systems involve some cryptographic techniques, typically a digital signature, that allows entities to authenticate data as their own and also to serve non-repudiation on the system. However, this process on a blockchain does not come without privacy leaks due to its transparent nature and the increasing number of parties (e.g., users and services). In the survey [Fen+19], the authors present that privacy issues on the blockchain, apart from being user-oriented, can also be categorised as being transaction-oriented. The primary motivation to improve on the identity management in decentralised systems stems from the need to preserve the identities (broadly, privacy) of these entities on the blockchain [YL20]. An immediate challenge with being able to have that is that the participants must reach an agreement on the state of the network and invalidate any transactions that do not seem legitimate, while still adhering to the external privacy constraints that need to be kept during the entire process [FG22]. This makes space for the need to adopt a single non-interactive message to be publicly verifiable by anyone.

2.1 Some Context on Zero Knowledge Proofs

Properties of zero-knowledge proofs

ZKPs are a fundamental cryptographic tool that allow two parties, a prover P and a verifier V to engage in a 'proof' that attests to the validity of a statement without revealing any information beyond the truth of the statement itself. First introduced by [GMR89] in 1985,

ZKP systems must satisfy three essential properties, as interpreted from the definitions in the article by [GO94]: *completeness, soundness,* and *zero-knowledge.*

- *Completeness.* If a statement that a witness x belongs to a language L is true, an honest prover can convince an honest verifier of $x \in L$.
- *Soundness.* For every program run by the prover where $x \notin L$, the verifier rejects x, except with a negligible probability. That is to say, no dishonest prover can convince the verifier of its truth.
- *Zero-Knowledge.* No information other than the validity of the statement is revealed to the verifier. That is to say, for a true statement $x \in L$ that a verifier accepts, they do not learn about x.

In interactive zero-knowledge proofs, the proof system takes place in 3 phases [Sun+21]. First is the *witness phase,* where the prover P computes a proof and sends it over to the verifier V. Next is the *challenge phase,* where V poses a bunch of queries to P, and in the *response phase, P* replies to V with the responses to the queries, which V evaluates to check whether the proof can be accepted or rejected.

Evolution of ZKPs

With time, researches in the field came up with ways of making zero-knowledge proofs non-interactive, where the prover and verifier would have have to engage in back-and-forth interactions to ultimately validate the proof. Goldwasser and Bellare [BG90] introduced Non-interactive Zero-Knowledge (NIZK) as an early significant development in ZKP, which enables for generic theorems to be made zero-knowledge non-interactively. A non-interactive zero-knowledge proof (NIZK) system for a language L is structured in two phases. In the first phase, which can involve interaction, the prover and verifier establish some shared information, and potentially some private information specific to each party. This preliminary phase is independent of the actual theorems being proven. In the second phase, the prover uses the established information to prove specific theorems to the verifier in a non-interactive manner, ensuring that the proof reveals nothing beyond the validity of the theorem itself.

This area of research then witnessed Rackoff and Simon [RS91] introduce non-interactive proofs of knowledge as a way of securing public-key cryptosystems against chosen ciphertext attacks. The authors utilize a trapdoor function to construct NIZK proofs for languages in **NP**, building on earlier works by Blum, Feldman, and Micali [BFM88]. The

paper demonstrates how these proofs can be used to ensure that messages encrypted in a cryptosystem also include a proof that the sender knows the decryption key for the ciphertext, effectively securing the system against a CCA. This work would open the world up to the possibility of making stronger non-interactive ZKPs for wider use-cases. Succinct Non-interactive Arguments (SNARGs) were born from the need to address both efficiency and scalability in cryptographic systems, focusing on minimizing proof sizes and improving verification times. Building on the foundation of NIZKPKs, Micali [Mic94] introduced SNARGs, which provided succinctness by ensuring that the size of the proofs and the time required for verification were both much smaller than the problem itself.

Later, Blum et al. [BFM88] showed that proofs in computational settings can be made non-interactive by having the prover and verifier share a random Common Reference String (CRS). This paved the way for SNARKs to be built and more efficient constructions of succinct proofs came with time. Succinct non-interative arguments of knowledge (SNARKs) for NP were first presented by Bitansky et al. [Bit+12], under the assumption that extractable collision-resistant hash functions exist. Zero-Knowledge non-interactive Proofs of Knowledge (zk-SNARKs) are a special instance of non-interactive proofs of knowledge. They differ from being proofs, in-fact. A proof of knowledge ensures that a dishonest prover is unable to convince a verifier of a false statement, regardless of the prover's resources. In contrast, an argument of knowledge offers the same guarantee but only against provers with limited computational power.

Over time, a wide range of zk-SNARK protocols were introduced, with notable ones whose applications lie within privacy-preservation of transaction data on the blockchain. The succinct proof and efficient verification nature of zk-SNARKs makes the protocol an ideal candidate to be employed on blockchain in the form of transaction data, the ownership of which can be computationally verified.

2.2 Practical zk-SNARKs

Blockchain has important use-cases for incorporating zero-knowledge digital signatures. A well-constructed digital signature scheme used on the blockchain provides a means of ensuring integrity, authentication, and non-repudiation of transactions. Although one could argue that digital signatures themselves are zero-knowledge in nature, there are some details that a non-interactive zero knowledge proof would contribute to the digital signature, that a traditional construction would not. In the immensely popular paper within the ZK community, the authors in [BG90] present that digital signatures should contain proofs that need to be *publicly verifiable*. That is to say, the proof should be constructed such

that anyone on the setup space, whether it is trusted or transparent, should be able to validate the proof, instead of having it constructed for a single verifier.

When we say practical zk-SNARKs, we refer to the proof systems that are based on complex real-world computations through Arithmetic Circuits (ACs) and Quadratic Arithmetic Programs (QAPs) [Gen+12]. Transforming a computational problem into an arithmetic circuit enables the zk-SNARK protocol to process and verify complex computations. By using arithmetic circuits, zk-SNARKs can efficiently encode the logic of statements that need to be proven. A QAP is a mathematical structure used to represent a circuit in such a way that it becomes easy to prove the correctness of the circuit's output without revealing any additional information about the inputs. QAPs are integral to zk-SNARKs because they allow for the construction of succinct proofs that are small and easy to verify [Ben+14b]. QAPs encode the verification of arithmetic circuits into a set of polynomial equations, and the zk-SNARK proof can then convince a verifier that these equations are satisfied without the verifier having to recompute the entire circuit. This method reduces the size of the proof and the complexity of verification, making zk-SNARKs more scalable to be used in blockchain.

Ben-Sasson [Ben+18b] introduced zk-STARKs (Zero-Knowledge Scalable Transparent Arguments of Knowledge), a class of cryptographic proof systems that provides zero-knowledge and transparency while not requiring a trusted setup. Although zk-STARKs offer efficient verification and hold significant promise for enhancing privacy on the blockchain, they are not without their challenges. [Ben+18b; PO21] highlight that the proof sizes in zk-STARKs are roughly 1000 times larger than that of zk-SNARKs. The latter are also proven to outperform zk-STARKs by a small margin. The primary motivation to use zk-STARKs over zk-SNARKs would be that the former: (i) has a transparent setup, (ii) Is post-quantum resilient, since they make use of cryptographic primitives that remain secure even for quantum computers, such as collision-resistant hash functions.

Existing and noteworthy practical zk-SNARKs

Polaris [FG22] is known to be the best of both worlds. This proposed digital signature scheme maintains the short proof size and efficient verification times of a zk-SNARK, while enabling a transparent setup, like in zk-STARKs. Like the latter, Polaris also implements the Fast Reed Solomon Interactive Oracle Proof of Proximity (FRI-IOP) protocol for low-degree test, which makes efficient verification possible. *The work done as a part of this thesis primarily serves the additive FRI protocol for use in Polaris, but is also presented in a general way for use in any zk-SNARK systems.*

Aurora [Ben+19], and Preon [Che+23](which calls some core components from Aurora), represent significant developments in the zk-SNARK space, particularly in the context of transparent setups that do not require a trusted party to generate secret parameters. Introduced in the paper by Eli Ben-Sasson et al., Aurora focuses on minimizing proof size and verification time by utilizing Reed-Solomon Interactive Oracle Proofs (FRI) for low-degree testing. The FRI protocol provides a way to verify polynomial proximity efficiently, a crucial step in transparent zk-SNARK systems. Aurora achieves proof sizes that are logarithmic in the size of the computation, with the verification time scaling as $O(\log^2 n)$, where n is the number of constraints in the computation. Aurora's use of FRI allows it to achieve proofs that are around $100KB$ for relatively large computations, making it more scalable as compared to earlier zk-SNARK systems. Preon extends Aurora by enhancing scalability and efficiency in practical systems. While efficient and post-quantum resilient, Aurora's (and by extension, Preon's) proof sizes can still be larger than zk-SNARK systems that use trusted setups, impacting bandwidth efficiency in some use cases.

Fractal [COS20] is another zk-SNARK system that leverages FRI to achieve transparency and efficiency. Fractal emphasizes optimizing the prover's time complexity while maintaining succinct proofs. Fractal uses Rank-1 Constraint Satisfiability (R1CS), a format for encoding arithmetic circuits that zk-SNARKs typically operate on. The advantage of Fractal lies in its ability to handle complex circuits with many constraints while keeping the proof size small and the verification time efficient. The proof size for Fractal is typically on the order of tens of kilobytes, with verification times scaling logarithmically, similar to Aurora. However, Fractal introduces further optimizations in the prover's work, making it more practical for real-world deployments where prover efficiency is a bottleneck. One of Fractal's key advantages is its ability to handle both transparent setups and post-quantum security, making it robust in scenarios requiring long-term cryptographic assurances.

Spartan [Set19] and Virgo [ZX19] offer zk-SNARK implementations that emphasize scalability and minimal (Spartan) or no (Virgo) trusted setup. Spartan achieves its goals by utilizing polynomial commitments and decomposing circuits into sparse multilinear polynomials. This method allows Spartan to achieve verification times of $O(\log n)$ for verifying circuits with up to a million gates. Its proof size is relatively larger than other zk-SNARK systems, around $200KB$ for circuits with a large number of constraints, and has a sub-linear verification time. Virgo, on the other hand, builds upon the Sum-Check Protocol and achieves proof sizes that are linear in the number of multiplications performed in the circuit. Virgo's verification time is sublinear for arithmetic circuits with depth greater than d, and the proof size for medium-sized computations is typically under $150KB$, depending on the number of gates involved. Virgo also uses fast Fourier transforms (FFT) for proving and verifying computations on polynomial functions.

These zk-SNARKs share a common feature: they all utilize FRI as a core component. FRI enables a shift from using a trusted setup to a transparent setup, allowing zk-SNARKs to produce not only succinct proofs but also post-quantum secure proofs. This is achieved by leveraging Arithmetic Circuits (ACs) and Quadratic Arithmetic Programs (QAPs), enabling these systems to transform large circuit computations into manageable polynomials, ensuring that even complex proofs can be efficiently verified.

2.3 Merkle Trees and Hashing

A crucial aspect of many zk-SNARK constructions is the ability to commit to a polynomial and later reveal evaluations of that polynomial at specific points. Merkle trees provide an efficient solution for this.

Merkle trees are hierarchical data structures used to efficiently verify the integrity of large datasets. They work by recursively hashing pairs of data blocks until a single root hash is obtained. This root hash represents the entire dataset. To verify the integrity of a specific data block, one only needs to provide a logarithmic number of hashes (the Merkle proof) along the path from the data block to the root.

Relevance of Merkle Tree Commitments to Polynomial Commitment Schemes

A polynomial commitment scheme allows a prover to commit to a polynomial $f(x)$ without revealing its coefficients. Later, the prover can open the commitment at a specific point $x = a$ to reveal $f(a)$ and prove that this evaluation is consistent with the original commitment. The authors of [PTT13] are among the first to formally introduce the concept of using Merkle trees for polynomial commitments. It presents a construction based on Merkle trees and proves its security under certain cryptographic assumptions.

Several proof systems utilize Merkle trees for efficient data management within the FRI protocol. Preon [Che+23] utilizes Aurora [Ben+19], a zk-SNARK proof system that employs Merkle trees to commit to the prover's polynomial evaluations. This commitment is then used in the FRI protocol to verify the correctness of the evaluations. Aurora uses the libsnark library [Sc17], which is written in C++, which employs SHA-256. Stark implements the zkSTARK library [Bc17] which employs SHA-256 for their Merkle Tree hashes as well. Virgo's [ZX19] implementation [UC 19] calls the open-source eXtended Keccak Code Package (XKCP) [Kec24b], indicating that the underlying Merkle tree hash function is SHA-3. Fractal [COS20] employs multiple hash functions to specially work with

for various roles across the protocol. Specifically, their algorithm uses Poseidon [Gra+21], a special family of hash functions that were built keeping large constraints of zero-knowledge proof systems in mind [Gro16; Bün+18; GWC19]. Preon makes use of the keccack library [Kec24a] to extend it to SHA3-256.

While we use the OpenSSL's [Ope24] SHA3-256 function to implement Merkle Trees on the CPU, this thesis also uses keccack's hash function for SHA3-256 for our GPU-specific code. Our implementation inputs 1024 bits to the Merkle Tree hash function and the output hash is 256 bits long. More details will be provided in future chapters.

2.4 The FRI-IOP Protocol

Digital Signature algorithms built with zk-SNARKs use the FRI-IOP protocol as one of their core components, because the system often requires these algorithms to efficiently compute proofs for very large polynomials. To do so without revealing the polynomials, these works employ Reed-Solomon codes that can encode the polynomials in question over finite fields. These settings enable for a transparent polynomial commit schemes (PCS) for short proof construction and their efficient verification due to few (interactive) or no (non-interactive) queries.

At the heart of FRI's utility is its ability to verify that a function is of low degree over a finite field. This low-degree test is crucial in zk-SNARKs, particularly when ensuring that a prover has correctly executed complex computations, such as those used in blockchain transactions or smart contracts. The FRI protocol, as seen in systems like Polaris, Aurora and STARK, ensures that zk-SNARK systems remain scalable, offering proof sizes that grow logarithmically with the size of the computation while maintaining transparent setup properties.

2.4.1 Reed-Solomon Codes and the RS Proximity Problem

The family of Reed-Solomon Codes (RS Codes) are foundational to the study of algebraic coding theory. operate on a finite field $\mathbb{F}$. They can be represented as a space of functions $f : S \to \mathbb{F}$ that are evaluations of polynomials of degree $d < \rho n$, where S is an evaluation set of n elements in $\mathbb{F}$ [RS60; Ben+18a]. Such an RS code is represented in [Ben+18a] as $\mathsf{RS}[\mathbb{F}, S, \rho]$. The *RS proximity problem* gives that a verifier, who has oracle access to $f : S \to \mathbb{F}$, is able to distinguish with *large* confidence and through minimal query

complexity, between the case that f is a **codeword** of $\mathsf{RS}[\mathbb{F}, S, \rho]$ and that f is $\delta-$far in relative Hamming distance from all codewords.

2.4.2 RS Proximity Testing

Initially defined and addressed in [RS92], when the verifier is not given any additional data, the RS proximity problem becomes a proximity test. They established that precisely $d+1$ queries are both necessary and sufficient for the verification process: the tester accepts valid codewords with a probability of 1, while functions that are δ-far from the code are rejected with a probability of at least $1-\delta$. In this model, the verifier receives no auxiliary information from the prover, meaning the prover requires no computational effort, performs zero rounds of interactions, and provides a proof of length 0 to convince the verifier that $f \in \mathsf{RS}[\mathbb{F}, S, \rho]$.

The PCPP Model

The Probabilistically Checkable Proofs of Proximity (PCPP) model extends the basic testing problem by providing the verifier with oracle access to both the function $f \in \mathsf{RS}[\mathbb{F}, S, \rho]$ and an auxiliary proof, referred to as the PCPP proof, denoted as π. This proof, generated by the prover, serves as evidence of proximity to the RS codeword. The prover complexity corresponds to the time required to produce π, and the size of π is the proof length.

Verifier complexity, in turn, measures the time needed to generate queries and validate the responses. Building on the foundational Probabilistically Checkable Proof (PCP) theorem [Ben+18a; AS98; Aro+98], PCPPs solve the proximity problem using constant query complexity with proof lengths and prover complexities of $n^{O(1)}$. Alternatively, constant query complexities are now achievable with verification complexity remaining polynomial in $\log n$.

The IOPP Model

Interactive Oracle Proofs of Proximity (IOPP), introduced independently in [Ben+16] and [RRR16], generalize Interactive Proofs (IPs), PCPs, and Interactive Probabilistically Checkable Proofs (IPCPs). Like interactive proofs, IOPPs involve multiple rounds of interaction between the prover and verifier. The prover sends messages $\pi_1, \pi_2, \ldots, \pi_r$, responding to the verifier's prompts. As in PCPs, the verifier reads only specific entries from

these messages, querying them at random locations rather than processing them in full. The query complexity, in this case, refers to the total number of entries read from f and the prover's messages.

2.4.3 The Fast RS IOPP (FRI)

The authors of [Ben+18a] introduce a novel Interactive Oracle Proof of Proximity (IOPP) specifically designed for Reed-Solomon (RS) codes, called the Fast RS IOPP (FRI). It draws inspiration from the Fast Fourier Transform (FFT) [CT65] and leverages a quasi-linear RS-PCPP model as discussed in [BS08]. FRI distinguishes itself by achieving three key properties: (i) linear-time arithmetic complexity for the prover, (ii) logarithmic-time arithmetic complexity for the verifier, and (iii) constant soundness, which guarantees consistent accuracy regardless of input size.

An IOP system, as discussed above, is denoted by $S = (P, V)$, consists of a prover P and a verifier V, both operating as randomized algorithms. For an input x of length n, the system engages in $r(n)$ rounds of interaction, termed as 'round complexity'. The verifier reads $q(n)$ entries from the input and receives messages from the prover. The total size of these messages constitutes the proof length, $\ell(n)$. After interacting with the prover, the verifier's output, denoted as $\langle P \leftrightarrow V \rangle(x)$, is either *accept* or *reject*.

An IOP is considered to be *transparent* (or public-randomness-based) if all verifier messages are generated using public random coins, with the prover aware of the randomness used. This feature ensures that both parties operate with complete knowledge of the random messages that are broadcast.

The IOPP Model for RS Codes

The IOPP extends the IOP framework by focusing on proximity testing, similar to how PCPs of proximity (PCPP) operate. An IOPP for a family of codes C comprises a prover-verifier pair, denoted (P, V). Both parties receive a description of a code $C \in \mathcal{C}$, where C represents a set of functions $f : S \to \Sigma$, defined over a finite set S with an alphabet Σ.

In this setting, the verifier has oracle access to a function $f^{(0)} : S \to \Sigma$ and interacts with the prover, which also receives $f^{(0)}$ explicitly. The interaction occurs over multiple rounds, denoted by the round complexity r, while the verifier performs a total of q queries. This query-based approach ensures that both computational overhead and communication are minimized, making IOPP protocols highly efficient for large-scale applications.

This conversation leads us to the fiat-shamir heuristic [FS87], which generally attests that with a sufficiently random hash function (as part of a random oracle model), it is possible to make an interactive proof system, non-interactive. While implementing a function that practically does this, randomness can be achieved through a seed obtained from a metric (such as the current state) of the system and fed into a hash function. Ofcourse, this cannot be truly random, but it provides that the verifier's randomness is publicly available. The implementation of this protocol is part of this thesis' contribution, and shall be detailed in the next subsection.

2.5 Implementing the FRI Protocol

In this thesis, we were able to translate all of this information-theoretic math to be able to program the protocol efficiently. The mathematical working of this protocol used in this thesis, including the notations and definitions, is credited to Fu and Gong, the authors of Polaris [FG22], to Gong [Gon24a], and to Badakhshan et al, authors of Ursa Minor [Bad+24].

2.5.1 Notations [Gon23]

Let $\mathbb{F} = \mathbb{F}_{2^N}$ and $L \subset \mathbb{F}$ be a linear space with dimension n. Let $\{\beta_0, \beta_1, \cdots, \beta_{n-1}\}$ be a **basis** in L which is ordered according to $i = i_0 + i_1 2 + \cdots + i_{n-1}2^{n-1}$, $i_j \in \mathbb{F}_2$ and $\alpha_i = i_0\beta_0, +i_1\beta_1 + \cdots + i_{n-1}\beta_{n-1} \in L$. That is to say, $L = \{\alpha_0, \cdots, \alpha_{2^n-1}\}$. ($L$ can an affine space, but for simplicity in FRI, we assume that it is a linear space of $\mathbb{F}$.

With the above notation, a Reed-Solomon code with an evaluation set L and rate σ: $0 < \sigma < 1$, denoted as $\mathsf{RS}[L, \sigma]$ is defined as a subset of $\mathbb{F}$, given by

$$RS[L, \sigma] = \{\boldsymbol{f}_L | f \in [x], deg(f) < k' = 2^\ell \sigma\}$$

where,

$$\boldsymbol{f}_L = (f(\alpha_0), f(\alpha_1), \cdots, f(\alpha_{2^n-1})).$$

This is a linear code with minimal distance at least $2^n - k'$. Since $deg(f) < k'$, f has at most k' roots, i.e., each codeword has at most k' zero entries. Thus, the minimal Hamming weight of a codeword is at least $2^n - k'$. We have $k' = 2^k$. (Note that the minimal distance of a RS code is determined by the degree of f, given in [MS77] in 1977.)

2.5.2 Phases of the FRI protocol [Bad+24; Gon24a]

The FRI Protocol is designed to have three functions/phases: (i) the commit phase, (ii) the query phase, (iii) the round-consistency check. We denote this as $FRI(com, query, RC - test)$. As this is an introduction, the optimizations in the protocol are not presented here yet.

Let $L_0 = L$ and $\mathsf{RS}[L, \sigma]$. Let $f(x) \in [x]$ with degree $< \sigma 2^n = 2^k$ ($\sigma = 2^{n-k}$), so $\boldsymbol{f}_L \in RS[L_0, \sigma]$. Given $f(x)$ and $\boldsymbol{f}_L \in RS[L_0, \sigma]$, the protocol is designed to prove to a verifier the following NP statement:

$$deg(f(x)) < 2^k.$$

The process is to iteratively reduce the degree of f *by half* at each step of the proof computation.

The protocol starts with the input consisting of a polynomial function $f(x)$, an initial subspace L_0, a soundness parameter σ, and an RS codeword $\boldsymbol{f}_L \in RS[L_0, \sigma]$. The output is a binary value $b \in \{0, 1\}$, indicating whether the verification passed.

Commit Phase

The commit phase of the FRI protocol involves both the prover and verifier interacting to commit to codewords derived from a polynomial over several rounds. The prover starts with the input $f^{(0)} : L_0 \to \mathbb{F}$, which represents an initial Reed-Solomon (RS) codeword corresponding to the polynomial $f_0(X)$ with a rate ρ.

Commitment Process: For each round $k = 0, \ldots, \mathsf{r} - 1$, the following steps occur:

1. Prover Commitment:
   The prover recursively defines codewords $\boldsymbol{f}^{(k)} : L_k \to \mathbb{F}$ for each round.
   It then computes a Merkle tree commitment for each codeword and sends the Merkle root to the verifier.

2. Verifier Challenge:
   The verifier sends a uniformly random element $\alpha^{(k)} \in \mathbb{F}$ to the prover.

3. Codeword Definition:
   The prover constructs the next codeword $\boldsymbol{f}^{(k+1)}$ with domain L_{k+1} such that, for every i where $0 \leq i < |L_{k+1}|$, the following holds:

$$q_k(L_k[2i]) = q_k(L_k[2i + 1]) = L_{k+1}[i].$$

Each element of the new codeword $\boldsymbol{f}^{(k+1)}$ is derived from the previous codeword:

$$f_{k+1}(L_{k+1}[i]) = \frac{f_k(L_k[2i]) - f_k(L_k[2i+1])}{L_k[2i] - L_k[2i+1]}(\alpha^{(k)} - L_k[2i]) + f_k(L_k[2i]). \tag{2.1}$$

The term $L_k[2i] - L_k[2i+1]$ is $\beta_0^{(k)}$ [Gon24a; Bad+24; Gon23].

At the final round $k = \mathsf{r}$, the prover constructs the codeword $\boldsymbol{f}^{(\mathsf{r})} : L_{\mathsf{r}} \to \mathbb{F}$. This last codeword is directly sent to the verifier as the final commitment.

Query Phase

In the query phase, the verifier extracts the Merkle roots, the challenges $\alpha^{(0)}, \alpha^{(1)}, \ldots, \alpha^{(\mathsf{r}-1)}$, and the final codeword $\boldsymbol{f}^{(\mathsf{r})}$. The verifier also gains oracle access to all intermediate codewords $\boldsymbol{f}^{(0)}, \boldsymbol{f}^{(1)}, \ldots, \boldsymbol{f}^{(\mathsf{r}-1)}$. In symbol, this is represented as $\boldsymbol{f}^{(k)} = (\boldsymbol{f}_{k,0}, \boldsymbol{f}_{k,1}, \cdots, \boldsymbol{f}_{k,|L|_k-1})$.

1. Interpolant Verification: The verifier computes the interpolant polynomial $f_{\mathsf{r}}(X)$ using points from $\boldsymbol{f}^{(\mathsf{r})}$. If the degree of $f_{\mathsf{r}}(X)$ exceeds $\rho \cdot |L_{\mathsf{r}}| - 1$, the verifier rejects the proof.

2. Round Consistency Check: This step ensures the consistency between two consecutive codewords. The verifier performs the following steps ℓ times: A random index $s^{(0)} = i$ is selected, where $0 \le i < |L_0|$. For each $0 \le k < \mathsf{r} - 1$, the index is updated as:

$$s^{(k+1)} = \lfloor s^{(k)}/2 \rfloor.$$

   If $s^{(\mathsf{r})}$ is sampled more than once, the verifier resamples $s^{(0)}$.

3. Verification of Codeword Values: For each k, the verifier queries the values:

$$f_{k+1}(L_{k+1}[s^{(k+1)}]), \quad f_k(x_1^{(k)}), \quad f_k(x_2^{(k)}),$$

   where:

$$x_1^{(k)} = L_k[2s^{(k+1)}], \quad x_2^{(k)} = L_k[2s^{(k+1)} + 1].$$

   The Merkle paths for these queried points are checked to ensure correctness.

Round Consistency Check

The verifier checks the following equation for every $k \in \{0, 1, \ldots, \mathsf{r} - 1\}$:

$$f_{k+1}(L_{k+1}[s^{(k+1)}]) = \frac{f_k(x_1^{(k)}) - f_k(x_2^{(k)})}{x_1^{(k)} - x_2^{(k)}}(\alpha^{(k)} - x_1^{(k)}) + f_k(x_1^{(k)}).$$

If any of these checks fail, the verifier rejects the proof.

Acceptance: If all checks pass, the verifier accepts the proof, confirming that the original polynomial $f_0(X)$ has a degree no more than $\rho \cdot |L_0| - 1$.

2.5.3 A Toy Example Showcasing Bases and Codeword Computations in Each Round

Let $\mathbb{F}_{2^m} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x^2 + 1)$, and let α be a root of the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$. Then, α serves as the generator of the multiplicative group $\mathbb{F}_{2^m}^* = \mathbb{F}_{2^m} \setminus \{0\}$. For a low-degree test, the prover aims to show that the degree of the polynomial

$$f = x^4 + \alpha x^3 + \alpha x + 1$$

is no more than 4.

Define $\mathcal{L}$ as the additive affine subspace of $\mathbb{F}_{2^m}$.

As a first step, consider $|\mathcal{L}| = 2^4$. Since $\mathcal{L}$ is an affine subspace of $\mathbb{F}_{2^m}$, let $s_0 = \alpha^4$. We have the polynomial

$$q(X) = X(X + \beta_0^0),$$

where β_0^0 is the first element of the basis of $\mathcal{L}_0$.

Additionally, for each j, we define:

$$\beta_j^1 = q_0(\beta_{j+1}^0), \quad 0 \le j \le m - 1. \tag{2.2}$$

Here, $m = \log_2(|\mathcal{L}_0|) = 4$.

Breaking down the basis, we obtain the following elements:

$$\beta_0^0 = 1, \quad \beta_1^0 = \alpha, \quad \beta_2^0 = \alpha^2, \quad \beta_3^0 = \alpha^3.$$

Thus, $\mathcal{L}_0$ is generated by the basis $\beta_0 = \langle 1, \alpha, \alpha^2, \alpha^3 \rangle$, with $s_0 = \alpha^4$.

Therefore, we have:

$$\mathcal{L}_0 = \langle 1, \alpha, \alpha^2, \alpha^3 \rangle + \alpha^4.$$

Expanding $\mathcal{L}_0$, we find:

$\mathcal{L}_0 = \{\alpha^4,\ \alpha^4+\alpha^3,\ \alpha^4+\alpha^2,\ \alpha^4+\alpha^3+\alpha^2,\ \alpha^4+\alpha,\ \alpha^4+\alpha^3+\alpha,\ \alpha^4+\alpha^2+\alpha,\ \alpha^4+\alpha^3+\alpha^2+\alpha,$

$\alpha^4+1,\ \alpha^4+\alpha^3+1,\ \alpha^4+\alpha^2+1,\ \alpha^4+\alpha^3+\alpha^2+1,\ \alpha^4+\alpha+1,\ \alpha^4+\alpha^3+\alpha+1,\ \alpha^4+\alpha^2+\alpha+1,\ \alpha^4+\alpha^3+\alpha^2+\alpha+1\}.$

Given $q_0(x) = x(x+1)$, we find that:

$$s_1 = q_0(s_0) = \alpha^4(\alpha^4+1) = \alpha^3 + \alpha^2 + 1.$$

Computing β_j^1:
From the above equation, we try to write the basis β_1 as below.

$$\beta_0^1 = q_0(\beta_1^0) = q_0(\alpha) = \alpha(\alpha+1).$$

In this way,

$$\beta_0^1 = \alpha^2 + \alpha, \beta_1^1 = \alpha^4 + \alpha^2, \beta_2^1 = \alpha^6 + \alpha^3.$$

$\mathcal{L}_1$ for this example is given by the basis $< \beta_0^1, \beta_1^1, \beta_2^1 >$.

$$\mathcal{L}_1 =< \alpha^2 + \alpha, \alpha^4 + \alpha^2, \alpha^6 + \alpha^3 > + s_1$$

In the above equation, $s_1 = q_0(s_0) = q_0(\alpha^4) = \alpha^3 + \alpha^2 + 1$.
So, $\mathcal{L}_1 = \{\alpha^6+\alpha+1, \alpha^6+\alpha^2+1, \alpha^3+\alpha+1, \alpha^3+\alpha^2+1, \alpha^6+\alpha^4+1, \alpha^6+\alpha^4+\alpha^2+\alpha+1, \alpha^4+\alpha^3+1,$
$\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1\}$.

$q_1(x) = x(x+(\alpha^2+\alpha))$, and $s_2 = q_1(s_1) = q_1(\alpha^3+\alpha^2+1) = \alpha^6+\alpha^5+\alpha^4+\alpha^3+\alpha^2+\alpha+1.$

Computing β_j^2:
Now, $\beta_0^2 = q_1(\beta_1^1) = q_1(\alpha^4+\alpha^2) = (\alpha^4+\alpha^2)(\alpha^4+\alpha^2+(\alpha^2+\alpha)) = \alpha^6+\alpha^5+\alpha^4+\alpha^2+1.$
In this way,

$$\beta_0^2 = \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + 1, \beta_1^2 = \alpha^5$$

$\mathcal{L}_2$ is given by the basis $< \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + 1, \alpha^5 >$ added with s_2.
And, $s_2 = \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1$.
So,

$$\mathcal{L}_2 = \{\alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1, \alpha^6 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1, \alpha^3 + \alpha, \alpha^5 + \alpha^3 + \alpha\}.$$

$q_2(x) = x(x + (\beta_0^2)) = x(x + (\alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + 1))$
$s_3 = q_2(s_2) = \alpha^5 + 1$

Similarly, we compute β_j^3 and β_j^4. The bases values from the computation are listed in table 2.1, and the domain values are shown in table 2.2.

Table 2.1: Values of $q_i(x)$ and basis elements for each domain.

i	$q_i(x)$	Basis
0	$x(x+1)$	$< 1, \alpha, \alpha^2, \alpha^3 >$
1	$x(x + \alpha^2 + \alpha)$	$< \alpha^2 + \alpha, \alpha^4 + \alpha^2, \alpha^6 + \alpha^3 >$
2	$x(x + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + 1)$	$< \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2 + 1, \alpha^5 >$
3	$x(x + \alpha^6 + \alpha^5 + \alpha^4 + \alpha)$	$< \alpha^6 + \alpha^5 + \alpha^4 + \alpha >$
4	$x(x + \alpha^6 + \alpha^5 + \alpha^4 + \alpha)$	–

Table 2.2: Elements of $\mathcal{L}_i$ for each iteration.

i	$\mathcal{L}_i$
0	$\{\alpha^4, \alpha^4+\alpha^3, \alpha^4+\alpha^2, \alpha^4+\alpha^3+\alpha^2, \alpha^4+\alpha, \alpha^4+\alpha^3+\alpha, \alpha^4+\alpha^2+\alpha, \alpha^4+\alpha^3+\alpha^2+\alpha, \alpha^4+1, \alpha^4+\alpha^3+1, \alpha^4+\alpha^2+1, \alpha^4+\alpha^3+\alpha^2+1, \alpha^4+\alpha+1, \alpha^4+\alpha^3+\alpha+1, \alpha^4+\alpha^2+\alpha+1, \alpha^4+\alpha^3+\alpha^2+\alpha+1\}$
1	$\{\alpha^6+\alpha+1, \alpha^6+\alpha^2+1, \alpha^3+\alpha+1, \alpha^3+\alpha^2+1, \alpha^6+\alpha^4+1, \alpha^6+\alpha^4+\alpha^2+\alpha+1, \alpha^4+\alpha^3+1, \alpha^4+\alpha^3+\alpha^2+\alpha+1\}$
2	$\{\alpha^6+\alpha^5+\alpha^4+\alpha^3+\alpha^2+\alpha+1, \alpha^6+\alpha^4+\alpha^3+\alpha^2+\alpha+1, \alpha^3+\alpha, \alpha^5+\alpha^3+\alpha\}$
3	$\{\alpha^5+1, \alpha^6+\alpha^4+\alpha+1\}$
4	$\{\alpha^7+\alpha^6+\alpha^5+1\}$

FRI Commit Phase

After obtaining the subspace elements, the codeword can be computed and committed in the commit phase.
$f^{(0)} : L_0 \to \mathbb{F}$, a purported Reed Solomn codeword corresponding to polynomial $f_0(X)$, with rate ρ. As a first step, we compute $f_0(L_0)$.

$f^{(0)} = f_0(L_0) = \{\alpha^6+\alpha^4+\alpha^3+\alpha^2+1, \alpha^7+\alpha^4+\alpha, \alpha^7+\alpha^3+\alpha^2+\alpha, \alpha^7+\alpha^6+\alpha^3+\alpha^2+1, \alpha^5+\alpha^4+\alpha, \alpha^7+\alpha^4+\alpha^3, \alpha^6+\alpha^5+\alpha^4+\alpha^3+\alpha^2+\alpha, \alpha^6+\alpha^4+\alpha^3, \alpha^7+\alpha^5+\alpha^4, \alpha, \alpha^6+\alpha^3+\alpha^2, \alpha^6+\alpha^5+\alpha^4+\alpha, \alpha^7+\alpha^5+\alpha^3, \alpha^6+\alpha^5+\alpha^4+\alpha^3+\alpha^2+\alpha+1, \alpha^7+\alpha^6+\alpha^3+\alpha+1, \alpha^7+\alpha^4\}$.

Here, $|f_0(L_0)| = 16$.

The values in codeword $f^{(k+1)}$ is derived from the previous codeword as shown in (2.1). In this equation, the denominator is $\beta_0^{(k)}$, whose inverse shall be pre-computed.

From (2.1), we try to compute f_1, f_2, f_3, f_4.

Computation of f_1:
For each point $x \in L_1$, we compute $f_1(x)$ using the evaluations $f_0(L_0[2i])$ and $f_0(L_0[2i+1])$ as follows:

$$f_1(L_1[i]) = \frac{f_0(L_0[2i]) + f_0(L_0[2i+1])}{\beta_0^{(0)}}(\alpha - L_0[2i]) + f_0(L_0[2i])$$

$$\begin{aligned} f_1(L_1) = \{&\alpha^7 + \alpha^5 + \alpha^4 + \alpha, \\ &\alpha^6 + \alpha^4 + \alpha^3 + \alpha^2 + 1, \\ &\alpha^6, \\ &\alpha^7 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2, \\ &\alpha^7 + \alpha^4 + \alpha^2, \\ &\alpha^5 + \alpha^4 + 1, \\ &\alpha^7 + \alpha^3 + \alpha^2 + \alpha, \\ &\alpha^5 + \alpha^4 + \alpha^3\} \end{aligned}$$

```
f2 = [
    alpha^7 + alpha^3,
    alpha^7 + alpha^6 + alpha^5 + alpha^3 + 1,
    alpha^7 + alpha^5 + alpha^4 + alpha^3 + alpha,
    alpha^7 + 1
]

f3 = [
    alpha^6 + alpha^2,
    alpha^7 + alpha^6 + alpha^5 + alpha^4 + alpha^2 + alpha + 1
]

f4 = [
    alpha^7 + alpha^4 + alpha^3 + alpha + 1
]
```

Listing 2.1: Computed values of f2, f3, and f4

It is clear that f_1 would be 8 elements. Similarly, f_2, f_3, and f_4 are computed by implementing (2.1) in SageMath [The] as shown in Listing 2.1.

Table 2.3: Computed values of $f_1(x)$ based on f_0 evaluations over domain L_0.

x	$f_0(L_0[2i])$	$f_0(L_0[2i+1])$	$f_1(x)$
$L_1[0]$	3	7	5
$L_1[1]$	8	6	7
Queried $L_1[2]$	**4**	**2**	**3**
$L_1[3]$	9	5	7

Query Phase

In the query phase, the verifier selects an index i (e.g., $i = 1$). The verifier computes the indices $2i$ and $2i+1$, which correspond to points in L_0, and requests these values from the prover. For the sake of abstraction, let us assume some numerical values for each of these computations, as presented in Table 2.3.

- The verifier picks $i = 1$.
- The verifier computes $2i = 2$ and $2i + 1 = 3$.
- The verifier queries the prover for $f_0(L_0[2])$ and $f_0(L_0[3])$.

The prover responds by providing the values $f_0(L_0[2]) = 4$ and $f_0(L_0[3]) = 2$, as well as the Merkle Tree commitment authentication path for these values.

Verification

The verifier performs the following steps to verify the correctness of the proof:

1. **Merkle Tree Verification:** The verifier checks the authentication paths for $f_0(L_0[2])$ and $f_0(L_0[3])$ in the Merkle tree, reconstructing the root r'.
2. **Root Comparison:** The verifier compares r' with the original root r provided by the prover. If $r = r'$, the verifier proceeds to the co-linearity check.

3. **Co-linearity Check:** The verifier computes $f_1(L_1[1])$ as: [1]

$$f_1(L_1[1]) = \frac{f_0(L_0[2]) + f_0(L_0[3])}{2}((*) + (*)) = \frac{4+2}{2} = 3$$

The verifier then compares this with the value $f_1(L_1[1]) = 3$ obtained in Table 2.3. If the values match, the co-linearity check passes.

If both the Merkle Tree verification and co-linearity check are successful, the verifier accepts the proof.

2.6 Software Implementations of FRI

In this section, we shall discuss the existing algorithmic implementations which call the FRI as one of their core components. Also, some discussion on why CUDA-C programming language was chosen for our GPU implementation is presented.

2.6.1 Existing Software for FRI

The algorithms [Ben+19; Che+23; COS20; Ben+18c] exhibit the following observations with relevance to software implementations.

- Aurora [Ben+19], Fractal [COS20], and another related work in the space, Ligero [Ame+17], all leverage libiop [Sc24], a C++ library specifically designed for zk-SNARK constructions, and which operates primarily on lightweight symmetric cryptography to enhance performance and security.
- To date, most official implementations of FRI-based protocols are designed for CPU execution only. This CPU-centric focus results in limitations on scalability and efficiency when processing large datasets or handling high-throughput environments.
- One notable drawback is the absence of GPU-accelerated implementations, which could offer substantial speedups given the parallel nature of FRI protocol computations, especially for polynomial interpolation and Merkle tree construction. This lack of GPU support has motivated recent work exploring alternative, hardware-friendly implementations.

[1] This is an abstracted version of the original equation, to enable simplistic numerical computation for the sake of demonstration. The actual line computation is shown in (2.1).

- Rust and Go have also been explored in zk-STARK protocol implementations but remain in early stages for FRI protocols. Rust's performance benefits and Go's concurrency model suggest promise, though the libraries currently lack the robustness and maturity of their C++ counterparts.

2.6.2 Programming the FRI Protocol in C (on the CPU) and CUDA-C (on the GPU)

Programming cryptographic protocols, such as the FRI protocol, in C can offer several advantages in terms of performance, control, and compatibility with existing libraries and software. Initial attempts were made to implement the protocol with C++, but since the implementation utilizes the standardised field operation and polynomial evaluation code of Preon [Che+23], C stood as the most appropriate choice. C was also chosen over C++ despite the latter supporting GNU Multi-Precision libraries to handle large variable sizes due to ease in scalability to GPU-compatible code, which does not understand abstracted memory handling of GMP-like libraries. More precisely, the programming language of choice (C) was made for the below major reasons.

- **Compatibility:** Being a part of a larger scheme, and by virtue of having to possibly implement a range of other libraries, implementing FRI in C would make the code compatible for segments of a (larger) algorithm calling this API to be straightforward.
- **Efficiency:** C is a low-level programming language that provides precise control over memory management, which is essential for cryptographic protocols that handle large data sets or perform intensive computations. Implementing these protocols in C can ensure faster execution compared to higher-level languages such as Python or Sage.
- **Fine-Grained Memory Management:** The protocol requires careful handling of sensitive data, such as large (256-bit for L2 security and 320-bit for L3 security) field elements and intermediate values, to prevent leaks or unauthorized access. C allowed for a controlled memory allocation and deallocation, helping mitigate risks such as memory leaks or buffer overflows.
- **Portability to GPU code:** Coding in C allows for the implementation to run on a variety of platforms with minimal modification, making it an ideal choice for implementing cryptographic protocols that need to work on hardware accelerators

like GPUs. CUDA, which is described about in the following subsection, can be integrated into C code to leverage GPU computations for tasks like field operations, polynomial evaluations, and hashing.

While trying to pick a programming language that would fit best for our implementation, a bunch of options were explored. At first, attempts were made to make our Sage code be ported to Python's Numba, which uses Numpy functions to use in GPU kernels. This version would also use Cython, which helps compile certain objects in Python code to speed-up the code's performance. But this required re-writing a lot of our Sage code in Python, which was not ideal for our computationally-intensive finite field and polynomial operations. Then, pyCUDA was explored but in-vain, since it had similar limitations. The ultimate choice was between CUDA-C and CUDA-C++, and to make our code compatible with Preon's [Che+23] finite field operations, which is borrowed for use in our implementation (as will be described in detail in the later chapters), CUDA-C was picked for our implementation.

Chapter 3

Optimizations in Air-FRI to Improve Overall Performance of the Protocol

The primary motivation behind the optimizations presented in this thesis lies in addressing the computational and memory limitations encountered in deploying zk-SNARKs for large-scale cryptographic applications, particularly those employing the FRI (Fast Reed-Solomon Interactive Oracle Proofs of Proximity) protocol. As zk-SNARKs evolve to support more complex circuits and transparent setups, ensuring both scalability and efficiency becomes paramount. In traditional zk-SNARK implementations, the prover often incurs significant overhead when handling large arithmetic circuits or polynomial evaluations, especially when encoding computations into Rank-1 Constraint Satisfaction (R1CS) systems. This bottleneck becomes even more pronounced with protocols like Aurora and Fractal, where polynomial commitments using FRI require extensive memory management and intensive field operations [Sc24] [Lev23].

Another pressing challenge addressed in this work is the prover's time complexity. Existing FRI-based zk-SNARK systems, despite offering post-quantum security and transparent setups, exhibit a trade-off between proof size and prover efficiency. For example, while Aurora achieves proof sizes in the range of 100–150 KB and verification times that scale logarithmically, it does so at the cost of increased prover workload, scaling with $O(n \log n)$ operations for large circuits. Similarly, Fractal introduces optimizations that reduce verification time but still imposes heavy computational demands on the prover due to multi-round polynomial evaluations. These constraints highlight the necessity for novel optimizations that mitigate the prover's computational burden while maintaining the succinctness and transparency of FRI-based proofs.

This thesis contributes to the field by proposing optimizations that focus on parallelizing polynomial computations and integrating Merkle tree commitments directly into GPU-accelerated workflows. By employing CUDA-enabled GPU kernels to offload polynomial evaluations and manage Merkle commitments in a streamlined manner, the proposed solution aims to reduce both memory overhead and prover time, without sacrificing the transparency and security guarantees inherent in FRI-based systems. These optimizations are expected to enable faster proof generation and better scalability for zk-SNARK applications, particularly those deployed on blockchain networks, where throughput and latency are critical considerations for adoption.

The optimizations to the FRI protocol that are contributed to be a part of this thesis is four-fold, which is shown in the following four sections.

3.1 Pre-computing the Inverses of Domain Bases for Next Codeword (Line) Computation

The **line computation** within the FRI protocol plays a pivotal role in recursively transforming polynomial evaluations across different rounds, ensuring proximity to low-degree polynomials. This computation is expressed through (2.1), as presented in the previous chapter.

In (2.1), $\alpha^{(k)}$ is a uniformly random challenge issued by the verifier, and the **denominator term**

$$\beta_0^{(k)} = L_k[2i] - L_k[2i+1]$$

is critical for the polynomial transformation.

Calculating the inverse of $\beta_0^{(k)}$ at each iteration is computationally intensive due to the overhead associated with division operations in finite fields $\mathbb{F}_p$. To address this bottleneck, it is advantageous to **pre-compute** these inverses before initiating the recursion. By storing these inverses in advance, subsequent rounds can reuse the pre-computed values, thus replacing division operations with **multiplications**, which are significantly faster within field arithmetic [Ben+19; COS20].

This strategy minimizes the prover's computational overhead, especially when working with large circuits that require multiple rounds of polynomial transformations. Each time a verifier sends a new challenge $\alpha^{(k)}$, the prover can use the pre-computed inverse to efficiently perform the necessary arithmetic without recomputing it during each query.

3.2 Repeated Hashing of Sampled Element for Each Round to Make Protocol Non-Interactive

The second optimization leverages the technique of **repeated hashing** of the uniformly-randomly-sampled element for codeword computation, denoted by α_k, during each round of the protocol. This method ensures that the prover does not need to transmit all intermediate data directly to the verifier. Instead, the challenges for each round are generated deterministically through a chain of hash operations. This reduces the need for interaction and makes the protocol non-interactive, while still ensuring that the verifier can faithfully reconstruct the necessary data to perform verification.

This optimization follows the principles of the **Fiat-Shamir heuristic**, which replaces interactive communication with cryptographic hashing to simulate verifier challenges. After sampling the initial element α_0 in the first round, the second challenge is computed as:

$$\alpha_1 = \text{Hash}(\alpha_0 \parallel \text{root}),$$

where root is the relevant state from the first round.

From the next round onward, the random element is computed as below.

$$\alpha_{k+1} = \text{Hash}(\alpha_k),$$

where $1 \leq k \leq r$, with r being the total number of rounds in the FRI protocol. By using the hash function iteratively, the prover ensures that every subsequent challenge is derived deterministically, making it unnecessary to communicate these challenges explicitly to the verifier.

This approach significantly reduces communication overhead, as the verifier only requires the initial α_0 and the hash function to compute the entire sequence of challenges. This is illustrated by authors Ben-Sasson et al. in the paper [BCS16b]. Section 5 of the referenced paper elaborates on how this method transforms the protocol into a non-interactive one, aligning with the Fiat-Shamir model.

The use of repeated hashing is not entirely novel. Protocols such as **Fractal** already incorporate similar techniques to achieve non-interactivity through deterministic challenge generation. However, the adaptation in this work builds on these frameworks, tailoring them to the specific requirements of the system being optimized. This optimization ensures

efficient data handling while preserving the soundness and succinctness of the proof system [Ben+19; COS20; FS87].

By minimizing interaction, this approach enhances the protocol's scalability and efficiency. It also facilitates integration with **GPU-accelerated** workflows, where parallelism can be exploited to further speed up hashing operations and proof generation. As with other transparent FRI-based zk-SNARKs, this optimization ensures that the system maintains the desired level of **post-quantum security** without compromising on performance [Lev23].

3.3 Parallel Computation of Codeword Elements on the GPU

In the FRI-based zk-SNARK systems implemented in this work, the parameter sizes are large, and the arithmetic is performed over finite fields with varying bit-widths across different levels of security. Specifically, the elements for each domain are 192-bit field elements for L_1, 256-bit field elements for L_3, and 320-bit field elements for L_5. This notation is provided in Preon's documentation [Che+23], which this thesis also follows.

Given the large size of these elements, direct computation on the CPU becomes both time- and memory-intensive. To address this challenge, this work leverages GPU parallelization to distribute the computation of each codeword element across multiple threads, which significantly accelerates the proof generation process.

The FRI protocol recursively reduces the size of the codeword in each round. For a given round k, the next codeword element $f_{k+1}(L_{k+1}[i])$ is computed from two elements of the previous codeword, as shown in (2.1).

The recursive relation of the equation, where two elements from the previous codeword— $f_k(L_k[2i])$ and $f_k(L_k[2i+1])$—are combined to compute the next codeword element, lends itself well to parallelization. Each thread on the GPU is assigned to compute one element of the next codeword in parallel, avoiding sequential bottlenecks.

CUDA kernels are used to assign each GPU thread a specific element of the next codeword. Each thread reads the values $f_k(L_k[2i])$ and $f_k(L_k[2i+1])$ from global memory, performs the arithmetic operations, and stores the result in the next codeword array. To minimize latency, the inverse of the denominator $L_k[2i] - L_k[2i+1]$ is precomputed and reused across all threads, eliminating redundant computations and further enhancing efficiency.

Figure 3.1 demonstrates the recursive computation process for the next codeword element. Each element i of the next codeword $f_{k+1}(L_{k+1}[i])$ is derived from two elements of the previous codeword: $2i$ and $2i+1$.

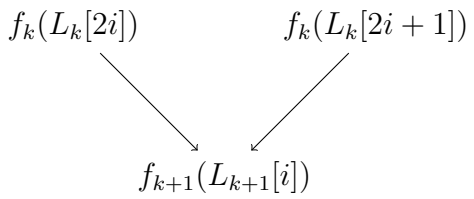


Figure 3.1: Computation of the next codeword element $f_{k+1}(L_{k+1}[i])$ from two previous codeword elements.

Figure 3.2 shows the tree-like structures of both the codeword computations and the Merkle tree commitment, essentially making the motivation to parallelize much clearer. In fact, through the figure, it is easy to imagine a superimposed structure that integrates both of these computations into one.

3.4 Implementing an Integrated FRI Merkle Tree

In this section, we describe the construction of the Integrated FRI Merkle tree and the integrated process of RS codeword evaluations and commitments. These steps are essential for generating and verifying low-degree polynomial approximations efficiently. The primary goal is to reduce the prover's computational overhead while ensuring the verifier's ability to validate the integrity of the proof through a recursive structure of polynomial evaluations. This construction relies on Reed-Solomon (RS) codewords evaluated over a sequence of domains and committed using a Merkle tree, ensuring both scalability and security in the zk-SNARK setting [Ben+19; COS20].

3.4.1 RS Tree Structure

The recursive nature of the FRI protocol can be represented as a binary tree. The elements of the initial codeword f_0 are placed as the leaf nodes at layer 0. Each subsequent layer $k+1$ is constructed from the elements of layer k using the linear combination described earlier. At the final layer r, there are $|L_r|$ nodes. This recursive binary structure is referred to as the RS tree with height r and initial codeword $\hat{f}_0$. The tree is denoted by $RST(f_0, r)$, where $f_0(x)$ is the original polynomial evaluated over the domain L_0.

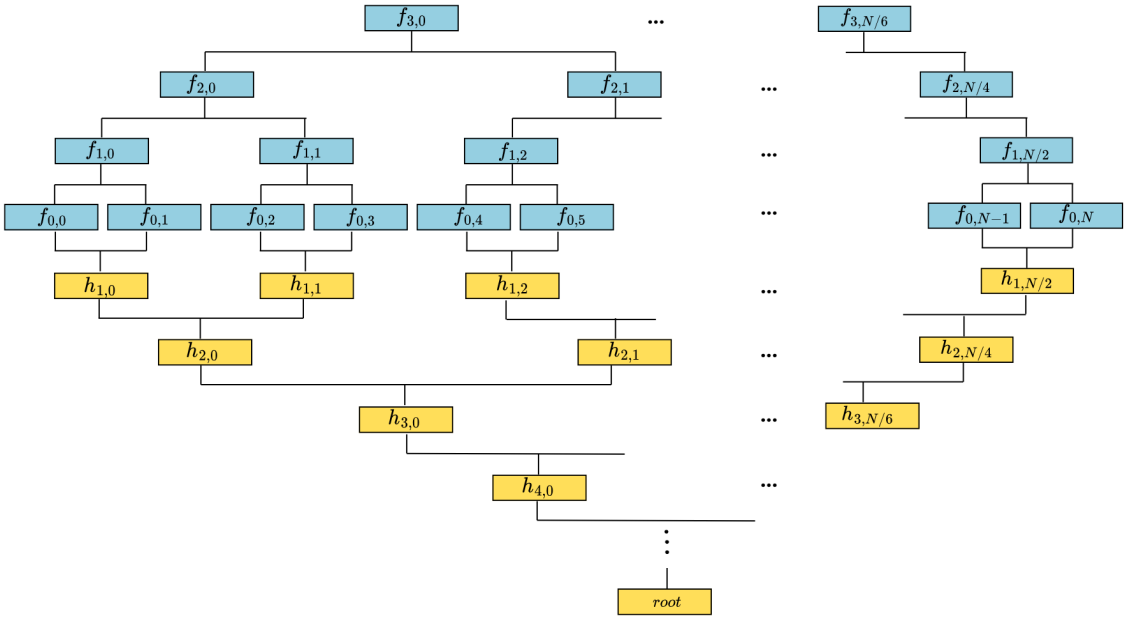


Figure 3.2: Parallel Computations of Codeword Elements and Merkle Tree Hashes

3.4.2 Merkle Tree Commitment of RS Codewords

To guarantee the integrity of Reed-Solomon (RS) codewords, we construct a Merkle tree that integrates seamlessly with the RS tree structure. This Merkle tree provides an efficient commitment scheme by hashing pairs of codeword elements along with their corresponding RS values at each layer. The construction proceeds as follows.

For layers $0 \leq k \leq r$, we compute hash values iteratively [1]:

$$\begin{aligned}
h_{1,i} &= h(f_{0,2i}, f_{0,2i+1}), && 0 \leq i < |L_1|, \\
h_{2,i} &= h(h_{1,2i} \parallel f_{1,2i}, h_{1,2i+1} \parallel f_{1,2i+1}), && 0 \leq i < |L_2|, \\
&\vdots \\
h_{k+1,i} &= h(h_{k,2i} \parallel f_{k,2i}, h_{k,2i+1} \parallel f_{k,2i+1}), && 0 \leq i < |L_{k+1}|.
\end{aligned}$$

[1] The mathematical working of this algorithm including the construction and Authentication Path verification, as stated in Chapter 1 and as described below, is referenced from the unpublished notes by Gong [Gon24b].

For subsequent layers, where $r \leq k < n$, we compute hashes without appending RS values:

$$h_{k+1,i} = h(h_{k,2i}, h_{k,2i+1}), \quad 0 \leq i < 2^{n-k}.$$

The final hash at the top layer provides the Merkle root:

$$h_{n,0} = h(h_{n-1,0}, h_{n-1,1}).$$

This root serves as the commitment for the FRI protocol's Merkle tree, thereby ensuring that all RS codewords are securely and efficiently committed. Using SHA3-256, which outputs 256-bit hashes, allows an input size of up to 1320 bits, enabling efficient processing of codeword elements: 1024-bit inputs are used for layers $0 < k < r$, while 512-bit inputs are employed for layers beyond r.

Figure 3.3 shows the structure of our specialized Merkle tree constructed for the purpose of efficiently parallelizing the FRI protocol.

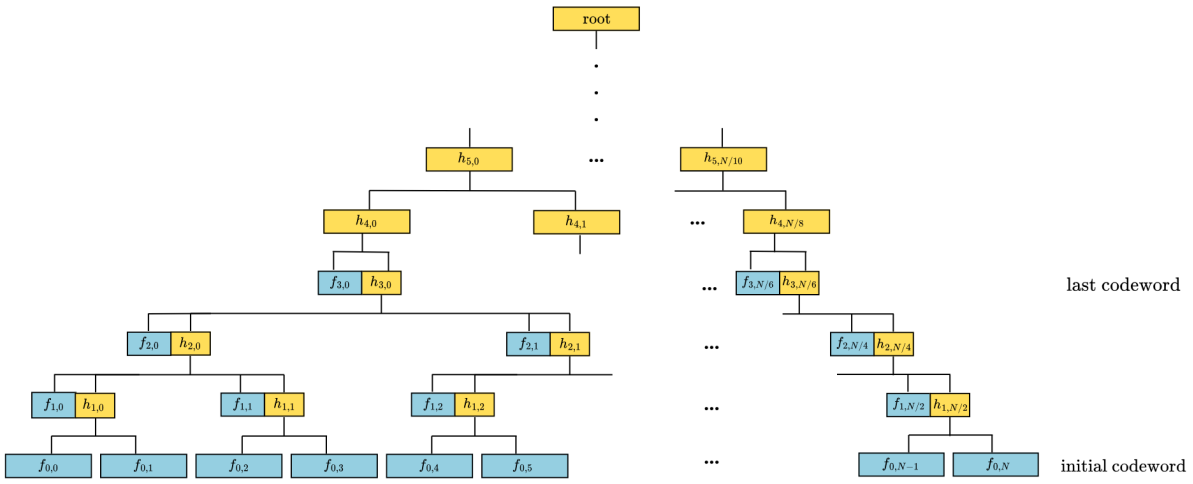


Figure 3.3: Integrated FRI Merkle Tree

Time efficiency of this optimization

As compared to that of the CPU, the number of computations that are reduced is highlighted below. On the CPU, the equation which provides the number of computations is given as below:

For $0 \leq n \leq log_2(N)$, where N is the length of the first codeword $\boldsymbol{f}_0$, the number of computations:

$$\boldsymbol{f}_0 = 2^n + 2^{n-1} + \cdots + 2 + 1 = 2^{n+1} - 1$$

And for $\boldsymbol{f}_1$:

$$\boldsymbol{f}_1 = 1 + 2 + \cdots + 2^{n-1} = 2^n - 1$$

So, for the entire Integrated FRI Merkle Tree leading up to the root, the total number of sequential computations (including computing codewords and hash values):

$$(2^{n+1} + 2^n + \cdots + 2 + 1) - n = 2^{n+2} - 1 - n.$$

By implementing this algorithm through parallel computations of codeword in the Integrated FRI Merkle Tree on the GPU, the total number of computations would be reduced to $2^{n+1} - 1$.

3.4.3 Verification and Authentication Paths

To verify the authenticity of two neighbouring points, starting with $s_0 \in L_0$ at layer 0 and extending to $s_k \in L_k$ for $0 < k < r$, the verifier uses authentication paths in the Merkle tree. These paths contain the concatenated hash values of the queried elements and their corresponding RS codewords. Notably, while the leaf layer hashes $\boldsymbol{f}_0$ separately, the intermediate layers $\boldsymbol{f}_k$ (for $1 \leq k < r$) do not require separate hashing, as their values are integrated into the hash tree structure.

The integration of RS codewords with Merkle tree commitments in the FRI protocol offers both scalability and security. The recursive construction of the RS tree ensures efficient polynomial evaluations, while the Merkle tree provides a compact and secure commitment to the underlying codewords. These combined structures are essential for deploying zk-SNARKs in real-world applications, such as blockchain networks, where both performance and security are paramount.

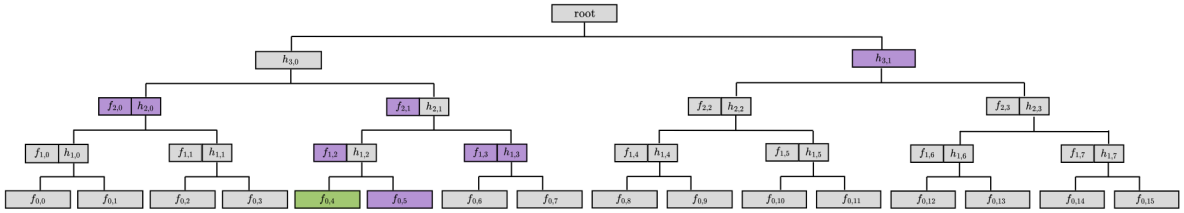


Figure 3.4: Example Authentication Path (in purple) for codeword f_{04} (in green)

An Example for Merkle Tree Authentication Path and Verification

If the verifier queries the index `4` in the first round of verification. This means that the queried codeword element is $f_{0,4}$.

The authentication of f_{04} follows the following path:

$$\{f_{0,4}, (f_{1,2}, h_{1,2}), (f_{2,1}, h_{2,1}), h_{3,0}, \text{root}\}.$$

In order to follow this path, the verifier needs the prover to open the commitment at $f_{0,4}$. Also, the verifier can only perform hash operations, everything else need to be handed to it. This means that the prover needs to send the verifier, the following information (Also see Figure 3.4):

$$\{f_{0,5}, f_{1,2}, (f_{1,3}, h_{1,3}), f_{2,1}, (f_{2,0}, h_{2,0}), h_{3,1}\}$$

With the above path, the verifier would be able to compute hashes as shown in figure 3.5.

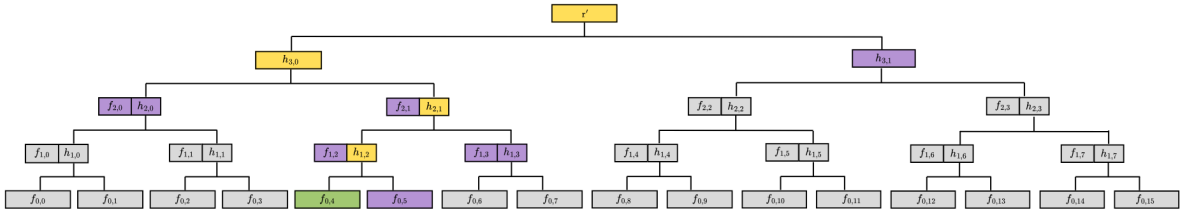


Figure 3.5: Example Verifier's computations (in yellow) leading up to the root

$$H(f_{0,4}, f_{0.5}) \rightarrow h_{1,2}$$
$$h((f_{1,2}, h_{1,2}), (f_{1,3}, h_{1,3})) \rightarrow h_{2,1}$$
$$h((f_{2,0}, h_{2,0}), (f_{2,1}, h_{2,1})) \rightarrow h_{3,0}$$
$$h(h_{3,0}, h_{3,1}) \rightarrow r'$$

The verifier checks if $r' =$ root, and returns 1 if true. Else, returns 0. This process also establishes authenticity in co-linearity tests.

Chapter 4

Implementation Methodology of Air-FRI

This chapter aims to shed light on the manner in which the protocol was implemented in code, and about the GPU system architecture that makes it work.

4.1 Architectural Overview

The architectural details of the protocol, on a system level, can be broken down into several key modules, each handling different aspects of the protocol.

4.1.1 Components

Figure 4.1 shows the overview of the software components that make the implementation. The tiles in green represent code that is borrowed from Preon's [Che+23] implementation. The purple tiles show our code. The blue tile represents open-source library functions.

4.1.2 Implementation Assumptions

The implementation of this FRI protocol is based on several key assumptions that ensure both functionality and performance in cryptographic settings. First and foremost, it is assumed that any entities or participants involved in using this protocol have access to

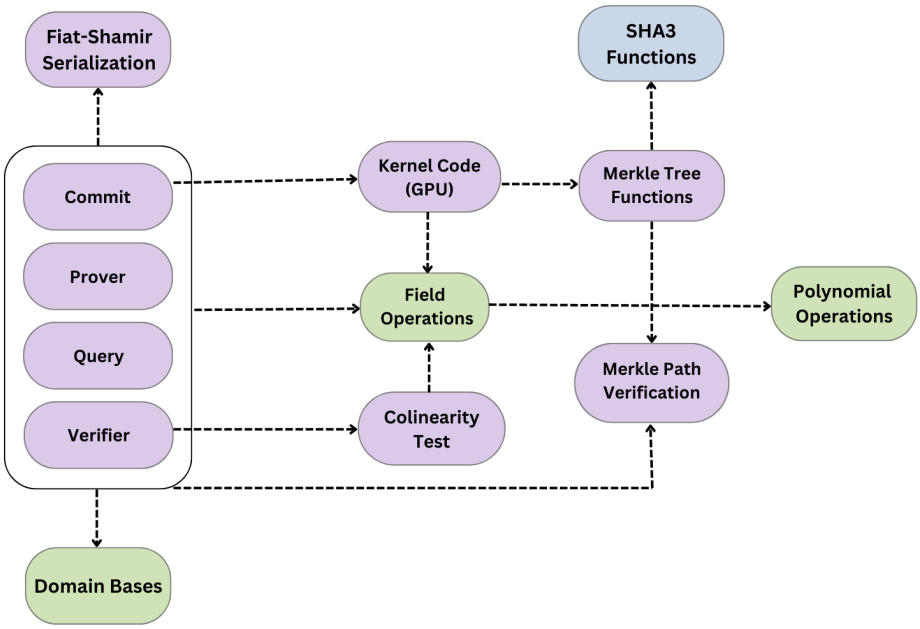


Figure 4.1: Software Components of Our Implementation

GPU systems capable of executing CUDA-based applications. The use of CUDA-C allows the protocol to take full advantage of modern GPU architectures, enabling the parallel execution of tasks like polynomial evaluation, hashing, and consistency checks in the commit and verify phases. These operations are significantly accelerated compared to their CPU counterparts, which is essential for maintaining performance, especially when dealing with large codewords or high-depth Merkle trees.

Another assumption underlying this implementation is that the code is fully compatible with CUDA-C, meaning that both the setup and runtime environments are configured to support GPU execution. This includes ensuring that the necessary CUDA libraries and drivers are correctly installed and that the hardware meets the required specifications for optimal performance. The CUDA toolkit and NVCC compiler are dependencies to compile and run the code. Moreover, the design assumes the presence of efficient memory management capabilities, as memory transfers between the CPU and GPU need to be handled with minimal overhead.

4.1.3 Use of Open Source Libraries

OpenSSL [Ope24] is used in the code primarily for cryptographic operations, particularly hashing. In the context of the FRI protocol written in C, Our code uses OpenSSL's SHA3-256 function while randomly sampling the element for line computation. The hash function ensures deterministic selection of the element based on a computed seed, as expressed in the previous chapter. OpenSSL is a widely adopted library for cryptography, known for its performance and compliance with cryptographic standards, making it an ideal choice for implementing secure hash functions in the FRI protocol.

The code snippet below shows the invocation of OpenSSL's EVP functions for computing hashes.

```
    void hash_sha3_256(const uint64_t *data, size_t len, uint64_t *out) {
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
    const EVP_MD *md = EVP_sha3_256();

    EVP_DigestInit_ex(mdctx, md, NULL);
    EVP_DigestUpdate(mdctx, data, len);
    EVP_DigestFinal_ex(mdctx, (unsigned char *)out, NULL);

    EVP_MD_CTX_free(mdctx);
}
```

For the GPU version of our implementation, we use Keccak [Kec24a] (or more specifically, the SHA3-256 variant) to compute cryptographic commitments such as Merkle tree roots and authentication paths. The sponge construction of Keccak makes it particularly suited for applications like zero-knowledge proofs and commitments in cryptographic protocols because of its flexibility and security. It is also optimized for performance and can be implemented efficiently on both CPUs and GPUs, which is crucial for the FRI protocol, where large amounts of data needs to be processed quickly in parallel. In GPU-accelerated cryptographic systems like this one, Keccak's design allows it to take full advantage of parallelism, improving the overall throughput of the protocol without sacrificing security. The open-source implementation of Keccack's SHA3-256 is made to GPU-compatible for use in our implementation.

On the performance side, libcudart (CUDA Runtime Library), among others, is crucial for GPU management in the protocol. This library facilitates interaction between the code and the GPU, managing tasks like memory allocation, kernel execution, and synchronization of GPU threads. In particular, libcudart is responsible for handling CUDA-specific operations such as transferring data between the host (CPU) and device (GPU)

and launching the parallel kernels that compute the polynomial evaluations and Merkle tree commitments.

4.1.4 Extending the Basis Elements

The domain basis in the context of the FRI protocol relates to the definition of the evaluation space over which polynomials are sampled and evaluated. In our implementation, this corresponds to the initialization of domain elements and their subsequent use in polynomial evaluations. Specifically, we handle the domain elements in data structures that resemble the Python Dictionary, that the proof utilizes to generate the field elements for its proof parameters.

For our different levels of security, L3 being for finite field $GF(2^{256})$ and L5 being for finite field $GF(2^{320})$, we start off from $GF(2^{64})$ and use polynomial computations to extend to the bigger fields.

A polynomial over the field $GF(2^{64})$ is typically represented as:

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_d x^d$$

where c_i are coefficients from $GF(2^{64})$. This base field forms the building blocks for larger extensions needed for higher security levels. For the higher security levels, such as L2 ($GF(2^{256})$) and L3 ($GF(2^{320})$), we extend the base field using polynomials derived from powers of α, along with another β and their products. The first 64 elements are generated as:

$$F_{2^{64}} =< 1, \alpha, \alpha^2, \ldots, \alpha^{63} >$$

We then extend the field by multiplying these elements with powers of β, where β is used to represent the basis elements for $\mathbb{F}_{2^3}$, as $\mathbb{F}_{2^{256}} = [\mathbb{F}_{2^{64}} \cdot \mathbb{F}_2^4]$, and $\mathbb{F}_{2^3} =< 1, \beta, \beta^2 >$. For L3, the elements span the field $GF(2^{256})$, constructed as:

$$\{1, \alpha, \cdots, \alpha^{63}, \beta, \beta\alpha, \cdots, \beta\alpha^{63}, \beta^2, \beta^2\alpha, \cdots, \beta^2\alpha^{63}, \beta^3, \beta^3\alpha, \cdots, \beta^2\alpha^{63}\}.$$

For L5, representing $GF(2^{320})$, the polynomial is further extended using higher powers of $var2$:

$$\{1, \alpha, \cdots, \alpha^{63}, \beta, \beta\alpha, \cdots, \beta\alpha^{63}, \beta^2, \beta^2\alpha, \cdots, \beta^2\alpha^{63}, \beta^3, \beta^3\alpha, \cdots, \beta^2\alpha^{63}, \beta^4, \beta^4\alpha, \cdots, \beta^4\alpha^{63}\}.$$

The scalability of these fields ensures that the proof system maintains its robustness and efficiency, even as the field grows in size, making the process both straightforward and secure for different levels of security requirements.

4.1.5 Borrowing Preon's [Che+23] Parameters

To make the protocol work, our code uses the FRI Domain parameters, finite field functions, and Polynomial evaluation functions as used in Preon's Optimised Implementation, which is available as in the public domain. These functions were made compatible with the GPU nodes using the **__device__** and **__host__** function qualifiers. These qualifiers specify where the function can be executed and where it can be called from.

The FRI domain parameters list the basis and offset (shift) elements that are used to compute the field elements. These parameters are crucial in determining the protocol's rounds, and for constructing the codeword elements that aids the process of the entire protocol. They define multiple FRI domains, each with a progressively smaller size, from 2^{20} down to 2^5, reflecting the recursive nature of the FRI protocol. Each domain is initialized with a specific set of bases elements, providing the foundation for interpolation and query operations that underpin the proof's correctness. The parameters also specify the number of co-linearity tests, which indicate the number of points that the verifier queries to check the codeword computation on. These parameters also allow for pre-computing the inverses of these bases, which is one of the optimizations that we implement in the protocol's software.

4.2 Proof Management

In this implementation of the FRI (Fast Reed-Solomon Interactive Oracle Proof) protocol, proof management plays a critical role in both the commit and verification phases. It encompasses the generation, organization, and verification of Merkle proofs, which are used to guarantee the correctness of the codeword reductions over successive rounds of the protocol. The core operations involved are rooted in both cryptographic hash functions (specifically SHA3-256) and the arithmetic over finite fields, ensuring the integrity and soundness of the FRI proof system.

The proof management begins in the `commit` phase, where codewords are progressively reduced through a series of rounds, each halving the size of the codeword while maintaining the required polynomial properties. At each round, the prover generates a Merkle tree over the current codeword and commits to the root of this Merkle tree. This commitment is pushed into the proof stream, allowing the verifier to later request and verify proofs of membership for selected elements in the codeword.

For each query during the `verify` phase, a Merkle proof path is opened. The opening consists of a sequence of hash values, which represent the sibling nodes along the path from a leaf to the root of the Merkle tree. These paths are provided to the verifier to check the consistency of the leaf value (an element of the codeword) with the previously committed Merkle root.

4.2.1 Merkle Tree Commitment and Opening

The core components of the Merkle commitment in the code involve functions like `merkle_commit`, `merkle_open`, and `merkle_verify`. During commitment, a Merkle tree is constructed from the leaves, which are the hash representations of the codeword elements. The commitment function recursively hashes sibling nodes up to the root, storing intermediate hashes for potential use in proof paths. The root of the tree is then committed to the verifier, while the tree itself is retained for opening.

The `merkle_open` function plays a crucial role in proof management, as it extracts the authentication paths for a given leaf, allowing the prover to demonstrate the correctness of the committed data. The proof paths include the necessary sibling hashes required to reconstruct the Merkle root from the leaf value. This allows the verifier to recompute the root and compare it with the committed value, ensuring that the queried leaf belongs to the committed set.

Recursion depth stack overflow error for large number of leaves

In recursive implementations, each function call consumes a portion of stack memory, and when the recursion depth exceeds the stack's limit, the system cannot allocate additional memory, leading to an invalid memory access error.

For instance, in this work, this error manifests as an EXC_BAD_ACCESS (on macOS), indicating that the program attempted to access memory beyond what was allocated to it. Specifically, when building trees with a high number of leaves, the recursive process for

node traversal or proof generation reaches the maximum stack depth, causing the program to access unallocated memory and consequently crash. This type of error highlights a significant limitation in recursive designs for memory-intensive tasks. The solution was to write an implementation that uses loops instead of recursive function calls, but to balance them in a way that the GPU kernel would still find it simplistic enough for correct parallelization.

```
push counted +1 in query for a_indices 5616
proof len b: 17
push counted +1 in query for b_indices 5624
Process 80669 stopped

- thread #1, queue = 'com.apple.main-thread', stop reason =
   EXC_BAD_ACCESS (code=1, address=0x320716ab72de2a4e)
frame #0: 0x00000001845e57f4 libsystem_platform.dylib`_platform_memmove +
    420 libsystem_platform.dylib`:
-> 0x1845e57f4 <+420>: ldr x6, [x1], #0x8
0x1845e57f8 <+424>: str x6, [x3], #0x8
0x1845e57fc <+428>: subs x2, x2, #0x8
0x1845e5800 <+432>: b.hs 0x1845e57f4 ; <+420>
Target 0: (fri) stopped.
(lldb) bt
```

4.2.2 Handling of Authentication Paths

The authentication paths (or Merkle proofs) are stored in a multi-dimensional array in the implementation. For each query, the path corresponding to the queried codeword element is extracted, and the sequence of hashes is pushed into the proof stream. The verifier retrieves the proof stream during the verification phase and validates the consistency of the proofs by recomputing the Merkle root. The proof lengths, which vary depending on the depth of the Merkle tree, are also managed as part of this process. The dynamic handling of the proof lengths ensures that the verifier knows how many sibling nodes to expect for each proof path.

4.2.3 Optimization Considerations

In the context of GPU-based FRI implementations, parallelization of both proof generation and verification phases can significantly speed up the protocol. That being said, managing the memory transfers between the host and device (CPU to GPU and vice-versa) becomes a

bottleneck, particularly when handling large Merkle trees or when proof paths are large due to high-depth trees. Optimizing memory management through careful allocation, transfer, and synchronization is essential for achieving high performance in the FRI protocol.

The implementation also focuses on robust error handling, ensuring that any inconsistencies in the proof (e.g., incorrect proof lengths or mismatches between computed and committed Merkle roots) are detected early. This is critical in cryptographic settings, where soundness and completeness must be guaranteed. The security of the FRI protocol relies on the hardness of breaking the Merkle commitment scheme, which in turn depends on the collision resistance of the underlying hash function (SHA3-256 in this case). By verifying that all queried leaves are consistent with the commitment, the verifier can trust the integrity of the proof provided by the prover.

4.2.4 Memory Management on the GPU

The FRI line computation is performed in parallel using CUDA. At each round of FRI, we reduce the size of the codeword by half, combining pairs of elements according to a linear function determined by the sampled value α.

This is computed in the commit_kernel function, where each thread processes one pair of elements from the codeword in parallel. The kernel is responsible for subtracting elements, multiplying by the pre-computed denominator inverse, and applying the linear function. The output is written to the next codeword. Below is the relevant code snippet that allocates memory to the GPU and invokes it:

```
// Allocate memory on the GPU
cudaMalloc((void**)&device_codeword, size);
cudaMalloc((void**)&device_codeword_nxt, size);
cudaMalloc((void**)&device_alpha, size);
cudaMalloc((void**)&device_offset, size);
cudaMalloc((void**)&device_denominator_inv, size);
cudaMalloc((void**)&device_eval_basis,size);

// Transfer data from host to device
cudaMemcpy(device_codeword, host_codeword, size, cudaMemcpyHostToDevice);
cudaMemcpy(device_alpha, host_alpha, size, cudaMemcpyHostToDevice);
cudaMemcpy(device_offset, host_offset, size, cudaMemcpyHostToDevice);
cudaMemcpy(device_denominator_inv, host_denominator_inv, size,
    cudaMemcpyHostToDevice);
cudaMemcpy(device_eval_basis, host_eval_basis, size,
    cudaMemcpyHostToDevice);

```

```
// Launch the kernel (assuming 1D grid and block structure)
int threadsPerBlock = 256;
int blocksPerGrid = (N / 2 + threadsPerBlock - 1) / threadsPerBlock;

commit_kernel<<<blocksPerGrid, threadsPerBlock>>>(device_codeword,
    device_codeword_nxt, device_alpha, device_offset,
    device_denominator_inv, device_eval_basis, N);

// Synchronize to ensure the kernel execution is finished
cudaDeviceSynchronize();

// Copy the result back to the host
cudaMemcpy(host_codeword_nxt, device_codeword_nxt, (N / 2) * FIELD_WORDS
    * sizeof(uint64_t), cudaMemcpyDeviceToHost);

// Clean up device memory
cudaFree(device_codeword);
cudaFree(device_codeword_nxt);
cudaFree(device_alpha);
cudaFree(device_offset);
cudaFree(device_denominator_inv);
cudaFree(device_eval_basis);
```

For each call to the kernel where the next codeword is computed, the kernel only needs so much memory to be transferred to it from the CPU to ensure the parallel computations successfully take place.

The Merkle tree integration is also parallelized. Once each round of FRI is completed, we generate the Merkle tree by hashing pairs of elements from the codeword to compute the next level of hashes. This process is repeated until we reach the root. The Merkle tree generation is encapsulated in the merkle_kernel, where each thread computes a hash for a pair of elements from the previous layer:

```
__global__ void compute_tree_layers(uint64_t *device_codeword_nxt,
    uint64_t *device_layer_hashes, uint64_t *device_tree_layer,
uint64_t *device_tree_layer_nxt, uint64_t *
    device_combined_sibling_codewords, uint64_t *
    device_concat_codeword_to_hash, uint64_t *device_digest, int N)
{
    size_t I = blockIdx.x * blockDim.x + threadIdx.x;

    if (I >= N / 2) return;

    if(I < N/2 && N < initial_codeword_length && N >=
    last_codeword_length) {
```

```
        int idx1 = 2 * I * (FIELD_WORDS + HASH_WORDS);
        int idx2 = (2 * I + 1) * (FIELD_WORDS + HASH_WORDS);
        int idx3 = I * (FIELD_WORDS + HASH_WORDS);//for next codeword
        int idx5 = I * HASH_WORDS; //for hash
        int idx4 = I * (FIELD_WORDS + HASH_WORDS);
        //step 1: combine sibling elements of current layer
        memcpy(&device_combined_sibling_codewords[idx3], &
    device_tree_layer[idx1], (FIELD_WORDS + HASH_WORDS) * sizeof(uint64_t
    ));
        memcpy(&device_combined_sibling_codewords[idx3 + (FIELD_WORDS +
    HASH_WORDS)], &device_tree_layer[idx2], (FIELD_WORDS + HASH_WORDS) *
    sizeof(uint64_t));
        //step 2: hash them
        SHA3((uint8_t *)&device_digest[idx5], (uint8_t *)&
    device_combined_sibling_codewords[idx3], 2 * (FIELD_WORDS +
    HASH_WORDS) * sizeof(uint64_t), 256);
        //step3: combine the hash with the codeword of the next layer
        memcpy(&device_concat_codeword_to_hash[idx4], &device_digest[idx5
    ], HASH_WORDS * sizeof(uint64_t));
        memcpy(&device_concat_codeword_to_hash[idx4 + HASH_WORDS], &
    device_codeword_nxt[idx3], FIELD_WORDS * sizeof(uint64_t));
        //step 4: save computation in the tree's data structure
        memcpy(&device_tree_layer_nxt[idx3], &
    device_concat_codeword_to_hash[idx4], (FIELD_WORDS + HASH_WORDS) *
    sizeof(uint64_t));

    }
}
```

This kernel ensures the Merkle tree is constructed efficiently on the GPU by processing multiple pairs of elements simultaneously.

4.2.5 Proof Stream and Verification

The proof stream is central to both the prover's and verifier's interactions in the protocol. The prover pushes commitments, authentication paths, and other necessary data into the proof stream. The verifier, on the other hand, pulls this data during the verification process. The code snippet managing the proof stream is based on functions like `push_object` and `pull_object`, which manage the serialization and deserialization of Merkle proof components. Correct memory handling, particularly in dynamic proof paths, is crucial for ensuring that all parts of the proof are accessible and verifiable by the verifier.

In the verification phase, once the proof paths are retrieved, the verifier reconstructs the Merkle root using the received leaf and proof path. This is done by successively hashing sibling nodes along the path and checking if the resulting root matches the previously committed root. The verifier must handle multiple queries and check the corresponding proof paths for all queried elements, ensuring that all elements are consistent with the original commitment. To make the protocol non-interactive, we employ a custom serialization method, our version of the Fiat-Shamir heuristic. The prover pushes objects into a global proof stream, including codewords, Merkle roots, and the sampled values (α), while the verifier pulls these objects in the correct order. The key limitation of this approach in C is that our serializer does not recognize object data types, so the pattern of pushing and pulling must exactly match. Here is an example of pushing and pulling data objects into the proof stream in the commit function:

```
push_object(global_proof_stream, root);
push_object(global_proof_stream, sampled_alpha_for_proofstream);
push_count++;

uint64_t *root_verify = (uint64_t *)pull_object(global_proof_stream);
sampled_alpha = (uint64_t *)pull_object(global_proof_stream);
pull_count++;
```

The idea of implementing serialization in this way was to perform a small-scale mimic of the Pickle module [Pyt23] in Python. Since C does not provide introspection or dynamic typing, ensuring correctness in serialization in our code depends heavily on maintaining the correct order of push and pull operations, as mismatches will lead to undefined behavior. There are further security limitations to such an implementation. When handling sensitive cryptographic proofs such as this, improper serialization can inadvertently expose intermediate computations or metadata about the proof stream, thereby weakening the zero-knowledge properties of the protocol when further deployed as part of bigger algorithms. For example, metadata about the sizes or structures of serialized objects could reveal information about underlying codeword, alpha or Merkle root lengths, which could be exploited in targeted attacks on the FRI protocol's soundness or completeness. We are aware of this limitation and note the encryption of serialized proof stream as a future work.

The overall structure ensures that all the necessary components—codewords, sampled element, indices, Merkle roots, verification paths, and proof objects are serialized for transmission to the verifier, enabling for a non-interactive proof system.

Chapter 5

Performance of Air-FRI

In this chapter, we highlight and discuss the performance of our implementation. Mainly, the chapter aims to bring to light the various metrics for our performance testing, how much time the prover and verifier take for their respective execution, and a bit about the memory management that went into the code.

5.1 Test Plan

This section outlines the test plan employed for evaluating the performance and correctness of the Fast Reed-Solomon Interactive Oracle Proof of Proximity (FRI) protocol. The tests are divided into two main categories: unit tests to verify functional correctness and performance tests to assess execution times under various optimization levels.

5.1.1 Unit Tests

The unit tests are designed to validate the correctness of individual modules within the FRI protocol implementation. The tests check for expected behaviour in both CPU and GPU implementations. The following functionalities are tested:

To ensure the accuracy, robustness, and correctness of each component within the FRI protocol implementation, a comprehensive suite of unit tests was developed. These tests cover core functionalities, including finite field operations, cryptographic commitments, polynomial evaluations, and optimizations. The following sections describe the specific areas tested.

- **Finite Field Operations:** Unit tests for finite field operations are fundamental to verifying the correct behaviour of addition, multiplication, inversion, and other arithmetic functions in $GF(2^{256})$. These operations underpin many higher-level processes in the protocol. Tests validate the following:

    1. **Addition and Multiplication:** Ensures that addition and multiplication conform to the field's properties, specifically closure, associativity, distributivity, and the existence of additive and multiplicative identities.
    2. **Inversion:** Verifies that every non-zero element has a unique multiplicative inverse within the field, a crucial aspect for line computations within the protocol.

- **FRI Commit Phase:** The commit phase of the FRI protocol involves multiple steps, including codeword reduction, Merkle tree generation, and proof commitment. Unit tests are designed to validate:

    1. **Codeword Reduction:** Verifies that each round reduces the codeword's size as expected and that the correct values are propagated through each reduction step.
    2. **Merkle Tree Generation:** Tests the Merkle tree construction by verifying that the tree is built correctly for a given set of codewords. Each node, from leaves to the root, is checked to ensure that it is the cryptographic hash of its child nodes.
    3. **Proof Commitment:** Validates that the Merkle root produced in this phase matches the expected root for the set of codewords. This ensures that the prover's commitment is consistent and that no alteration of data occurs before passing it to the verifier.

- **FRI Verify Phase:** This phase focuses on the verifier's ability to correctly validate proofs generated during the commit phase. Tests in this category include:

    1. **Merkle Proof Verification:** Ensures that the verifier can accurately validate a Merkle proof by reconstructing the path to the root from any leaf. The test checks for correctness in verifying sibling hashes and recomputing the root.
    2. **Consistency Checks:** Validates the verifier's ability to check the consistency of line reductions and codeword transformations. This includes ensuring that each round's output matches the expected values given the proof and the protocol parameters.

- **Polynomial Evaluation:** Polynomial evaluations are performed at various domain points throughout the FRI protocol. Unit tests for this component ensure:

    1. **Correctness of Evaluation:** Validates that the polynomial evaluation function correctly computes values at specified domain points.
    2. **Consistency Across Rounds:** Ensures that evaluations are consistent across rounds of the protocol and that transformations between rounds adhere to expected outcomes.
    3. **Edge Cases:** Tests boundary conditions, such as evaluations at zero or other special points in the domain, ensuring that calculations remain stable and accurate.

- **Precomputed Inverses and Non-Interactiveness:** Optimizations like precomputed inverses and the Fiat-Shamir transform are essential in our contribution for efficiency. Unit tests for these optimizations verify:

    1. **Precomputed Inverses:** Confirms that precomputed inverse values are correctly utilized in the protocol, minimizing redundant calculations and ensuring consistency with direct inverse calculations.
    2. **Fiat-Shamir Transform (Non-Interactiveness):** Validates that the Fiat-Shamir transform correctly replaces interactive proof steps by generating challenge values deterministically from hash values, thus maintaining protocol integrity without external randomness. Ofcourse, this code also takes care of the proof system and all the objects that the prover wishes to send to the verifier.

- **Merkle Tree Code:** The Merkle tree is central to cryptographic commitments in the FRI protocol. It is not an exaggeration to say that this part of the code is the most sophisticated of the entire protocol, with great demand on correct memory management and root computation. Unit tests for the Merkle tree code could ensure some features were correct, but the real issues were captured with actual parameters. Some unit tests done for the Merkle tree code include:

    1. **Merkle Tree Construction:** Confirms that the Merkle tree is built correctly from the provided data, verifying each node's value as the hash of its child nodes and ensuring a correct path from leaves to root.
    2. **Hash Consistency:** Tests that the hash function consistently produces the same output for identical inputs across all nodes, ensuring no discrepancies in the cryptographic commitments.

3. **Authentication Path Generation (Proof Path):** Ensures that the function generating Merkle proofs produces correct paths from any leaf node to the root, including each intermediate hash in the correct order.
4. **Verification of Proof Paths:** Validates that given a Merkle proof path, the verifier can reconstruct the correct root by hashing along the path. This ensures that proof paths provided in the protocol are trustworthy and complete.
5. **Edge Cases and Boundary Conditions:** Checks special cases such as single-leaf trees, maximum tree depth, and scenarios where sibling nodes are identical, ensuring that the Merkle tree functions remain reliable under all conditions.

For the sake of unit tests, testing our Verifiable FRI Merkle Tree was limited to basic functionality. Most essential testing was done with actual parameters in terms of having it work correctly for the protocol.

All unit tests ensure that the outputs match the theoretical expectations, and no undefined behavior or memory issues (such as segmentation faults) occur.

5.1.2 Performance Test Design

The performance tests are divided into two categories: CPU-based tests and GPU-accelerated tests.

1. **CPU-based tests:** The performance of the C-based FRI protocol is measured for baseline comparison. Initial tests use no optimizations, followed by tests with precomputed inverses, repeated hashes (Fiat-Shamir), and FRI Merkle tree implementation.
2. **GPU-accelerated tests:** Performance is evaluated on the GPU version of the code. The tests include precomputed inverses and parallel line computation, with the final implementation featuring FRI Merkle tree optimizations. These tests assess how GPU parallelism improves the runtime of each phase of the protocol.

Each test is run multiple times (up to 100 iterations) to ensure the consistency and reliability of the timing results.

5.1.3 Test Environment

The tests were conducted on a system with the following configuration:

- **CPU:** Apple M3 (ARM Processor), 8GB Unified Memory
- **CPU cores:** 8
- **GPU:** NVIDIA 40 Ampere
- **NVIDIA Architecture-based CUDA cores:** 10,752 [NVI20]
- **RAM (GPU):** 48GB VRAM

The time was measured in seconds, and the system clock was used for recording execution times.

5.1.4 Test Metrics

The performance tests for the FRI protocol implementation are designed to evaluate and compare the execution times across various optimizations and implementations. The following metrics will be measured for both CPU and GPU-based implementations:

1. **Prover Time:** This measures the total time taken by the prover to commit to a proof, including codeword reductions, Merkle tree generation, and all relevant hash computations.
2. **Verifier Time:** This measures the time taken by the verifier to validate the proof, including Merkle proof verification, co-linearity tests, and consistency checks.
3. **Average Time:** This metric records the average time over multiple test iterations to ensure consistency and mitigate any potential outliers in performance data. The goal is to establish a reliable baseline for each implementation.
4. **Minimum Time:** This represents the best-case scenario of the prover and verifier, i.e., the fastest recorded execution time during the test iterations.
5. **Maximum Time:** This represents the worst-case scenario, i.e., the slowest recorded execution time during the test iterations.
6. **Speedup Factor:** For GPU-accelerated implementations, this metric compares the prover and verifier times between the GPU and the CPU versions, calculating the performance gains achieved through parallelization in percentage.

5.2 Results

This section presents the results for various parameters and briefly compares them with existing implementations.

5.2.1 Performance of the Code Run with L3 Parameters

The code for our implementation is majorly written for Preon's [Che+23] L3 parameters, as described in table 5.1. The low-degree test was conducted for 2^{12} in finite field 2^{256}. Managing memory for these parameters is relatively convenient, because the codeword element size and Merkle tree hash size at each index of the tree are the same, *i.e*, 256 bits.

Parameter	**Value**
Finite Field (L3)	2^{256}
Degree Tested	2^{12}
Expansion Factor	32
Least Codeword Length	2^5 (32)
Initial Codeword Length	2^{17} (131072)
Number of Co-linearity Tests	14
Number of Codeword Reductions (Rounds)	12
Merkle Tree Height	17
Hash Function Input $\rightarrow$ Hash Function Output	1024 $\rightarrow$ 256

Table 5.1: Parameters for L3 FRI Protocol

Table 5.1 shows the parameters for running the protocol as given by Preon [Che+23] for L3 security. Apart from these metrics, these security levels also differ in handling memory while programming in C or CUDA-C. Specifically, L3 parameters are 256-bit, so in C, each codeword element is handled as `4 * sizeof(uint64_t)`, where `uint64_t` is of 8 bytes in length. For L5, each codeword element is 320-bit long, represented as `5 * sizeof(uint64_t)`. This required careful allocation and de-allocation of memory and also demanded handling of codeword arrays in 3D.

Table 5.2 shows the progressive decrease in Average Time taken to run the protocol for the same parameters with each optimization implementation. Rows 1, 2, 3 show the times

Table 5.2: Performance Comparison of the FRI protocol on CPU vs GPU for L3 Parameters

S.No.	Category	Iterations	Average (s)	Max (s)	Min (s)
1	C Code - No Optimisations	100	22.516272	24.487644	21.283454
2	C Code - Precomputed Inverses & Fiat-Shamir	100	17.949764	18.285129	17.674678
3	C Code - Precomputed Inverses, Non-interactiveness, & Verifiable FRI Merkle Tree	100	13.466491	14.387460	12.974454
4	GPU Code - Precomputed Inverses, Non-interactiveness, Parallel Line Computation	100	9.355615	10.225553	9.138932
5	**GPU Code - Parallel Line Computation & Verifiable FRI Merkle Tree**	**100**	**1.490716**	**1.626001**	**1.459887**

taken by the implementation in C, while 5 and 6 clearly show the advantages of our major and novel optimizations using the GPU.

For the implementation of the protocol for the low-degree test of 4096, we measure the individual prover and verifier times over 10 iterations, the results of which are shown in table 5.3.

Metric	Prover Time (s)	Verifier Time (s)
Minimum	0.66909	0.16506
Maximum	0.67360	0.16651
Average	0.67026	0.16586

Table 5.3: Minimum, Maximum, and Average Times for Prover and Verifier for degree 2^{12} over 10 iterations

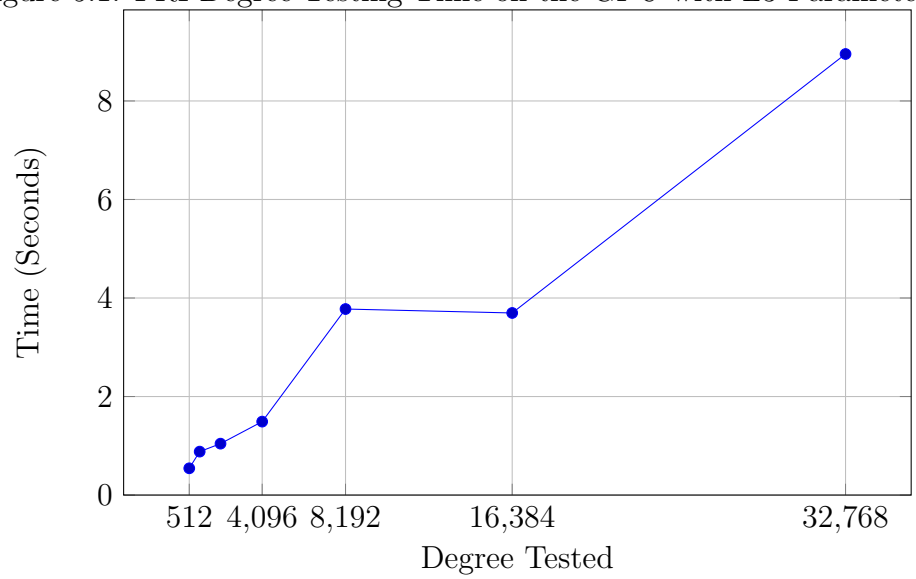
The graph 5.2 below shows the times for various degrees as tested on the GPU. One of the implications of an uneven curve could be the fact that time efficiency is also dependent on the number of threads and blocks that a GPU kernel is allocated before invocation. This is in-turn dependent on the length of the input, and how the workload is distributed across threads and blocks. Effective scheduling and memory coalescing techniques can reduce bottlenecks in such GPU workloads by ensuring that memory access is streamlined across multiple threads, thus improving throughput for numerically-driven computations.

Based on the CPU and GPU times for the same degrees, we measure the percentage in performance efficiency termed as the 'Speedup Factor' (Table 5.4). Since the degree test for 2^{12} is the main focus of our implementation, we consider the speedup factor of this degree as the benchmark for our work.

5.2.2 Performance on L5 Parameters

For L5 security, Air-FRI's results show notably seminal performance despite large codeword lengths to begin the protocol with. The parameters are shown in table 5.5.

Figure 5.1: FRI Degree Testing Time on the GPU with L3 Parameters



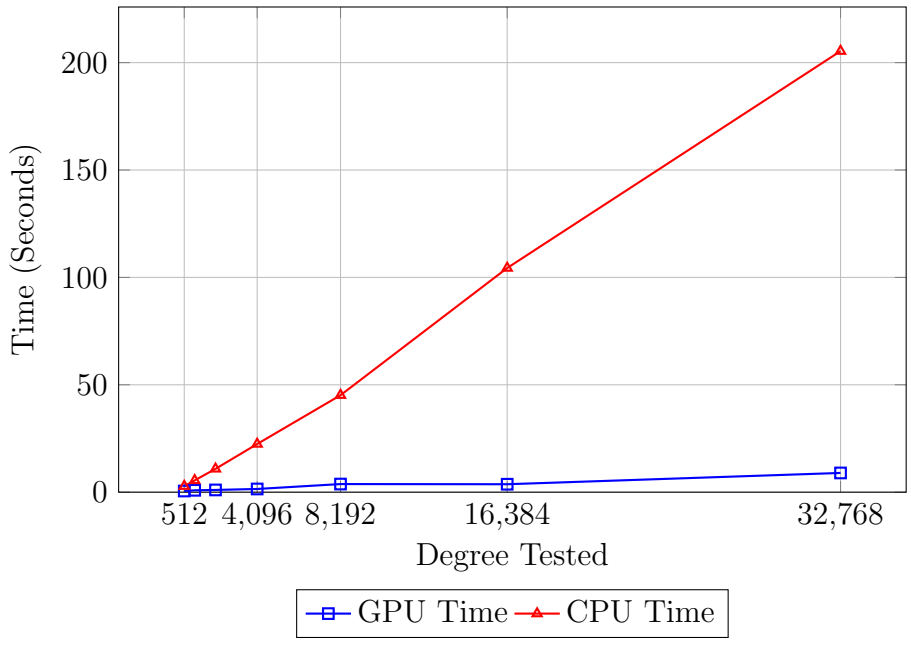


Figure 5.2: CPU vs GPU Execution Times for FRI Degree Testing for L3 Parameters

Degree	CPU Time (s)	GPU Time (s)	Speedup Factor (%)
512	2.798574	0.543291	80.59%
1024	5.476513	0.881827	83.90%
2048	10.856021	1.043323	90.39%
4096	**22.425451**	**1.490716**	**93.35%**
8192	45.155425	3.776132	91.64%
16384	104.365670	3.696402	96.46%
32768	205.366330	8.953130	95.64%

Table 5.4: Speedup Factor (%) of GPU over CPU for Various Degrees for L3 Parameters

Parameter	Value
Finite Field (L5)	2^{320}
Degree Tested	2^{16}
Expansion Factor	32
Least Codeword Length	2^5 (32)
Initial Codeword Length	2^{21} (2097152)
Number of Co-linearity Tests	10
Number of Codeword Reductions (Rounds)	16
Merkle Tree Height	21
Hash Function Input $\rightarrow$ Hash Function Output	1152 $\rightarrow$ 256

Table 5.5: Parameters for L5 FRI Protocol

Tested on the metrics given above in table 5.5. Table 5.6 shows the prover and verifier times measured over 10 iterations on the GPU. As compared to L3's results, it is shown that the proof generation takes only a few seconds more even when the codeword constraints are large. This shows significant promise in realizing our implementation in post-quantum secure algorithms which require such large constraints and strong security levels.

5.3 Performance of Current FRI Implementations

To compare our test results with the existing FRI implementations, which are currently compatible to be run only on the CPU, we ran the pre-written tests of the libiop library [Sc24], which Aurora [Ben+19], Ligero [Ame+17], and Fractal [COS20] use to run their

Metric	Prover Time (s)	Verifier Time (s)
Minimum	13.85859	1.12372
Maximum	13.88316	1.12699
Average	13.87129	1.12488

Table 5.6: Minimum, Maximum, and Average Times for Prover and Verifier for degree 2^{16} over 10 iterations on the GPU

entire protocol. The tests are generated with the library's build and metrics like the vector size and subgroup size **are much lesser** as compared to those tested for in our implementation. Table 5.7 shows the times for the three algorithms that use the libiop library for their low-degree test implementation. Further, to compare the entire performance, we must also consider the Merkle tree test times for these algorithms as shown in table 5.8.

Table 5.7: Performance Comparison of Existing FRI Implementations on the CPU

Implementation	Prover Time (s)	Verifier Time (s)	Vector Size	Subgroup Size
Aurora (libiop)	0.077	0.006	2^9	2^{12}
Fractal (libiop)	0.071	0.009	2^9	2^{12}
Ligero (libiop)	0.049	0.022	2^9	2^{12}

Table 5.8: Merkle Tree Test Summary for Libiop Implementations

Test Case	Execution Time (s)	Verification Time
Merkle Tree	1.008	1 second per index in one tree

Note: Preon's [Che+23] FRI implementation is tested for a large polynomial of degree 2^{12} with a subgroup size of 2^{17}, while the Aurora, Fractal, and Ligero implementations from the libiop library are tested for a smaller polynomial degree of 2^9 within a subgroup of 2^{12}. In scale, when the size of the extension field exponentially increases, the prover time approximately doubles (see 5.1). This is why, although Preon calls Aurora for its inner working, including for running the FRI Low-Degree Test, the time taken by Preon's code is much greater to account to the exponential growth in codeword sizes and to cater to parameters of larger extension basis. This is shown in table 5.9 below.

These timings on the CPU shown in table 5.9, when compared with our implementation the GPU, has significant performance improvements through our optimizations. This shows promise that our software-oriented solution for accelerating the FRI protocol can serve

Metric	Prover Time (seconds)	Verifier Time (seconds)
Minimum	126.926	0.803
Maximum	139.683	1.167
Average	129.248	0.875

Table 5.9: Preon's Prover and Verifier Times for Testing of Degree 2^{12} over 50 Iterations on the CPU

algorithmic efficiency when carefully deployed in those zk-SNARK schemes which contain the FRI as one of their core components.

Chapter 6

Conclusions and Future Work

This chapter presents a summary of the thesis and future work of the implementations listed in Chapter 3.

6.1 Summary of Thesis

In this thesis, novel and efficient optimizations of the FRI protocol for low-degree testing of polynomials are implemented in software on the GPU, and results are presented. The parameters for implementation (specifically, the domain elements) are borrowed from Preon's [Che+23] parameter values. We started off with addressing basic algorithmic changes that would save prover computational time and make the protocol non-interactive. These include the pre-computation of inverses of domain bases elements, which are parameters involved in the computation of the next codeword element in an RS-codeword-based proof system. Then, we implement a one-time sampling of a uniformly-random element on the prover side that is used in the next codeword elements' computation as well. This sampled element is sent to the prover, who, at a later time, can repeatedly hash while performing the co-linearity check on a queried index.

Our two major novel contributions optimize the performance of the (non-interactive) FRI protocol by computing the codeword elements in parallel on the GPU using NVIDIA's CUDA-C programming language. We then also implement a unique verifiable Merkle tree that integrates the codeword elements in order to reduce the proof size. This tree is also computed in parallel on a GPU Kernel, along with the line computation for next codeword elements.

We test the performance of these optimized implementations against the non-optimized version of the FRI protocol written to be executed on the CPU. For codeword elements computed on finite field $GF(2^{256})$, we find that the GPU version of the code is 93.3% more efficient than its CPU counterpart, which takes 22 seconds on average to execute, while the protocol on the GPU takes 1.49 seconds on average. For codeword computations on finite field $GF(2^{320})$, i.e, for L5 security parameters, the prover time measured is 13.89 seconds, which is approximately only 11 seconds slower than parameters of L3 despite having significantly large codeword sizes and greater number of reductions. In all cases, it is proven that the verifier takes significantly lesser time than the prover, as is expected, since prover does all the hard work to make sure verification is efficient.

This work aims to highlight the importance of identifying the underlying optimizations in standardized cryptographic algorithms in software in order to speed up the implementations. This implementation can be integrated into any algorithm or protocol that utilizes the FRI protocol as one of its core components, although it was primarily written for integration into the software for Polaris [FG22].

6.2 Limitations

An essential security assumption, which is noted as a limitation, is that our implementation does not account for software-induced vulnerabilities in the protocol. By that, we refer specifically to the fact that the code might still be prone to side-channel attacks which the code is not assigned to address at this point. Further, the proof system manages several objects of varied memory sizes. In C, this memory is allocated dynamically and freed to avoid memory leaks, but there could always be certain instances where more memory is allocated than utilized. In that case, memory might be mismanaged. To address memory safety while still reaping the advantages of GPU's performance, the protocol can also de developed in Rust using Rustacuda [The24]. Another final identified limitation is that the implementation of the integrated Merkle tree does not ease on-the-fly Merkle authentication path computation. This means that the prover would have to either store the entire Merkle tree in order to compute the authentication path, or it would have to gain access to all codewords of all FRI rounds during the query phase. This implies more complex memory management to avoid leaks and segmentation faults.

6.3 Future Work

6.3.1 Further Reduction of the Proof Size

For a finite field byte size of 256, the current implementation approximately shares 9,000 objects to the verifier via the proof stream, majority of which include the co-linearity test objects for all the rounds. To reduce this number is definitely in the scope of future work to ensure that the effective proof size further reduces. Possibly a trade-off can be made by increasing the localization parameters and reducing the number of co-linearity tests performed to still ensure correctness of the protocol.

6.3.2 Integrating with a Faster Implementation of the FFT Algorithm

Polaris [FG22] uses additive FFT to evaluate the polynomial over interpolated points, essentially acting as a predecessor for the FRI protocol which uses this polynomial to perform low-degree testing on. Currently, the code uses Preon's [Che+23] implementation of FFT, but using a special basis and subfield structures [Gon24a], it would be possible to implement an FFT algorithm that is more efficient with the polynomial evaluation on any given domain.

6.3.3 Implementing Zero-Knowledge with this FRI Protocol

In order to integrate this code to work with any zk-SNARK-based algorithm, we would have to implement the zk aspect by masking the polynomial whose degree is being tested. For example, in each round of the FRI protocol, as the degree of the polynomial is progressively reduced, a masking polynomial can be added at each reduction step. This additive masking technique ensures that the verifier cannot infer any specific properties of the committed polynomial beyond the fact that it lies within the specified degree. The masked polynomial is generated in such a way that its terms cancel out in the overall proof verification, thereby maintaining the proof's integrity while concealing sensitive data. The use of masking polynomials aligns well with our GPU-accelerated FRI implementation, as they can be generated on-the-fly and in parallel.

6.3.4 Encryption of Serialized Data

As described in Section 4.2.5, the serialized data that the prover shares to the verifier is currently unencrypted. The logic for the serialized Proofstream, although encapsulated, is still vulnerable to a side-channel attack (which is presented in section 6.2 as being out-of-scope of our work). While the best way to secure this data for the FRI Protocol is not evident at the moment, during the integration of Zero-knowledge with the protocol, steps to conceal the Proofstream data from illegal access is noted as an essential part of our future work.

6.3.5 Comparison of Energy Consumption between CPU and GPU Implementations of the Protocol

While GPUs make the protocol run faster, attention needs to be drawn towards the amount of energy they consume to estimate the ecological implications of this implementation. Static analysis results prove that uncoalesced global memory accesses are identified as a major factor influencing energy use in NVIDIA GPUs [Che+11]. Further, close to half of the total energy consumption of a 100A NVIDIA GPU is that of a static kernel [Del+24]. Identifying trade-offs between efficiency and energy by measuring the energy consumed by our code on the GPU versus on the CPU is identified as an important work for the future.

6.3.6 Memory Profiling of the GPU Code

Since the protocol's performance relies heavily on shared memory between the host (CPU) and device (GPU kernel), a robust implementation should incorporate memory profiling results to make optimization decisions about thread and block usage, ensuring efficient GPU kernel launches across varying codeword lengths and Merkle Tree computations. Although our current implementation does some benchmarking to figure out which part of the code takes the most time to execute, a GPU-oriented memory profiling using nvprof or nsys profile [NVI24] tools provided within the cuda-toolkit.

References

[Ame+17] Scott Ames, Carmit Hazay, Yuval Ishai, et al. "Ligero: Lightweight sublinear arguments without a trusted setup". In: *Proceedings of the 2017 acm sigsac conference on computer and communications security*. 2017, pp. 2087–2104.

[ANV13] Brian Akins, Jeffrey Ng, and Rodrigo S. Verdi. "The use of blockchain for public services". In: *Journal of Public Economics* (2013).

[Aro+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, et al. "Proof verification and the hardness of approximation problems". In: *Journal of the ACM (JACM)* 45.3 (1998), pp. 501–555.

[AS98] Sanjeev Arora and Shmuel Safra. "Probabilistic checking of proofs: A new characterization of NP". In: *Journal of the ACM (JACM)* 45.1 (1998), pp. 70–122.

[ATM21] Sidra Aslam, Aleksandar Tošić, and Michael Mrissa. "Secure and Privacy-Aware Blockchain Design: Requirements, Challenges and Solutions". In: *Journal of Cybersecurity and Privacy* 1.1 (2021), pp. 164–194. DOI: 10.3390/jcp1010009.

[Bab85] László Babai. "Trading Group Theory for Randomness". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC)*. 1985, pp. 421–429.

[Bad+24] Mohammadtaghi Badakhshan, Guiwen Luo, Tanmayi Jandhyala, et al. "Ursa Minor: The Implementation Framework for Polaris". In: *International Workshop on the Arithmetic of Finite Fields (WAIFI)*. Accepted for publication. Ottawa, Canada, June 2024.

[Bc17] Eli Ben-Sasson and contributors. *libSTARK: A library for zero knowledge (ZK) scalable transparent argument of knowledge (STARK)*. https://github.com/elibensasson/libSTARK. Accessed: October 13, 2024. 2017.

[BCS16a] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. *Interactive Oracle Proofs*. Cryptology ePrint Archive, Paper 2016/116. 2016. URL: https://eprint.iacr.org/2016/116.

[BCS16b] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. "Interactive oracle proofs". In: *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*. Springer. 2016, pp. 31–60.

[Ben+14a] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474.

[Ben+14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, et al. "Succinct non-interactive zero knowledge for a von Neumann architecture". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 781–796. ISBN: 9781931971157.

[Ben+16] Eli Ben-Sasson, Alessandro Chiesa, Ariel Gabizon, et al. "Short Interactive Oracle Proofs with Constant Query Complexity, via Composition and Sumcheck." In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 324.

[Ben+18a] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, et al. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity". In: *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. Ed. by Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, et al. Vol. 107. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 14:1–14:17. URL: https://doi.org/10.4230/LIPIcs.ICALP.2018.14.

[Ben+18b] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, et al. "Scalable, Transparent, and Post-Quantum Secure Computational Integrity". In: *Advances in Cryptology – CRYPTO 2018*. 2018, pp. 201–234.

[Ben+18c] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, et al. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Paper 2018/046. 2018. URL: https://eprint.iacr.org/2018/046.

[Ben+18d] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, et al. "Scalable, transparent, and post-quantum secure computational integrity". In: *Cryptology ePrint Archive* (2018).

[Ben+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, et al. "Aurora: Transparent succinct arguments for R1CS". In: *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*. Springer. 2019, pp. 103–128.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. "Non-interactive zero-knowledge and its applications". In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1988, pp. 103–112. ISBN: 0897912640. DOI: 10.1145/62212.62222. URL: https://doi.org/10.1145/62212.62222.

[BG90] Mihir Bellare and Shafi Goldwasser. "New Paradigms for Digital Signatures and Message Authentication Based on Non-Interactive Zero Knowledge Proofs". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. Springer New York, 1990, pp. 194–211. ISBN: 978-0-387-34805-6.

[Bit+12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, et al. "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. Association for Computing Machinery, 2012, pp. 326–349. ISBN: 9781450311151. DOI: 10.1145/2090236.2090263. URL: https://doi.org/10.1145/2090236.2090263.

[BKP14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. "Deanonymisation of Clients in Bitcoin P2P Network". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Association for Computing Machinery, 2014, pp. 15–29. ISBN: 9781450329576. DOI: 10.1145/2660267.2660379. URL: https://doi.org/10.1145/2660267.2660379.

[Bon+20] Dan Boneh, Justin Drake, Ben Fisch, et al. "Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme". In: *Cryptology ePrint Archive* (2020).

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. "How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios". In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Springer Berlin Heidelberg, 2012, pp. 626–643. ISBN: 978-3-642-34961-4.

[BS08] Eli Ben-Sasson and Madhu Sudan. "Short PCPs with polylog query complexity". In: *SIAM Journal on Computing* 38.2 (2008), pp. 551–607.

[BT19] Alex Biryukov and Sergei Tikhomirov. "Deanonymization and linkability of cryptocurrency transactions based on network analysis". In: *2019 IEEE European symposium on security and privacy (EuroS&P)*. IEEE. 2019, pp. 172–184.

[Bün+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, et al. "Bulletproofs: Short proofs for confidential transactions and more". In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 315–334.

[Che+11] Jianmin Chen, Bin Li, Ying Zhang, et al. "Statistical GPU power analysis using tree-based methods". In: *2011 International Green Computing Conference and Workshops*. 2011, pp. 1–6. DOI: 10.1109/IGCC.2011.6008582.

[Che+23] Ming-Shing Chen, Yu-Shian Chen, Chen-Mou Cheng, et al. "Preon: zk-SNARK based Signature Scheme". In: (2023).

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. "Fractal: Post-quantum and transparent recursive proofs from holography". In: *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*. Springer. 2020, pp. 769–793.

[CT65] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[Del+24] Paul Delestrac, Jonathan Miquel, Debjyoti Bhattacharjee, et al. "Analyzing GPU Energy Consumption in Data Movement and Storage ". In: *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE Computer Society, July 2024, pp. 143–151. DOI: 10.1109/ASAP61560.2024.00038. URL: https://doi.ieeecomputersociety.org/10.1109/ASAP61560.2024.00038.

[Fen+19] Qi Feng, Debiao He, Sherali Zeadally, et al. "A survey on privacy protection in blockchain system". In: *Journal of Network and Computer Applications* 126 (2019), pp. 45–58. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2018.10.020. URL: https://www.sciencedirect.com/science/article/pii/S1084804518303485.

[FG22] Shihui Fu and Guang Gong. "Polaris: transparent succinct zero-knowledge arguments for R1CS with efficient verifier". In: *Proceedings on Privacy Enhancing Technologies* (2022).

[FS87] 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.

[FT15] George Foroglou and Aphrodite-Leda Tsilidou. "Further applications of the blockchain". In: *Proceedings of the 12th International Conference on Business Information Systems*. 2015.

[Gen+12] Rosario Gennaro, Craig Gentry, Bryan Parno, et al. *Quadratic Span Programs and Succinct NIZKs without PCPs*. Cryptology ePrint Archive, Paper 2012/215. 2012. URL: https://eprint.iacr.org/2012/215.

[GMR89] 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. eprint: https://doi.org/10.1137/0218012. URL: https://doi.org/10.1137/0218012.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that Yield Nothing But Their Validity". In: *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS)* (1986), pp. 174–187.

[GO94] Oded Goldreich and Yair Oren. "Definitions and properties of zero-knowledge proof systems". In: *Journal of Cryptology* 7.1 (1994), pp. 1–32.

[Gon23] Guang Gong. *FRI*. Unpublished note. Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. Dec. 2023.

[Gon24a] Guang Gong. "Uni/multi variate polynomial embeddings for zkSNARKs". In: *Cryptography and Communications* (2024), pp. 1–32.

[Gon24b] Guang Gong. *Verifiable Integrated Merkle Tree for FRI*. Unpublished note. Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. Sept. 2024.

[Gra+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, et al. "Poseidon: A new hash function for {Zero-Knowledge} proof systems". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 519–535.

[Gro10] Jens Groth. "Short pairing-based non-interactive zero-knowledge arguments". In: *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer. 2010, pp. 321–340.

[Gro16] Jens Groth. "On the size of pairing-based non-interactive arguments". In: *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer. 2016, pp. 305–326.

[GWC19] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge". In: *Cryptology ePrint Archive* (2019).

[Hop+16] Daira Hopwood, Sean Bowe, Taylor Hornby, et al. "Zcash protocol specification". In: *GitHub: San Francisco, CA, USA* 4.220 (2016), p. 32.

[Kap+18] George Kappos, Haaroon Yousaf, Mary Maller, et al. "An empirical analysis of anonymity in zcash". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 463–477.

[Kec24a] Keccak Team. *Keccak Specifications Summary*. Accessed: November 11, 2024. 2024. URL: https://keccak.team/keccak_specs_summary.html.

[Kec24b] Keccak Team. *XKCP: The eXtended Keccak Code Package*. https://github.com/XKCP/XKCP. Accessed: October 13, 2024. 2016–2024.

[KKM14] Philip Koshy, Diana Koshy, and Patrick McDaniel. "An Analysis of Anonymity in Bitcoin Using P2P Network Traffic". In: *Financial Cryptography and Data Security*. Ed. by Nicolas Christin and Reihaneh Safavi-Naini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 469–485. ISBN: 978-3-662-45472-5.

[Kos+16] Ahmed Kosba, Andrew Miller, Elaine Shi, et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2016, pp. 839–858.

[Lev23] Alex Levinin. *libiop-fp2: A C++ library for IOP-based zkSNARKs over elliptic curves*. https://github.com/levanin/libiop-fp2. Accessed: October 13, 2024. 2023.

[Mic94] Silvio Micali. "CS Proofs (Extended Abstract)". In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)* (1994), pp. 436–453.

[MS77] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Elsevier, 1977.

[NB08] Satoshi Nakamoto and A Bitcoin. "A peer-to-peer electronic cash system". In: *Bitcoin* 4.2 (2008), p. 15. URL: https://bitcoin.org/bitcoin.pdf.

[Noy16] Catherine Noyes. "Blockchain security model for the next generation of IoT services". In: *International Journal of Security and Networks* (2016).

[NVI20] NVIDIA Corporation. *NVIDIA A40 Datasheet*. https://images.nvidia.com/content/Solutions/data-center/a40/nvidia-a40-datasheet.pdf. Accessed: October 23, 2024. 2020.

[NVI24] NVIDIA Corporation. *CUDA Profiler User's Guide*. https://docs.nvidia.com/cuda/profiler-users-guide/. Accessed: November 12, 2024. 2024.

[Ope24] OpenSSL Team. *OpenSSL Documentation*. Accessed: November 11, 2024. 2024. URL: https://docs.openssl.org/master/.

[PO21] Andreea-Elena Panait and Ruxandra F Olimid. "On using zk-SNARKs and zk-STARKs in blockchain-based identity management". In: *Innovative Security Solutions for Information Technology and Communications: 13th International Conference, SecITC 2020, Bucharest, Romania, November 19–20, 2020, Revised Selected Papers 13*. Springer. 2021, pp. 130–145.

[PP15] Gareth W. Peters and Efstathios Panayi. "Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money". In: *Journal of Banking Regulation* 16.3-4 (2015), pp. 217–233. DOI: 10.1057/jbr.2015.19.

[PTT13] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. "Signatures of Correct Computation". In: *Theory of Cryptography Conference*. Springer. 2013, pp. 222–242.

[Pyt23] Python Software Foundation. *pickle — Python object serialization*. 2023. URL: https://docs.python.org/3/library/pickle.html (visited on 10/19/2024).

[RRR16] Omer Reingold, Guy N Rothblum, and Ron D Rothblum. "Constant-round interactive proofs for delegating computation". In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 49–62.

[RS60] Irving S Reed and Gustave Solomon. "Polynomial codes over certain finite fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.

[RS91] Charles Rackoff and Daniel R Simon. "Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack". In: *Annual international cryptology conference*. Springer. 1991, pp. 433–444.

[RS92] Ronitt Rubinfeld and Madhu Sudan. "Self-Testing Polynomial Functions Efficiently and Over Rational Domains." In: *SODA*. 1992, pp. 23–32.

[Sc17] SCIPR Lab and contributors. *libsnark: a C++ library for zkSNARK proofs*. https://github.com/scipr-lab/libsnark. Accessed: October 13, 2024. 2012–2017.

[Sc24] SCIPR Lab and contributors. *libiop: A C++ library for IOP-based zkSNARKs*. https://github.com/scipr-lab/libiop. Accessed: October 13, 2024. 2017–2024.

[SD15] Mike Sharples and John Domingue. "The blockchain as a public good: Impact on reputation systems". In: *Proceedings of the European Conference on Digital Economy*. 2015.

[Set19] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. Cryptology ePrint Archive, Paper 2019/550. 2019. URL: https://eprint.iacr.org/2019/550.

[Sun+21] Xiaoqiang Sun, F Richard Yu, Peng Zhang, et al. "A survey on zero-knowledge proof in blockchain". In: *IEEE network* 35.4 (2021), pp. 198–205.

[The] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 10.3)*. https://www.sagemath.org.

[The24] The RustaCUDA Developers. *RustaCUDA: A safe wrapper for the CUDA Driver API*. Accessed: November 10, 2024. 2024. URL: https://docs.rs/rustacuda/latest/rustacuda/.

[UC 19] UC Berkeley RISELab. *Virgo: Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof*. https://github.com/sunblaze-ucb/Virgo. Accessed: October 13, 2024. 2019.

[YL20] Xiaohui Yang and Wenjie Li. "A zero-knowledge-proof-based digital identity management scheme in blockchain". In: *Computers & Security* 99 (2020), p. 102050. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2020.102050. URL: https://www.sciencedirect.com/science/article/pii/S0167404820303230.

[ZW15] Yuan Zhang and Jianping Wen. "An IoT electric business model based on the protocol of bitcoin". In: *2015 18th International Conference on Intelligence in Next Generation Networks* (2015), pp. 184–191.

[ZX19] Jiaheng Zhang and Tiancheng Xie. "Virgo: Zero Knowledge Proofs System without Trusted Setup". In: 2019. URL: https://api.semanticscholar.org/CorpusID:221606763.

[ZXL19] Rui Zhang, Rui Xue, and Ling Liu. “Security and privacy on blockchain”. In: *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1–34.