UNIVERSITAS INDONESIA

SKEMA KZG POLYNOMIAL COMMITMENT DALAM KONSTRUKSI ZK-SNARKS DAN IMPLEMENTASINYA

UNDERGRADUATE THESIS

MOHAMMAD FERRY HUSNIL ARIF
2106709112

FACULTY OF COMPUTER SCIENCE
COMPUTER SCIENCE STUDY PROGRAM
DEPOK
JULY 2025

UNIVERSITAS INDONESIA

KZG POLYNOMIAL COMMITMENT SCHEME ON ZK-SNARKS CONSTRUCTION AND ITS IMPLEMENTATION

UNDERGRADUATE THESIS

Submitted in partial fulfillment of the requirements for the degree
Sarjana Ilmu Komputer

MOHAMMAD FERRY HUSNIL ARIF
2106709112

FACULTY OF COMPUTER SCIENCE
COMPUTER SCIENCE STUDY PROGRAM
DEPOK
JULY 2025

PERNYATAAN ORISINALITAS

Skripsi ini adalah karya saya sendiri,

dan semua sumber yang dikutip maupun dirujuk

telah saya nyatakan dengan benar.

Nama	:	**Mohammad Ferry Husnil Arif**
NPM	:	**2106709112**
Tanda Tangan	:	
Tanggal	:	**10 Juni 2025**

HALAMAN PENGESAHAN

Skripsi ini diajukan oleh	:	
Nama	:	Mohammad Ferry Husnil Arif
NPM	:	2106709112
Program Studi	:	Ilmu Komputer
Judul Skripsi	:	Skema KZG Polynomial Commitment dalam Konstruksi zk-SNARKs dan Implementasinya

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana pada Program Studi Ilmu Komputer, Fakultas Ilmu Komputer, Universitas Indonesia.

DEWAN PENGUJI

Pembimbing 1	:	Drs. Lim Yohanes Stefanus, M.Math, Ph.D.	( )
Penguji 1	:	Amril Syalim, S.Kom., M.Eng., Ph.D.	( )
Penguji 2	:	Made Harta Dwijaksara, S.T., M.Sc., Ph.D.	( )

Ditetapkan di	:	Depok
Tanggal	:	20 Juni 2025

PREFACE

The author would like to express praise towards Allah SWT, for it is by His grace that the author can complete this final project titled ”KZG Polynomial Commitment Scheme on zk-SNARKs Construction and Its Implementation” which is a requirement to achieve the degree in Computer Science at Fakultas Ilmu Komputer, Universitas Indonesia.

The author expresses sincere gratitude to the following individuals who made this work possible:

1. Lim Yohanes Stefanus, Ph.D., final project supervisor, whose guidance and commitment to clear exposition improved the quality of this project.
2. Prastudy Fauzi, Research Fellow at Nanyang Technological University, for valuable feedback that significantly improved this work's technical accuracy.
3. Imam al-Fath, Cryptography Engineer at zkSecurity, who provided helpful guidance during the early learning journey and directed the author to essential resources in this field.
4. The faculty of the Computer Science Department whose cryptography courses provided essential foundations for understanding these advanced protocols.
5. The author's parents, for their constant support throughout the years of study. Their encouragement helped make this achievement possible.
6. Peers and friends who provided companionship, discussion, and encouragement throughout this journey.

This exposition aims to make zero-knowledge proofs accessible to undergraduate students by providing detailed mathematical foundations, concrete examples, and practical implementations. The author hopes this work serves as a useful resource for future students exploring cryptographic proof systems. Feedback and questions are welcome at `mohammad.ferry@ui.ac.id`.

Depok, 10 June 2025

Author

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama	:	Mohammad Ferry Husnil Arif
NPM	:	2106709112
Program Studi	:	Ilmu Komputer
Jenis Karya	:	Undergraduate Thesis

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Royalti Noneksklusif (*Non-exclusive Royalty Free Right*)** atas karya ilmiah saya yang berjudul:

Skema KZG Polynomial Commitment dalam Konstruksi zk-SNARKs dan Implementasinya

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyataan ini saya buat dengan sebenarnya.

Dibuat di	:	Depok
Pada tanggal	:	10 Juni 2025

Yang menyatakan

(Mohammad Ferry Husnil Arif)

ABSTRAK

Nama	:	Mohammad Ferry Husnil Arif
Program Studi	:	Ilmu Komputer
Judul	:	Skema KZG Polynomial Commitment dalam Konstruksi zk-SNARKs dan Implementasinya
Pembimbing	:	Drs. Lim Yohanes Stefanus, M.Math, Ph.D.

Zero-Knowledge Proof (ZKP) merupakan protokol kriptografi yang memungkinkan pembuktian kebenaran suatu pernyataan tanpa perlu mengungkapkan informasi tambahan selain validitas pernyataan itu sendiri. Salah satu konstruksi ZKP yang paling menjanjikan adalah *zero-knowledge succinct non-interactive arguments of knowledge* (zk-SNARKs), yang menawarkan ukuran *proof* yang ringkas dan tidak memerlukan interaksi antara *prover* dan *verifier*. Keunggulan ini membuka peluang aplikasi yang dapat menjaga privasi di berbagai bidang. Namun demikian, terdapat kesenjangan pemahaman yang cukup besar antara penjelasan konseptual umum dan pemahaman matematis mendalam yang diperlukan untuk memahami cara kerja protokol ini. Tugas akhir ini menyajikan pembahasan menyeluruh tentang skema KZG *polynomial commitment* dan menunjukkan peranannya yang fundamental dalam protokol zk-SNARKs modern, khususnya Marlin dan Plonk. Pembahasan dimulai dari dasar-dasar matematis mencakup teori *group*, *field*, *polynomial ring*, *elliptic curve*, dan *billinear pairing*, disertai contoh-contoh numerik untuk memperjelas setiap konsep. Skema KZG dibahas secara lengkap termasuk bukti keamanan dan berbagai teknik optimisasi. Selanjutnya dianalisis bagaimana Marlin dan Plonk memanfaatkan KZG untuk menghasilkan sistem dengan *universal and updatable structured reference strings*, masing-masing dengan pendekatan *arithmetization* yang berbeda sesuai tujuan desainnya. Implementasi kedua protokol dalam *SageMath* disediakan untuk menghubungkan teori dengan praktik. Dengan pendekatan sistematis dan contoh-contoh perhitungan pada *finite field*, tugas akhir ini dapat menjadi referensi pembelajaran bagi mahasiswa S1 Ilmu Komputer dan Matematika yang ingin memahami dan mengembangkan teknologi ZKP lebih lanjut.

Kata kunci:
KZG *polynomial commitment*, *zero-knowledge proofs*, *zk-SNARKs*, Marlin, Plonk

ABSTRACT

Name	:	Mohammad Ferry Husnil Arif
Study Program	:	Computer Science
Title	:	KZG Polynomial Commitment Scheme on zk-SNARKs Construction and Its Implementation
Supervisor	:	Drs. Lim Yohanes Stefanus, M.Math, Ph.D.

Zero-Knowledge Proofs (ZKPs) are cryptographic protocols that enable a prover to convince a verifier of the validity of a statement without revealing any information beyond the statement's truth. Among the various ZKP constructions, zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) have emerged as particularly powerful primitives due to their succinct proof sizes and non-interactive nature, enabling privacy-preserving applications across various domains. Despite their growing importance, a significant educational gap exists between high-level conceptual descriptions and the mathematical details necessary for understanding their construction. This final project provides a comprehensive exposition of the KZG polynomial commitment scheme and demonstrates its central role in modern zk-SNARKs protocols, specifically Marlin and Plonk. The exposition builds mathematical foundations from first principles, covering groups, fields, polynomial rings, elliptic curves, and bilinear pairings, with each concept illustrated through detailed numerical examples. The KZG polynomial commitment scheme is presented with complete proofs of its security properties and optimization techniques. The project then examines how both Marlin and Plonk leverage KZG commitments to achieve universal and updatable structured reference strings while pursuing different design objectives through distinct arithmetization strategies. To bridge theory and practice, complete implementations of both protocols are provided in SageMath, allowing readers to experiment with concrete instantiations. Through systematic exposition and extensive examples computed over finite fields, this work serves as an educational resource for undergraduate students in Computer Science and Mathematics, providing the foundational knowledge necessary to understand and engage with ongoing developments in ZKPs.

Key words:
KZG polynomial commitment, zero-knowledge proofs, zk-SNARKs, Marlin, Plonk

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

LIST OF APPENDIX

CHAPTER 1
INTRODUCTION

1.1 Background

Zero-Knowledge Proofs (ZKPs) are cryptographic protocols that enable a prover to convince a verifier of the validity of a statement without revealing any information beyond the statement's truth. Since their introduction in the 1980s [GMR85], ZKPs have evolved from theoretical constructs to practical cryptographic primitives with widespread applications across various domains. ZKPs have found extensive use in privacy-preserving cryptocurrencies such as Zcash [BSCG$^+$14, FMMO19]. Beyond cryptocurrencies, ZKPs are deployed in secure healthcare data management [GHK$^+$22, TDNHDS20], confidential financial auditing [LLH$^+$22, RMMY12], and law verification [BCG$^+$22].

The development of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) marked a significant milestone in making ZKPs practical [BSCTV14]. These protocols achieve succinctness in proof size and verification time while maintaining zero-knowledge properties. The core innovation lies in their ability to transform computational statements into algebraic representations that can be efficiently verified. This transformation process, known as arithmetization, converts programs or circuits into constraint system that form the basis for proof generation and verification.

zk-SNARKs constructions follow a modular approach in their design. The typical workflow involves two main steps: first, an information-theoretically secure protocol is developed; second, this protocol is combined with cryptographic primitives to enforce the required security properties in the presence of computationally bounded adversaries [Tha22]. The polynomial commitment scheme serves as the crucial cryptographic component in this second step, enabling the prover to commit to polynomials and later reveal evaluations at specific points without exposing the entire polynomial.

The KZG polynomial commitment scheme [KZG10], named after Kate, Zaverucha, and Goldberg, has emerged as a fundamental building block for many zk-SNARKs constructions [MBKM19, GWC19, CHM$^+$20, CBBZ23, Lip24]. Its efficiency and algebraic properties make it particularly suitable for protocols that require polynomial evaluations and verification. The scheme leverages bilinear pairings to achieve constant-size commitments and evaluation proofs, regardless of the polynomial degree. This property is essential for maintaining the succinctness of zk-SNARKs.

1.2 Motivation

The field of ZKPs presents unique educational challenges for undergraduate students. While ZKPs are increasingly important in modern cryptography and blockchain technologies, the learning resources available often fall into two extremes: high-level conceptual overviews that lack mathematical rigor, or research papers dense with specialized notation and assumed background knowledge. This gap creates barriers for students seeking to understand the concrete mechanisms behind these protocols.

Prerequisites and target audience This final project is designed for undergraduate students in Computer Science and Mathematics interested in understanding zk-SNARKs from both theoretical and practical perspectives. Readers should have a foundation in basic cryptographic mathematics, particularly modular arithmetic, introductory finite field theory, elementary group theory, and fundamental probability concepts. Beyond these technical prerequisites, successful engagement with this material requires mathematical maturity: the ability to parse formal definitions, follow logical arguments, and construct mathematical proofs.

Readers who feel their mathematical background needs reinforcement will find Hoffstein et al.'s "An Introduction to Mathematical Cryptography" [HPS14] particularly helpful for building cryptographic mathematical foundations. For those seeking deeper understanding of zero-knowledge proofs specifically, Thaler's "Proofs, Arguments, and Zero-Knowledge" [Tha22] provides thorough coverage of the field. This final project develops concepts progressively, but acknowledges that some topics may require consulting these supplementary resources for full comprehension. The combination of this exposition with selective reference to these texts provides a complete learning experience tailored to individual backgrounds.

The KZG polynomial commitment scheme, despite being fundamental to recent zk-SNARKs constructions, exemplifies this educational challenge. Academic papers introducing KZG and its applications assume familiarity with bilinear pairings, polynomial rings over finite fields, and various cryptographic assumptions. The connections between abstract algebraic structures and their practical implementation remain unclear without guided exposition.

Furthermore, understanding how the same cryptographic primitive enables different protocol designs provides valuable insight into the modular nature of modern cryptography. Examining protocols that share a common foundation but achieve different performance characteristics through distinct design choices offers an excellent case study for students to appreciate how theoretical tools translate into practical implementations.

1.3 Related Work

Current state-of-the-art Recent advances in zk-SNARKs have focused on achieving universal and updatable structured reference strings (SRS). Protocols like Marlin [CHM$^+$20] and Plonk [GWC19] utilize KZG commitments to enable a single trusted setup that can be used for multiple circuits up to a predetermined size bound. This universality significantly improves the practical deployment of zk-SNARKs by eliminating the need for circuit-specific trusted setups. The updatable property further enhances security by allowing multiple parties to contribute to the SRS generation, ensuring that the system remains secure as long as at least one participant is honest [GKM$^+$18].

Among these state-of-the-art protocols, Marlin and Plonk stand out as particularly instructive for educational purposes. The Marlin protocol introduces algebraic holographic proofs that optimize prover efficiency, while Plonk simplifies the constraint system representation through custom gates and permutation arguments. Both protocols represent the current state-of-the-art in preprocessing zk-SNARKs with universal SRS, offering different trade-offs between prover complexity, verifier efficiency, and proof size.

These protocols were selected for detailed study in this work for several reasons. First, both utilize the KZG polynomial commitment scheme as their core cryptographic primitive, demonstrating its versatility. Second, they achieve similar security guarantees through fundamentally different arithmetization strategies: Marlin through R1CS and sophisticated sumcheck protocols, Plonk through custom gates and elegant permutation arguments. Third, both have gained significant adoption in real-world applications, with implementations across multiple frameworks including Arkworks, Zokrates, Gnark, SnarkJS, Noir, Dusk-PLONK, and Halo2 [SAKK25]. This widespread adoption indicates their practical importance and makes understanding their construction valuable for students entering the field.

Existing implementations Production frameworks prioritize performance and are designed for deployment rather than education. For instance, arkworks is a comprehensive Rust-based library that provides highly optimized implementations of various zk-SNARKs protocols. While Rust enables excellent performance and memory safety, its syntax and type system complexity can obscure the underlying mathematical constructions for learners. Similarly, other production frameworks like Gnark (Go) and SnarkJS (JavaScript) are optimized for their respective deployment contexts, whether high-performance servers or browser environments. These implementations necessarily involve language-specific idioms, performance optimizations, and architectural decisions that, while excellent for production use, create barriers for students trying to understand the core cryptographic constructions. The gap between the mathematical definitions in papers and these production-ready implementations motivates the need for educational code that prioritizes clarity and direct correspondence to theory.

1.4 Research Position

This final project positions itself as an educational bridge between theoretical papers and production implementations. Unlike the performance-optimized frameworks surveyed in [SAKK25], this work prioritizes pedagogical clarity, implementing protocols in a step-by-step manner that directly corresponds to theoretical constructions.

This final project addresses the educational needs identified in the motivation by providing a systematic exposition that builds mathematical concepts from first principles. Each theoretical development is accompanied by numerical examples computed over small finite fields, making abstract concepts concrete and verifiable. The inclusion of SageMath implementations allows students to experiment with the protocols, modify parameters, and observe immediate connections between theory and computation.

The SageMath implementations provided here are designed for understanding rather than performance. SageMath is chosen for its built-in support for finite field arithmetic, polynomial operations, and elliptic curves, along with its readable Python-like syntax that makes the code accessible to students. Unlike production libraries that optimize for speed through parallel processing, assembly code, and specialized data structures, these educational implementations prioritize clarity and direct correspondence to the mathematical definitions.

By focusing on educational clarity rather than novelty, this work contributes to the broader goal of making advanced cryptographic concepts accessible to the next generation of computer scientists and mathematicians. The systematic progression from mathematical foundations through concrete examples to working implementations creates a complete learning pathway. Understanding these foundational protocols, particularly how Marlin and Plonk leverage the same KZG primitive in different ways, equips students with the knowledge necessary to engage with ongoing developments in verifiable computation and privacy-preserving technologies.

This educational approach fills a crucial gap in the existing literature. While research papers assume extensive background knowledge and production implementations hide mathematical details behind optimizations, this work provides the missing middle ground where students can see exactly how theoretical constructions translate into executable code.

1.5 Objectives

This final project provides a comprehensive educational exposition of the KZG polynomial commitment scheme and its implementation in zk-SNARKs protocols. The primary objective is to make these advanced cryptographic concepts accessible to undergraduate students through clear mathematical development and concrete examples. The specific objectives are:

1. To develop the mathematical foundations of polynomial commitment schemes from first principles, including groups, fields, polynomial rings, elliptic curves, and bilinear pairings, with each concept illustrated through worked examples.
2. To present a detailed exposition of the KZG polynomial commitment scheme,

explaining its construction, proving its security properties, and demonstrating optimization techniques through concrete calculations.

3. To explain the theoretical framework of preprocessing zk-SNARKs with universal structured reference strings, showing how polynomial commitments enable the transformation from interactive proofs to non-interactive arguments.
4. To analyze the core techniques used in zk-SNARKs, including polynomial encoding, polynomial identity testing, univariate sumcheck, and permutation arguments, with step-by-step examples for each technique.
5. To provide complete expositions of the Marlin and Plonk protocols, demonstrating how both utilize KZG commitments while achieving different design objectives through distinct arithmetization approaches.
6. To implement both protocols in SageMath with educational clarity, prioritizing direct correspondence to mathematical definitions over performance optimization.
7. To compare and contrast how Marlin and Plonk leverage the same polynomial commitment primitive, illustrating the flexibility of modular cryptographic design.
8. To provide self-assessment opportunities throughout the exposition, ensuring readers can verify their understanding of each concept before proceeding to more advanced topics.

Through this systematic exposition, students will gain both theoretical understanding and practical experience with polynomial commitment schemes and their role in ZKPs. This foundation enables further exploration of advanced topics in verifiable computation and cryptographic protocol design.

1.6 Contributions

This final project makes several educational contributions to the understanding of polynomial commitment schemes and zk-SNARKs:

1. Comprehensive mathematical exposition: This work provides enough self-contained development of all mathematical concepts required for understanding KZG and its applications. Starting from basic algebraic structures and building through bilinear pairings and polynomial commitments, each concept is carefully explained with definitions, theorems, and proofs accessible to undergraduate students.

2. Extensive worked examples: Every major concept is illustrated with detailed numerical examples computed over small finite fields. These examples show exact calculations for group operations, polynomial arithmetic, elliptic curve points, pairing computations, and complete protocol executions, making abstract theory concrete and verifiable. The examples serve as checkpoints for understanding, allowing readers to verify their comprehension before proceeding.
3. Accessible protocol explanations: The Marlin and Plonk protocols are presented with clear exposition of their construction, adding explanation some missing detail in the original paper. The step-by-step development shows how polynomial commitments enable different arithmetization strategies and performance trade-offs.
4. Educational SageMath implementations: Complete implementations of both protocols are provided in SageMath, chosen for its accessibility to students and built-in support for finite field arithmetic and elliptic curves. The code follows the theoretical constructions closely. This direct correspondence between theory and implementation reinforces conceptual understanding and allows students to experiment with the protocols.
5. Comparative analysis: By examining how two prominent protocols utilize the same KZG primitive, this work illustrates the modular nature of cryptographic design. Students can observe how different design choices in arithmetization and constraint systems lead to distinct performance characteristics. This comparison demonstrates that cryptographic primitives are flexible building blocks that can be composed in various ways.
6. Bridge to advanced topics: This exposition provides sufficient background for students to engage with current research in ZKPs. By understanding these foundational protocols, students are prepared to explore recent developments in polynomial commitment schemes, transparent zk-SNARKs, and other advances in verifiable computation.
7. Validation of educational approach: The structure of this final project, progressing from mathematical foundations through concrete examples to working implementations, creates a complete learning pathway that addresses the educational objectives. Each contribution supports the goal of making advanced cryptography accessible to undergraduates by providing multiple perspectives on the same concepts.

These contributions collectively serve to lower the barrier to entry for students interested in ZKPs, providing both theoretical grounding and practical experience necessary for further study in this rapidly evolving field.

How to use this final project This final project is structured to accommodate readers with different backgrounds and learning objectives. For beginners without prior exposure to cryptographic mathematics, starting with Chapter 2 and working through all examples is recommended to build a solid foundation. For those with cryptography background who are familiar with basic group theory and finite fields, Chapter 3 on the KZG polynomial commitment scheme can be the starting point, though Sections 2.5, 2.9, 2.10, and 2.11 from Chapter 2 provide important context specific to current zk-SNARKs. Implementation-focused readers interested in understanding the code should pay particular attention to Section 2.5 for FFT operations, Section 3.2 for the core KZG construction, and Sections 6.1 and 6.2 for the complete protocol implementations.

Learning pathway The exposition follows a deliberate progression: mathematical foundations establish the algebraic structures and cryptographic tools, the KZG scheme introduces polynomial commitments as a fundamental primitive, the general framework explains how interactive proofs become non-interactive zk-SNARKs, core techniques show the common patterns across protocols, and finally the main protocols demonstrate complete constructions. The SageMath implementations in Appendices 1 can be run alongside reading the corresponding theory chapters, reinforcing understanding through experimentation. Each chapter builds on previous material, but the modular structure allows readers to focus on specific topics of interest after establishing the prerequisite knowledge.

1.7 Report Outline

This final project is organized as follows:

1. **Introduction** presents the background, motivation, objectives, and contributions of this educational exposition on polynomial commitment schemes and zk-SNARKs.
2. **Preliminaries** establishes the mathematical foundations required for understanding polynomial commitments and zk-SNARKs, including groups, fields, polynomial rings, elliptic curves, bilinear pairings, and cryptographic assumptions.

3. **KZG Polynomial Commitment Scheme** provides a detailed exposition of the KZG construction, proving its completeness, soundness, and extractability properties, followed by optimization techniques for practical implementations.
4. **Preprocessing zk-SNARKs with Universal SRS** introduces the theoretical framework for preprocessing arguments, explaining how interactive proofs transform into non-interactive zk-SNARKs through polynomial commitments.
5. **Techniques in zk-SNARKs** examines core techniques including polynomial encoding, polynomial identity testing, univariate sumcheck, and permutation arguments that enable efficient proof systems.
6. **Application of KZG on zk-SNARKs Protocols** presents complete constructions of Marlin and Plonk, demonstrating how both protocols utilize KZG commitments while achieving different design objectives.
7. **Conclusion** summarizes the key insights gained from this exposition and suggests directions for further study in polynomial commitment schemes and ZKPs.

CHAPTER 2
PRELIMINARIES

This chapter establishes the mathematical and cryptographic foundations for understanding KZG polynomial commitments and their application in zk-SNARKs. The exposition progresses from algebraic structures through cryptographic assumptions and security models.

Key results include: polynomial division enabling KZG evaluation proofs (Corollary 2.3.4), the Schwartz-Zippel lemma for efficient polynomial identity testing, FFT operations over fields with specific structure, elliptic curves providing the commitment group $\mathbb{G}_1$, and bilinear pairings enabling constant-size verification.

The Strong Diffie-Hellman assumption ensures evaluation binding (Lemma 3.2.5), while the algebraic group model enables proving extractability (Theorem 3.2.6). References follow Hoffstein et al. [HPS14] for algebraic structures, Malik et al. [MMS96] for polynomial rings, and original papers for specialized topics.

2.1 Group

Groups are fundamental algebraic structures consisting of a set with a binary operation satisfying specific axioms.

Definition 2.1.1 (Group). *A group $(G, \cdot)$ consists of a set G with a binary operation $\cdot : G \times G \to G$ satisfying:*

1. ***Closure**: For all $a, b \in G$, $a \cdot b \in G$*
2. ***Associativity**: For all $a, b, c \in G$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$*
3. ***Identity**: There exists $e \in G$ such that $e \cdot a = a \cdot e = a$ for all $a \in G$*
4. ***Inverses**: For each $a \in G$, there exists $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$*

A group is *abelian* (or *commutative*) if $a \cdot b = b \cdot a$ for all $a, b \in G$. In cryptography, groups are typically written multiplicatively with operation $\cdot$ and identity 1. For elliptic curves, additive notation is used: operation $+$, identity 0, inverse $-a$, and scalar multiplication $na = a + a + \cdots + a$ (n times).

A group G is *cyclic* if there exists an element $\mathrm{g} \in G$ such that $G = \{\mathrm{g}^k : k \in \mathbb{Z}\}$. The element g is called a *generator*, and $G = \langle \mathrm{g} \rangle$ denotes that G is generated by g. The notation $\langle \mathrm{g} \rangle$ represents the smallest subgroup containing g, which in this case is the entire group G. For finite cyclic groups, every element can be expressed as g^k for some $k \in \{0, 1, \ldots, |G| - 1\}$.

The *order* of a finite group G, denoted $|G|$, is the number of elements in G. The *order* of an element $a \in G$, denoted $\mathrm{ord}(a)$, is the smallest positive integer n such that $a^n = e$.

A subset $H \subseteq G$ is a *subgroup* if H forms a group under the operation inherited from G.

Theorem 2.1.2 (Lagrange's Theorem)**.** *Let G be a finite group and H be a subgroup of G. Then $|H|$ divides $|G|$.*

Lagrange's theorem has crucial implications for polynomial operations in zk-SNARKs. The subgroup divisibility property ensures that multiplicative subgroups of finite fields have orders that divide the field's multiplicative group order. For a multiplicative subgroup $H \subseteq \mathbb{F}_q^*$ of order n, Lagrange's theorem guarantees that $n \mid (q-1)$. This divisibility constraint is essential for the existence of primitive n-th roots of unity, which enable efficient Fast Fourier Transform (FFT) operations over finite fields.

Example 2.1.3. Consider $\mathbb{Z}_{13}^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ under multiplication modulo 13, which has order 12. Element $g = 2$ generates the entire group:

$$\begin{aligned} &2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 8, \quad 2^4 = 3, \quad 2^5 = 6, \quad 2^6 = 12 \\ &2^7 = 11, \quad 2^8 = 9, \quad 2^9 = 5, \quad 2^{10} = 10, \quad 2^{11} = 7, \quad 2^{12} = 1 \end{aligned}$$

Thus $\mathbb{Z}_{13}^* = \langle 2 \rangle$ is cyclic.

The subgroup $H = \langle 5 \rangle = \{1, 5, 12, 8\}$ has order 4, which divides 12 as guaranteed by Lagrange's theorem. Computing: $5^1 = 5$, $5^2 \equiv 12 \pmod{13}$, $5^3 \equiv 8 \pmod{13}$, $5^4 \equiv 1 \pmod{13}$.

To verify closure, consider $5 \cdot 12 \equiv 60 \equiv 8 \pmod{13}$ and $12 \cdot 8 \equiv 96 \equiv 5 \pmod{13}$. Indeed, any product of elements in H remains in H.

Element orders: $\mathrm{ord}(3) = 12$ (generator), $\mathrm{ord}(5) = 4$, $\mathrm{ord}(12) = 2$ since $12^2 \equiv 1$

$\pmod{13}$. Each order divides the group order 12, confirming Lagrange's theorem.

2.2 Ring and field

Rings and fields extend groups by incorporating two operations, typically called addition and multiplication.

Definition 2.2.1 (Ring). *A ring $(R,+,\cdot)$ consists of a set R with two binary operations satisfying:*

1. *$(R,+)$ is an abelian group with identity element 0*
2. ***Multiplicative associativity**: For all $a,b,c \in R$, $(a\cdot b)\cdot c = a\cdot(b\cdot c)$*
3. ***Distributivity**: For all $a,b,c \in R$,*
   - *(a)* $a\cdot(b+c) = (a\cdot b)+(a\cdot c)$
   - *(b)* $(a+b)\cdot c = (a\cdot c)+(b\cdot c)$

A ring is *commutative* if multiplication is commutative: $a\cdot b = b\cdot a$ for all $a,b \in R$. A ring has *unity* if there exists a multiplicative identity $1 \neq 0$ such that $1\cdot a = a\cdot 1 = a$ for all $a \in R$.

Definition 2.2.2 (Field). *A field $(F,+,\cdot)$ is a commutative ring with unity where every non-zero element has a multiplicative inverse. That is, for every $a \in F$ with $a \neq 0$, there exists $a^{-1} \in F$ such that $a\cdot a^{-1} = 1$.*

Common examples include the rational numbers $\mathbb{Q}$, real numbers $\mathbb{R}$, and complex numbers $\mathbb{C}$. In cryptography, finite fields with finitely many elements are essential.

The *characteristic* of a field F, denoted $\text{char}(F)$, is the smallest positive integer n such that $\underbrace{1+1+\cdots+1}_{n \text{ times}} = 0$. If no such n exists, then $\text{char}(F) = 0$. The characteristic is always either 0 or a prime number.

Example 2.2.3. The integers $\mathbb{Z}$ form a commutative ring with unity but not a field, since only ± 1 have multiplicative inverses. For instance, $2 \in \mathbb{Z}$ has no inverse: there is no $x \in \mathbb{Z}$ such that $2x = 1$.

In contrast, $\mathbb{F}_5 = \{0,1,2,3,4\}$ with arithmetic modulo 5 forms a field. Every non-zero element has an inverse: $1^{-1} = 1$, $2^{-1} = 3$ (since $2\cdot 3 \equiv 1 \pmod{5}$), $3^{-1} = 2$, and $4^{-1} = 4$

(since $4 \cdot 4 \equiv 1 \pmod 5$). The field $\mathbb{F}_5$ has characteristic 5 since $1+1+1+1+1 \equiv 0$ (mod 5).

2.3 Polynomial ring

Throughout this section, rings are assumed to be commutative with unity unless stated otherwise.

Definition 2.3.1 (Polynomial Ring)**.** *For a ring R, the polynomial ring $R[X]$ consists of all polynomials with coefficients from R. An element $f \in R[X]$ has the form*

$$f(X) = a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$$

where $a_i \in R$ for all $i \in \{0,1,\ldots,n\}$ and $a_n \neq 0$ if f is non-zero.

For a non-zero polynomial $f(X) = a_0 + a_1X + \cdots + a_nX^n$ with $a_n \neq 0$, its *degree* is $\deg(f) = n$. A polynomial is *monic* if its leading coefficient equals 1. For polynomials $f, g \in R[X]$, g *divides* f (denoted $g \mid f$) if there exists $h \in R[X]$ such that $f = gh$. A polynomial $p \in R[X]$ of degree at least 1 is *irreducible* over R if it cannot be expressed as a product of two polynomials in $R[X]$ of strictly smaller degree.

Theorem 2.3.2 (Polynomial Division)**.** *Let R be a ring and $f, g \in R[X]$ with g monic. Then there exist unique polynomials $q, r \in R[X]$ such that*

$$f = qg + r$$

where either $r = 0$ or $\deg(r) < \deg(g)$.

Theorem 2.3.3 (Remainder Theorem)**.** *Let R be a ring. For $f \in R[X]$ and $a \in R$, there exists $q \in R[X]$ such that*

$$f(X) = (X-a)q(X) + f(a)$$

Corollary 2.3.4. *Let R be a ring, $f \in R[X]$, and $c, v \in R$. Then $f(c) = v$ if and only if $X - c$ divides $f(X) - v$.*

Proof. By Theorem 2.3.3, $f(X) = (X-c)q(X) + f(c)$ for some $q(X) \in R[X]$.

If $f(c) = v$, then $f(X) - v = (X - c)q(X) + f(c) - v = (X - c)q(X)$, so $X - c$ divides $f(X) - v$.

Conversely, if $X - c$ divides $f(X) - v$, then $f(X) - v = (X - c)r(X)$ for some $r(X) \in R[X]$. Evaluating at $X = c$ gives $f(c) - v = 0$, hence $f(c) = v$. □

This corollary is crucial for construction KZG polynomial commitments in Section 3.2. When a prover claims that polynomial p evaluates to v at point z, they prove this by showing that $p(X) - v$ is divisible by $X - z$. The KZG opening proof $\pi = w(x)G_1$ where $w(X) = \frac{p(X)-v}{X-z}$ exists precisely because this division has no remainder when $p(z) = v$.

Corollary 2.3.5 (Factorization Theorem). *Let R be a ring. For $f \in R[X]$ and $a \in R$, $X - a$ divides f if and only if a is a root of f.*

Theorem 2.3.6. *Let F be a field and $f \in F[X]$ be a non-zero polynomial of degree d. Then f has at most d roots in F.*

Proof. By induction on d. For $d = 0$, f is a non-zero constant with no roots. For $d \geq 1$, if f has no roots, the claim holds. Otherwise, let $c \in F$ be a root. By Corollary 2.3.5, $f(X) = (X - c)q(X)$ for some $q \in F[X]$ with $\deg(q) = d - 1$. Any root of f is either c or a root of q. By induction, q has at most $d - 1$ roots, so f has at most d roots. □

Corollary 2.3.7. *Let F be a field and $f \in F[X]$ be a non-zero polynomial of degree at most d. For a finite subset $S \subseteq F$,*

$$\Pr[f(r) = 0 \mid r \leftarrow\$\, S] \leq \frac{d}{|S|}$$

Proof. By Theorem 2.3.6, f has at most d roots in F. Let $Z = \{c \in F \mid f(c) = 0\}$. Then $|Z \cap S| \leq |Z| \leq d$. The probability that a uniformly random $r \in S$ satisfies $f(r) = 0$ is $\frac{|Z \cap S|}{|S|} \leq \frac{d}{|S|}$. □

For multiple variables, the *multivariate polynomial ring* $R[X_1, X_2, \ldots, X_n]$ consists of all polynomials in n variables with coefficients from R. An element has the form

$$f(X_1, X_2, \ldots, X_n) = \sum_{\alpha \in \mathbb{Z}^n_{\geq 0}} c_\alpha X_1^{\alpha_1} X_2^{\alpha_2} \cdots X_n^{\alpha_n}$$

where only finitely many $c_\alpha \in R$ are non-zero. A *monomial* is a product $X_1^{\alpha_1} X_2^{\alpha_2} \cdots X_n^{\alpha_n}$ with degree $\alpha_1 + \alpha_2 + \cdots + \alpha_n$. The *total degree* of a multivariate polynomial is the maximum degree among its monomials.

Lemma 2.3.8 (Schwartz-Zippel Lemma [Sch80, Zip79]). *Let F be a field and $f \in F[X_1, X_2, \ldots, X_n]$ be a non-zero polynomial of total degree at most d. For a finite subset $S \subseteq F$,*

$$\Pr[f(r) = 0 \mid r \leftarrow\!\!\$\; S^n] \le \frac{d}{|S|}$$

Proof. By induction on n. For $n = 1$, apply Corollary 2.3.7. For $n > 1$, write

$$f(X_1, \ldots, X_n) = \sum_{i=0}^{d_n} g_i(X_1, \ldots, X_{n-1}) \cdot X_n^i$$

where $d_n = \deg_{X_n}(f)$ and $g_{d_n} \neq 0$. The polynomial g_{d_n} has total degree at most $d - d_n$.

By the induction hypothesis, $\Pr[g_{d_n}(r_1, \ldots, r_{n-1}) = 0] \le \frac{d-d_n}{|S|}$.

When $g_{d_n}(r_1, \ldots, r_{n-1}) \neq 0$, the polynomial $f(r_1, \ldots, r_{n-1}, X_n)$ is non-zero of degree d_n in X_n. By Corollary 2.3.7, $\Pr[f(r) = 0 \mid g_{d_n}(r_1, \ldots, r_{n-1}) \neq 0] \le \frac{d_n}{|S|}$.

When $g_{d_n}(r_1, \ldots, r_{n-1}) = 0$, we have $\Pr[f(r) = 0 \mid g_{d_n}(r_1, \ldots, r_{n-1}) = 0] \le 1$.

By the law of total probability:

$$\begin{aligned}
\Pr[f(r) = 0] &= \Pr[f(r) = 0 \mid g_{d_n} \neq 0] \cdot \Pr[g_{d_n} \neq 0] \\
&\quad + \Pr[f(r) = 0 \mid g_{d_n} = 0] \cdot \Pr[g_{d_n} = 0] \\
&\le \frac{d_n}{|S|} \cdot 1 + 1 \cdot \frac{d - d_n}{|S|} \\
&= \frac{d_n + d - d_n}{|S|} = \frac{d}{|S|}
\end{aligned}$$

□

The Schwartz-Zippel lemma enables efficient polynomial identity testing in Section 5.2, allowing zk-SNARKs to verify polynomial relations by checking equality at random points rather than comparing all coefficients.

2.4 Finite field

Finite fields, also known as Galois fields, are fields with finitely many elements.

Theorem 2.4.1. *A finite field has p^n elements for some prime p and positive integer n. The prime p is the characteristic of the field.*

For a prime p and positive integer n, the finite field with p^n elements is denoted $\mathbb{F}_{p^n}$. When $n = 1$, $\mathbb{F}_p$ denotes the field with p elements, represented as $\{0,1,2,\dots,p-1\}$ with arithmetic modulo p.

Theorem 2.4.2. *The multiplicative group $\mathbb{F}_p^* = \mathbb{F}_p \setminus \{0\}$ is cyclic of order $p-1$.*

For $n > 1$, the field $\mathbb{F}_{p^n}$ is constructed as the quotient ring $\mathbb{F}_p[X]/(f)$ where $f \in \mathbb{F}_p[X]$ is an irreducible polynomial of degree n, and (f) denotes the ideal generated by f. Elements are represented as polynomials of degree less than n with coefficients in $\mathbb{F}_p$.

Example 2.4.3. The field $\mathbb{F}_{2^3} = \mathbb{F}_8$ can be constructed using the irreducible polynomial $f(X) = X^3 + X + 1$ over $\mathbb{F}_2$. Elements are polynomials of degree at most 2: $\{0,1,X,X+1,X^2,X^2+1,X^2+X,X^2+X+1\}$.

Addition: $(X^2+X)+(X^2+1) = X+1$ (since $2 \equiv 0 \pmod 2$).

Multiplication: $(X+1)\cdot X^2 = X^3+X^2$. Since $X^3+X+1 = 0$ implies $X^3 = X+1$, we get $(X+1)\cdot X^2 = (X+1)+X^2 = X^2+X+1$.

The multiplicative group $\mathbb{F}_{2^3}^*$ has order 7. Element X is a generator: $X^1 = X$, $X^2 = X^2$, $X^3 = X+1$, $X^4 = X^2+X$, $X^5 = X^2+X+1$, $X^6 = X^2+1$, $X^7 = 1$.

2.5 Fast Fourier Transform over finite fields

The FFT over finite fields enables efficient polynomial operations in zk-SNARKs. Unlike classical FFT over complex numbers, this variant operates over finite fields $\mathbb{F}_q$ where q is a prime supporting the required operations.

Let $H = \langle \mathrm{g} \rangle$ be a multiplicative subgroup of $\mathbb{F}_q^*$ of order $n = 2^k$ for some $k \in \mathbb{N}$, where g is a primitive n-th root of unity. The existence of such a subgroup requires $n \mid (q-1)$

by Theorem 2.1.2. The FFT enables efficient conversion between two representations of a polynomial $f \in \mathbb{F}_q[X]$:

- *Coefficient representation*: $f(X) = \sum_{i=0}^{n-1} a_i X^i$
- *Evaluation representation*: $(f(\mathrm{g}^0), f(\mathrm{g}^1), \ldots, f(\mathrm{g}^{n-1}))$

The forward FFT transforms coefficients to evaluations, while the inverse FFT (IFFT) performs the reverse. Both operations require $O(n \log n)$ field operations using the Cooley-Tukey algorithm.

For a polynomial $f(X) = \sum_{i=0}^{n-1} a_i X^i$, the FFT recursively splits into even and odd coefficients:

$$f(X) = f_{\text{even}}(X^2) + X \cdot f_{\text{odd}}(X^2)$$

where $f_{\text{even}}(X) = \sum_{i=0}^{n/2-1} a_{2i} X^i$ and $f_{\text{odd}}(X) = \sum_{i=0}^{n/2-1} a_{2i+1} X^i$.

This decomposition allows shared computations when evaluating at g^j and $\mathrm{g}^{j+n/2}$, since $(\mathrm{g}^{n/2})^2 = 1$ implies $(\mathrm{g}^{j+n/2})^2 = \mathrm{g}^{2j}$. The algorithm recurses on subproblems of size $n/2$ until reaching constant polynomials.

The relationship between representations is given by the Vandermonde matrix V where $V_{ij} = \mathrm{g}^{ij}$:

$$\begin{bmatrix} f(\mathrm{g}^0) \\ f(\mathrm{g}^1) \\ \vdots \\ f(\mathrm{g}^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mathrm{g} & \mathrm{g}^2 & \cdots & \mathrm{g}^{n-1} \\ 1 & \mathrm{g}^2 & \mathrm{g}^4 & \cdots & \mathrm{g}^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{g}^{n-1} & \mathrm{g}^{2(n-1)} & \cdots & \mathrm{g}^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

The FFT efficiently computes $V\boldsymbol{a}$ (coefficient to evaluation), while IFFT computes $V^{-1}\boldsymbol{v}$ (evaluation to coefficient).

The inverse matrix satisfies $V^{-1} = \frac{1}{n}\overline{V}$ where $\overline{V}_{ij} = \mathrm{g}^{-ij}$. To verify, the (i,j)-th entry of $V\overline{V}$ is:

$$(V\overline{V})_{ij} = \sum_{k=0}^{n-1} \mathrm{g}^{ik} \cdot \mathrm{g}^{-kj} = \sum_{k=0}^{n-1} \mathrm{g}^{k(i-j)}$$

This sum equals n when $i = j$, and 0 otherwise. When $i \neq j$, let $d = i - j \neq 0$. Since g has order n, $\mathrm{g}^d \neq 1$. Using the geometric series formula:

$$\sum_{k=0}^{n-1} \mathrm{g}^{kd} = \frac{\mathrm{g}^{nd} - 1}{\mathrm{g}^d - 1} = \frac{(\mathrm{g}^n)^d - 1}{\mathrm{g}^d - 1} = \frac{1-1}{\mathrm{g}^d - 1} = 0$$

Thus $V\overline{V} = nI$, yielding $V^{-1} = \frac{1}{n}\overline{V}$.

Therefore, the IFFT computes coefficients from evaluations using:

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} v_k \cdot \mathrm{g}^{-jk}$$

The IFFT uses the same recursive structure as FFT but with g^{-1} instead of g, and scales the result by n^{-1}. The algorithm is shown in Code 2.1.

```python
def fft(coeffs, omega, F):
    """FFT: coefficients to evaluations"""
    n = len(coeffs)
    if n == 1:
        return coeffs

    # Split and recurse
    even = fft(coeffs[0::2], omega**2, F)
    odd = fft(coeffs[1::2], omega**2, F)

    # Combine results
    result = [F(0)] * n
    omega_power = F(1)
    for i in range(n//2):
        result[i] = even[i] + omega_power * odd[i]
        result[i + n//2] = even[i] - omega_power * odd[i]
        omega_power *= omega

    return result

def ifft(values, omega, F):
    """IFFT: evaluations to coefficients"""
    n = len(values)
    # Use omega^(-1) and scale by n^(-1)
    result = fft(values, omega**(-1), F)
    n_inv = F(n)**(-1)
    return [x * n_inv for x in result]
```

Code 2.1: FFT and IFFT implementation

In zk-SNARKs, FFT is essential for:

- *Polynomial interpolation*: Given evaluations at H, IFFT computes the unique polynomial of degree less than n
- *Polynomial multiplication*: Multiply in $O(n \log n)$ time via pointwise multiplication in evaluation form

Example 2.5.1. Consider polynomial interpolation in $\mathbb{F}_{17}$ using FFT. Let $H = \langle 13 \rangle = \{1, 13, 16, 4\}$ be a subgroup of order 4, where 13 is a primitive 4th root of unity (since $13^4 \equiv 1 \pmod{17}$).

Given evaluations $v = (8, 1, 14, 12)$ at points $(13^0, 13^1, 13^2, 13^3) = (1, 13, 16, 4)$, find the polynomial f of degree less than 4 such that $f(13^i) = v_i$.

Using IFFT with $\mathrm{g}^{-1} = 13^{-1} \equiv 4 \pmod{17}$:

First recursion splits $v = (8, 1, 14, 12)$ into even indices $(8, 14)$ and odd indices $(1, 12)$.

For even indices: Recursively apply IFFT with $(\mathrm{g}^{-1})^2 = 16$. Combining gives $a_0 = 8 + 14 = 5$ and $a_1 = 8 - 14 = 11$.

For odd indices: With $(\mathrm{g}^{-1})^2 = 16$, we get $a_0 = 1 + 12 = 13$ and $a_1 = 1 - 12 = 6$.

Combining with powers of $\mathrm{g}^{-1} = 4$:

$$\begin{aligned}
a_0 &= 5 + 4^0 \cdot 13 = 1 \\
a_1 &= 11 + 4^1 \cdot 6 = 1 \\
a_2 &= 5 - 4^0 \cdot 13 = 9 \\
a_3 &= 11 - 4^1 \cdot 6 = 4
\end{aligned}$$

Final scaling by $n^{-1} = 4^{-1} \equiv 13 \pmod{17}$ gives coefficients $(13, 13, 15, 1)$.

Therefore $f(X) = X^3 + 15X^2 + 13X + 13$. The polynomial can be verified by checking that $f(1) = 8$, $f(13) = 1$, $f(16) = 14$, and $f(4) = 12$.

2.6 Elliptic curve

Definition 2.6.1 (Elliptic Curve)**.** *Let K be a field with characteristic neither 2 nor 3. An elliptic curve E over K is the set of solutions $(x,y) \in K \times K$ to an equation of the form*

$$y^2 = x^3 + ax + b$$

where $a, b \in K$ and the discriminant $\Delta = -16(4a^3 + 27b^2) \neq 0$, together with a special point $\mathcal{O}$ called the point at infinity.

The equation $y^2 = x^3 + ax + b$ is the *short Weierstrass form*. The discriminant condition $\Delta \neq 0$ ensures the curve is non-singular (no self-intersections or cusps).

Definition 2.6.2 (Group Operation on Elliptic Curves)**.** *An elliptic curve E forms an abelian group with $\mathcal{O}$ as identity. For points $P, Q \in E$, the sum $P + Q$ is defined:*

1. *If $P = \mathcal{O}$, then $P + Q = Q$*
2. *If $Q = \mathcal{O}$, then $P + Q = P$*
3. *If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $x_P = x_Q$ and $y_P = -y_Q$, then $P + Q = \mathcal{O}$*
4. *Otherwise, $P + Q = R$ where $R = (x_R, y_R)$ with:*

$$x_R = \lambda^2 - x_P - x_Q$$
$$y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \frac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{ (point addition)} \\ \frac{3x_P^2 + a}{2y_P} & \text{if } P = Q \text{ (point doubling)} \end{cases}$$

Theorem 2.6.3. *Let E be an elliptic curve over field K. Then $(E, +)$ forms an abelian group.*

For elliptic curves over finite fields $\mathbb{F}_q$, the set $E(\mathbb{F}_q)$ forms a finite abelian group.

Theorem 2.6.4 (Hasse's Theorem)**.** *Let E be an elliptic curve over $\mathbb{F}_q$. The number of points satisfies:*

$$|q + 1 - \#E(\mathbb{F}_q)| \leq 2\sqrt{q}$$

Hasse's theorem provides tight bounds on the group order, showing that $\#E(\mathbb{F}_q)$ is approximately $q+1$ with deviation at most $2\sqrt{q}$.

Example 2.6.5. Consider $E : y^2 = x^3 + x + 1$ over $\mathbb{F}_{11}$. The discriminant $\Delta = -16(4+27) \equiv 6 \pmod{11}$ is non-zero.

Finding all points: for each $x \in \mathbb{F}_{11}$, compute $x^3 + x + 1$ and check if it's a quadratic residue.

- $x = 0$: $y^2 = 1$, so $y = \pm 1 \equiv 1, 10$
- $x = 1$: $y^2 = 3$, no solutions (3 is not a square mod 11)
- $x = 2$: $y^2 = 11 \equiv 0$, so $y = 0$
- $x = 3$: $y^2 = 31 \equiv 9$, so $y = \pm 3 \equiv 3, 8$

Continuing this process yields $E(\mathbb{F}_{11})$ is:

$$\begin{gathered}\{\mathcal{O}, (0,1), (0,10), (1,5), (1,6), (2,0), (3,3) \\ (3,8), (4,5), (4,6), (6,5), (6,6), (8,2), (8,9)\}\end{gathered}$$

with 14 points which falls within the Hasse bound $[5.34, 17.66]$.

Point addition: $P = (3,3) + Q = (4,5)$

$$\begin{aligned} \lambda &= \frac{5-3}{4-3} = 2 \\ x_R &= 2^2 - 3 - 4 = -3 \equiv 8 \pmod{11} \\ y_R &= 2(3-8) - 3 = -13 \equiv 9 \pmod{11} \end{aligned}$$

Thus $P + Q = (8,9)$.

Point doubling: $2P = 2(3,3)$

$$\begin{aligned} \lambda &= \frac{3 \cdot 3^2 + 1}{2 \cdot 3} = \frac{28}{6} \equiv 28 \cdot 2 \equiv 1 \pmod{11} \\ x_R &= 1^2 - 2 \cdot 3 = -5 \equiv 6 \pmod{11} \\ y_R &= 1(3-6) - 3 = -6 \equiv 5 \pmod{11} \end{aligned}$$

Thus $2P = (6,5)$.

2.7 Bilinear pairing

Definition 2.7.1 (Bilinear Pairing). *Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ be cyclic groups of prime order q. A bilinear pairing is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with the following properties:*

1. ***Bilinearity**: For all $a, b \in \mathbb{Z}_q$, $P \in \mathbb{G}_1$, and $Q \in \mathbb{G}_2$:*

$$e(aP, bQ) = e(P, Q)^{ab}$$

2. ***Non-degeneracy**: If P generates $\mathbb{G}_1$ and Q generates $\mathbb{G}_2$, then $e(P, Q)$ generates $\mathbb{G}_T$*
3. ***Efficiency**: The pairing e can be computed efficiently*

A pairing is *symmetric* if $\mathbb{G}_1 = \mathbb{G}_2$, otherwise *asymmetric*. In practice, asymmetric pairings offer better efficiency.

A *bilinear group* is represented as $bp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G_1, G_2, q)$ where:

- $\mathbb{G}_1, \mathbb{G}_2$: source groups (typically elliptic curve groups)
- $\mathbb{G}_T$: target group (subgroup of a finite field extension)
- e: the pairing operation
- G_1, G_2: generators of $\mathbb{G}_1, \mathbb{G}_2$ respectively
- q: prime order of all groups

The *embedding degree* of an elliptic curve E over $\mathbb{F}_p$ with subgroup of order q is the smallest positive integer k such that $q \mid (p^k - 1)$. For practical pairings, curves need embedding degree balancing security and efficiency.

Two pairing-friendly curves dominate modern implementations:

BN254 The Barreto-Naehrig curve [BN06] with embedding degree $k = 12$:

- Base field: $\mathbb{F}_p$ where $p \approx 2^{254}$
- Base curve: $E/\mathbb{F}_p : y^2 = x^3 + 3$
- Prime order: $q \approx 2^{254}$ with $2^{28} \mid (q - 1)$

BLS12-381 The Barreto-Lynn-Scott curve [BLS03] with embedding degree $k = 12$:

- Base field: $\mathbb{F}_p$ where $p \approx 2^{381}$
- Base curve: $E/\mathbb{F}_p : y^2 = x^3 + 4$
- Prime order: $q \approx 2^{255}$ with $2^{32} \mid (q-1)$

For KZG commitments, the field $\mathbb{F}_q$ requires $q-1$ to have a large smooth factor (specifically large powers of 2) to enable efficient FFT operations as discussed in Section 2.5. By Lagrange's Theorem 2.1.2, multiplicative subgroups of order 2^k can exist in $\mathbb{F}_q^*$ only when $2^k \mid (q-1)$. Both BN254 and BLS12-381 pairing-friendly curves are designed with this property: BN254 has $2^{28} \mid (q-1)$ and BLS12-381 has $2^{32} \mid (q-1)$, supporting FFT operations on domains of size up to 2^{28} and 2^{32} respectively.

Example 2.7.2. Let $bp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G_1, G_2, q)$ be a bilinear group. The bilinearity property enables:

$$\begin{aligned} e(3G_1, 5G_2) &= e(G_1, G_2)^{3\cdot 5} = e(G_1, G_2)^{15} \\ e(aG_1 + bG_1, G_2) &= e((a+b)G_1, G_2) = e(G_1, G_2)^{a+b} \\ &= e(G_1, G_2)^a \cdot e(G_1, G_2)^b = e(aG_1, G_2) \cdot e(bG_1, G_2) \end{aligned}$$

Bilinear pairings are the key cryptographic tool enabling constant-size evaluation proofs in KZG in Section 3.2. The pairing equation $e(C - vG_1, G_2) = e(\pi, xG_2 - zG_2)$ allows verification of polynomial evaluations without revealing the polynomial itself.

2.8 Hash function

Hash functions are fundamental cryptographic primitives that compress arbitrary-length inputs into fixed-length outputs [KL20]. A *hash function* is a deterministic algorithm $H : \{0,1\}^* \to \{0,1\}^\ell$ that takes an input string of any length and produces an output of fixed length ℓ. This compression property is essential: hash functions map a potentially infinite domain to a finite range, creating compact representations of large data.

The primary security requirement for cryptographic hash functions is *collision resistance*. A hash function is collision resistant if it is computationally infeasible for any efficient adversary to find two distinct inputs $x \neq x'$ such that $H(x) = H(x')$. Such a pair (x, x') is called a *collision*. Since hash functions compress their inputs, collisions must exist by the pigeonhole principle, but finding them should be computationally hard.

Hash functions satisfy additional security properties beyond collision resistance. *Preimage resistance* requires that given a hash output $y = H(x)$ for a randomly chosen input x, it should be hard to find any input x' such that $H(x') = y$. *Second-preimage resistance* requires that given an input x, it should be hard to find a different input $x' \neq x$ such that $H(x') = H(x)$. Collision resistance implies second-preimage resistance, but not vice versa.

Real-world cryptographic hash functions such as the SHA-2 [Nat15a] and SHA-3 [Nat15b] families are designed to satisfy these security properties. These standardized functions have undergone extensive cryptanalytic scrutiny and are widely deployed in practice. While theoretical analysis often considers keyed hash functions to avoid certain technical complications, practical implementations use these unkeyed, standardized functions.

Hash functions serve as foundational building blocks for numerous cryptographic constructions. They provide a way to create compact ”digital fingerprints” of large data, enabling efficient verification and comparison. In the context of ZKPs, hash functions play crucial roles in the Fiat-Shamir transform for achieving non-interactive protocols in Section 4.4.

2.9 Commitment scheme

Commitment schemes [Tha22] are cryptographic protocols involving two parties: a *committer* and a *verifier*. The committer wishes to bind itself to a message without revealing the message to the verifier. This creates a digital analogue of placing a message in a sealed envelope: the commitment binds the committer to a specific value while keeping that value hidden until a later revelation phase.

A commitment scheme must satisfy two fundamental security properties. The *binding* property ensures that once the committer sends a commitment to some message m, it should be unable to ”open” the commitment to any value other than m. The *hiding* property guarantees that the commitment itself should not reveal information about m to the verifier. These properties create a temporal separation between commitment and revelation, enabling protocols where parties must commit to choices before learning others’ decisions.

Formally, a commitment scheme consists of three algorithms: KeyGen, Commit, and Verify. The key generation algorithm KeyGen is randomized and generates a commitment key ck and verification key vk that are available to the committer and verifier respectively. The commitment algorithm Commit is randomized, taking as input the commitment key ck and message m to produce a commitment c along with possible opening information d that the committer retains. The verification algorithm Verify takes the commitment c, verification key vk, a claimed message m', and opening information d, then decides whether to accept m' as a valid opening. The commitment scheme is *correct* if $\mathsf{Verify}(\mathsf{vk}, c, m, d) = 1$ with probability 1 whenever $(c, d) \leftarrow \mathsf{Commit}(\mathsf{ck}, m)$, meaning honest committers can always successfully open their commitments.

Polynomial commitment schemes represent a specialized and powerful class of commitment schemes where the committed messages are polynomials. Polynomial commitments leverage the algebraic structure of polynomials to enable additional functionality. Most importantly, they support *evaluation proofs*: after committing to a polynomial $p(X)$, the committer can later prove that $p(z) = v$ for any point z and claimed evaluation v, without revealing the entire polynomial. This capability transforms polynomial commitments from simple hiding primitives into sophisticated tools for verifiable computation. The formal treatment of polynomial commitment schemes is developed in Section 3.1.

2.10 Cryptographic assumptions

Modern cryptographic constructions rely on computational hardness assumptions. A function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every positive polynomial p, there exists Λ such that for all $\lambda > \Lambda$, $\mathsf{negl}(\lambda) < \frac{1}{\mathsf{p}(\lambda)}$ [KL20]. This formalizes cryptographically small probabilities.

In cryptographic applications, field size $|\mathbb{F}_q|$ is chosen to be exponential in the security parameter: $|\mathbb{F}_q| = 2^{\Theta(\lambda)}$. This ensures that probability bounds from Lemma 2.3.8 are negligible. For a polynomial of degree $d = \mathsf{poly}(\lambda)$ over such a field, the probability of randomly hitting a root is at most $\frac{d}{|\mathbb{F}_q|} = \frac{\mathsf{poly}(\lambda)}{2^{\Theta(\lambda)}} = \mathsf{negl}(\lambda)$

Let BP be a bilinear group sampler outputting $bp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G_1, G_2, q)$.

Assumption 2.10.1 (Discrete Logarithm (DL) Assumption)**.** *For any polynomial-time* *adversary $\mathcal{A}$:*

$$\Pr\left[x' = x \;\middle|\; \begin{array}{l} bp \leftarrow \mathsf{BP}(1^\lambda) \\ x \leftarrow\$ \mathbb{F}_q \\ \Sigma \leftarrow \{x^i G_1\}_{i=0}^d \cup \{xG_2\} \\ x' \leftarrow \mathcal{A}(bp, \Sigma) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

Assumption 2.10.2 (Strong Diffie-Hellman (SDH) Assumption [BB04])**.** *For any polynomial-time adversary $\mathcal{A}$ and degree bound d:*

$$\Pr\left[C = \frac{1}{x+c} G_1 \;\middle|\; \begin{array}{l} bp \leftarrow \mathsf{BP}(1^\lambda) \\ x \leftarrow\$ \mathbb{F}_q \\ \Sigma \leftarrow \{x^i G_1\}_{i=0}^d \cup \{xG_2\} \\ (c, C) \leftarrow \mathcal{A}(bp, \Sigma) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

where $c \in \mathbb{F}_q \setminus \{-x\}$.

The DL assumption is well-established and forms the security foundation for many cryptographic protocols, including Diffie-Hellman key exchange [DH76] and elliptic curve cryptography [Kob87, Mil86]. Its hardness on properly chosen elliptic curves has withstood decades of cryptanalytic effort [HPS14].

The SDH assumption is a stronger than DL: breaking DL enables breaking SDH by computing x, choosing any $c \neq -x$, and outputting $\frac{1}{x+c} G_1$. This more specialized assumption introduced specifically for pairing-based cryptography. While less time-tested than DL, SDH is necessary for proving the security of KZG polynomial commitments scheme. The additional structure in SDH (providing multiple powers of x) reflects the polynomial evaluation capabilities that KZG commitments enable.

2.11 Idealized model

Idealized models provide frameworks for analyzing cryptographic security by abstracting certain components. These models balance theoretical rigor with practical relevance for proving security properties.

Random oracle model The *random oracle model* (ROM) [BR93] idealizes hash functions as truly random functions. A random oracle $\mathcal{H} : \{0,1\}^* \to \{0,1\}^n$ operates by:

- Returning the same output for repeated queries
- Selecting uniformly random outputs for new queries

The ROM is essential for the Fiat-Shamir transform [FS86], which converts interactive protocols into non-interactive ones by replacing verifier challenges with hash function evaluations of the transcript. This enables practical zk-SNARKs from interactive protocols.

Algebraic group model The *algebraic group model* (AGM) [FKL18] restricts adversaries to algebraic operations on group elements. An algebraic adversary outputting $Y \in \mathbb{G}$ must provide coefficients $(a_1, \ldots, a_n) \in \mathbb{F}_q^n$ such that $Y = \sum_{i=1}^{n} a_i X_i$, where $X_1, \ldots, X_n$ are previously received group elements.

The AGM together with Assumption 2.10 is particularly important for proving the extractability of KZG polynomial commitments in Theorem 3.2.6, where the algebraic representation allows extraction of the committed polynomial from any valid commitment. This means that in the AGM, any adversary who produces a valid KZG opening proof must actually "know" the polynomial they committed to. They cannot create valid proofs through non-algebraic manipulation of group elements. This extractability property is crucial for ensuring that a cheating prover cannot convince the verifier of false statements without possessing a valid witness.

The AGM is intermediate between the standard model and the *generic group model* (GGM) [Sho97]:

- GGM: Adversaries access group elements only through random encodings
- AGM: Adversaries see actual encodings but must use them algebraically
- Standard model: No restrictions on adversary behavior

Security in AGM implies security in GGM, but not vice versa. The AGM provides sufficient guarantees for polynomial commitment extractability while making fewer idealizations than GGM.

CHAPTER 3
KZG POLYNOMIAL COMMITMENT SCHEME

The KZG polynomial commitment scheme, introduced by Kate, Zaverucha, and Goldberg [KZG10], is a cryptographic primitive that allows a prover to commit to a polynomial and later reveal evaluations of this polynomial at specific points. This scheme leverages the mathematical foundations established in Chapter 2: groups for commitment representation, polynomial division properties for creating evaluation proofs, and bilinear pairings for succinct verification.

In this chapter, the KZG polynomial commitment scheme is presented following the formalization given in [CHM$^+$20, Appendix B]. The focus is on a specific variant that supports committing to multiple polynomials with a single degree bound and evaluating them at a single point. The chosen variant is also non-hiding for simplicity.

The chapter proceeds as follows: Section 3.1 formalizes polynomial commitment schemes and their required properties. Section 3.2 presents the KZG construction, demonstrating how bilinear pairings enable constant-size commitments and proofs regardless of polynomial degree. Section 3.3 explores crucial optimizations including batch verification and opening linear combinations of polynomials, techniques that make KZG practical for large-scale zk-SNARKs applications.

For the remainder of this final project, $[n]$ denotes the set $\{1,2,\ldots,n\}$. The notation $\boldsymbol{a} = [a_i]_{i=1}^n$ serves as shorthand for the list $[a_1, a_2, \ldots, a_n]$. For polynomial lists, $\deg(\boldsymbol{p}) = \max_{i\in[n]} \deg(p_i)$ denotes the maximum degree of any polynomial in $\boldsymbol{p}$. The set $\mathbb{F}^{<d}[X]$ denotes univariate polynomials over the field $\mathbb{F}$ with degree strictly less than d. For a list of polynomials $\boldsymbol{p} = [p_i]_{i\in[n]}$ and a field element $z \in \mathbb{F}$, $\boldsymbol{p}(z) = [p_i(z)]_{i=1}^n$ represents the list of evaluations of each polynomial in $\boldsymbol{p}$ at the point z.

3.1 Definition

Polynomial commitment schemes enable a prover to commit to a polynomial without revealing it, then later prove evaluations at specific points. This primitive is essential for zk-SNARKs, where polynomial relations encode computational statements. The commitment hides the polynomial while binding the prover to consistent evaluations..

A polynomial commitment scheme over a field family $\mathcal{F}$ for a single degree bound and single evaluation point is a tuple of algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Check})$ with the following syntax:

$\mathsf{PC.Setup}(1^\lambda, D) \rightarrow (\mathsf{ck}, \mathsf{rk})$ On input a security parameter λ (in unary), and a maximum degree bound $D \in \mathbb{N}$, $\mathsf{PC.Setup}$ samples a key pair $(\mathsf{ck}, \mathsf{rk})$. The keys contain a description of a finite field $\mathbb{F} \in \mathcal{F}$.

$\mathsf{PC.Commit}(\mathsf{ck}, \mathsf{pk}, \boldsymbol{p}) \rightarrow \boldsymbol{c}$ On input ck and univariate polynomials $\boldsymbol{p} = [p_i]_{i=1}^n$ over the field $\mathbb{F}$ with $\deg(\boldsymbol{p}) \leq D$, $\mathsf{PC.Commit}$ outputs commitments $\boldsymbol{c} = [c_i]_{i=1}^n$ to the polynomials $\boldsymbol{p}$.

$\mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p}, z, \xi) \rightarrow \pi$ On input ck, univariate polynomials $\boldsymbol{p} = [p_i]_{i=1}^n$, evaluation point $z \in \mathbb{F}$, and opening challenge ξ, $\mathsf{PC.Open}$ outputs an evaluation proof π.

$\mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}, \pi, \xi) \in \{0,1\}$ On input rk, commitments $\boldsymbol{c} = [c_i]_{i=1}^n$, evaluation point $z \in \mathbb{F}$, alleged evaluations $\boldsymbol{v} = [v_i]_{i=1}^n$, evaluation proof π, and opening challenge ξ, $\mathsf{PC.Check}$ outputs 1 if π attests that, for each $i \in [n]$, the polynomial committed in c_i has degree at most D and evaluates to v_i at z.

The polynomial commitment scheme must satisfy the following properties:

Definition 3.1.1 (Completeness). *For every maximum degree bound $D \in \mathbb{N}$ and efficient adversary $\mathcal{A}$ it holds that*

$$\Pr\left[\begin{array}{c|r} & (\mathsf{ck}, \mathsf{rk}) \leftarrow \mathsf{PC.Setup}(1^\lambda, D) \\ \deg(\boldsymbol{p}) \leq D & (\boldsymbol{p}, z, \xi) \leftarrow \mathcal{A}(\mathsf{ck}, \mathsf{rk}) \\ \Downarrow & \boldsymbol{c} \leftarrow \mathsf{PC.Commit}(\mathsf{ck}, \boldsymbol{p}) \\ \mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}, \pi, \xi) = 1 & \boldsymbol{v} \leftarrow \boldsymbol{p}(z) \\ & \pi \leftarrow \mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p}, z, \xi) \end{array}\right] = 1$$

Completeness ensures honest provers can always convince verifiers. This property is typically straightforward to achieve and verify.

Definition 3.1.2 (Extractability)**.** *For every maximum degree bound $D \in \mathbb{N}$ and efficient adversary $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$ such that for every round bound $r \in \mathbb{N}$, efficient public-coin challenger $\mathcal{C}$, efficient query sampler $\mathcal{Q}$, and efficient adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ the following probability is negligibly close to 1:*

$$\Pr\left[\begin{array}{c|r} & (\mathsf{ck},\mathsf{rk}) \leftarrow \mathsf{PC.Setup}(1^\lambda, D) \\ \hline & \textit{For } i = 1,\ldots,n: \\ & \rho_i \leftarrow \mathcal{C}(\mathsf{ck},\mathsf{rk},i) \\ \mathsf{PC.Check}(\mathsf{rk},\boldsymbol{c},z,\boldsymbol{v},\pi,\xi) = 1 & \boldsymbol{c}_i \leftarrow \mathcal{A}(\mathsf{ck},\mathsf{rk},[\rho_j]_{j=1}^{i}) \\ \Downarrow & \boldsymbol{p}_i \leftarrow \mathcal{E}(\mathsf{ck},\mathsf{rk},[\rho_j]_{j=1}^{i}) \\ \hline \deg(\boldsymbol{p}) \leq D \textit{ and } \boldsymbol{v} = \boldsymbol{p}(z) & z \leftarrow \mathcal{Q}(\mathsf{ck},\mathsf{rk},[\rho_j]_{j=1}^{r}) \\ & (\boldsymbol{v},\cdot \leftarrow \mathcal{B}_1(\mathsf{ck},\mathsf{rk},[\rho_j]_{j=1}^{r},z) \\ & \textit{Sample opening challenge } \xi \\ & \pi \leftarrow \mathcal{B}_2(\xi) \\ & \textit{Set } \boldsymbol{c} := [\boldsymbol{c}_i]_{i\in[n]}, \boldsymbol{p} := [\boldsymbol{p}_i]_{i\in[n]} \end{array}\right]$$

Extractability guarantees that any commitment corresponds to an actual polynomial of bounded degree. This property, proven in the algebraic group model for KZG, prevents malicious provers from creating commitments that cannot be opened consistently.

Definition 3.1.3 (Succinctness)**.** *A polynomial commitment scheme is succinct if the size of commitments, the size of evaluation proofs, and the time to check an opening are all independent of the degree of the committed polynomials. That is, $|\boldsymbol{c}| = n \cdot \mathsf{poly}(\lambda)$, $|\pi| = \mathsf{poly}(\lambda)$, and $\mathsf{time}(\mathsf{Check}) = n \cdot \mathsf{poly}(\lambda)$.*

Without succinctness, polynomial commitments offer no advantage over sending the polynomial directly. KZG achieves optimal succinctness with commitments and proofs consisting of single group elements.

These three properties establish the security and efficiency framework for polynomial commitment schemes. Completeness ensures that honest provers can always generate valid proofs accepted by verifiers when committed polynomials truly evaluate to claimed values. Extractability provides the crucial security guarantee that any commitment-evaluation pair passing verification must correspond to an actual polynomial of bounded degree. This prevents malicious provers from crafting non-polynomial commitments or claiming false evaluations. Succinctness makes the scheme practical for cryptographic applications by keeping commitment sizes and verification costs independent of polynomial degree. This allows efficient handling of high-degree polynomials without prohibitive computational or communication overhead.

3.2 Construction

This section presents the KZG polynomial commitment scheme based on bilinear pairings. The construction leverages the bilinearity property to enable constant-size commitments and evaluation proofs, regardless of polynomial degree.

Let $bp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G_1, G_2, q)$ be a bilinear group as defined in Section 2.7. Figure 3.1 summarizes the polynomial commitment scheme.

Setup On input a security parameter λ (in unary) and a maximum degree bound $D \in \mathbb{N}$, the setup algorithm samples a bilinear group $bp \leftarrow \mathsf{BP}(1^\lambda)$ and generates a random element $x \leftarrow\$ \mathbb{F}_q$. It then computes the powers $\{x^i G_1\}_{i=0}^{D} \in \mathbb{G}_1^{D+1}$, which serve as the public parameters. The commitment key is set as $\mathsf{ck} := (bp, \{x^i G_1\}_{i=0}^{D})$, while the verification key is set as $\mathsf{rk} := (bp, xG_2)$. The algorithm outputs the key pair $(\mathsf{ck}, \mathsf{rk})$.

Commit On input ck and univariate polynomials $\boldsymbol{p} = [p_i]_{i=1}^{n}$ over $\mathbb{F}_q$, the commitment algorithm first checks that each polynomial has degree at most D, aborting otherwise. For each polynomial p_i represented as $p_i(X) = \sum_{j=0}^{D} a_{i,j} X^j$ where $a_{i,j} \in \mathbb{F}_q$, it computes the commitment $c_i := p_i(x)G_1 = \sum_{j=0}^{D} a_{i,j} \cdot (x^j G_1)$. This is possible without knowing the secret value x directly, as the commitment key contains all required powers $x^j G_1$. The algorithm outputs the list of commitments $\boldsymbol{c} = [c_i]_{i=1}^{n}$.

Open On input ck, univariate polynomials $\boldsymbol{p} = [p_i]_{i=1}^{n}$, evaluation point $z \in \mathbb{F}_q$, and opening challenge $\xi \in \mathbb{F}_q$, the algorithm first verifies that all polynomials have degree at most D. It then computes the linear combination $p(X) := \sum_{i=1}^{n} \xi^i p_i(X)$. According to Corollary 2.3.4, the polynomial $X - z$ divides $p(X) - p(z)$. This allows the algorithm to compute the witness polynomial $w(X) := \frac{p(X)-p(z)}{X-z}$ that is well-defined as polynomial with degree less than D. The final proof is $\pi := w(x)G_1$, computed using the parameter in the commitment key.

Check On input rk, commitments $\boldsymbol{c} = [c_i]_{i=1}^{n}$, evaluation point $z \in \mathbb{F}_q$, alleged evaluations $\boldsymbol{v} = [v_i]_{i=1}^{n}$, evaluation proof π, and opening challenge $\xi \in \mathbb{F}_q$, the verification algorithm computes the linear combinations $C := \sum_{i=1}^{n} \xi^i c_i$ and $v := \sum_{i=1}^{n} \xi^i v_i$. It then verifies the evaluation proof by checking the pairing equation equality $e(C - vG_1, G_2) = e(\pi, xG_2 - zG_2)$. The algorithm outputs 1 if the check passes, and 0 otherwise.

```
$\mathsf{PC.Setup}(1^\lambda, D) \to (\mathsf{ck}, \mathsf{rk})$
    $bp \leftarrow \mathsf{BP}(1^\lambda)$
    $x \leftarrow\!\!\$\ \mathbb{F}_q$
    Compute $\{x^i G_1\}_{i=0}^{D} \in \mathbb{G}_1^{D+1}$
    $\mathsf{ck} := (bp, \{x^i G_1\}_{i=0}^{D})$
    $\mathsf{rk} := (bp, xG_2)$
    return $(\mathsf{ck}, \mathsf{rk})$

$\mathsf{PC.Commit}(\mathsf{ck}, \boldsymbol{p}) \to \boldsymbol{c}$
    Check that deg($\boldsymbol{p}$) $\leq D$, abort otherwise
    For each $p_i(X) = \sum_{j=0}^{D} a_{i,j} X^j$ :
        $c_i := \sum_{j=0}^{D} a_{i,j} \cdot (x^j G_1)$
    return $\boldsymbol{c} = [c_i]_{i=1}^{n}$

$\mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p}, z, \xi) \to \pi$
    Check that deg($\boldsymbol{p}$) $\leq D$
    $p(X) := \sum_{i=1}^{n} \xi^i p_i(X)$
    $w(X) := \frac{p(X) - p(z)}{X - z}$
    return $\pi := w(x) G_1$

$\mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}, \pi, \xi) \to 0/1$
    $C := \sum_{i=1}^{n} \xi^i c_i$
    $v := \sum_{i=1}^{n} \xi^i v_i$
    check $e(C - vG_1, G_2) \stackrel{?}{=} e(\pi, xG_2 - zG_2)$
    return 1 if check passes, 0 otherwise
```

Figure 3.1: KZG polynomial commitment scheme

The following example demonstrates these abstract operations with concrete values, showing how polynomial commitments, evaluation proofs, and verification work in practice over a small finite field.

Example 3.2.1. Consider a simple KZG commitment over $\mathbb{F}_{23}$ with secret $x = 7$. The prover commits to polynomials $p_1(X) = 2X + 5$ and $p_2(X) = X^2 + 3$.

Setup The commitment key contains:

$$\begin{aligned}\mathsf{ck} &= (\{G_1, 7G_1, 49G_1\}) = (\{G_1, 7G_1, 3G_1\})\\ \mathsf{rk} &= (7G_2)\end{aligned}$$

Commit The prover computes commitments:

$$\begin{aligned}c_1 &= p_1(7)G_1 = (2\cdot 7+5)G_1 = 19G_1\\ c_2 &= p_2(7)G_1 = (7^2+3)G_1 = 6G_1\end{aligned}$$

Open For evaluation point $z = 5$ and opening challenge $\xi = 10$:

The prover forms the linear combination:

$$\begin{aligned}p(X) &= \xi p_1(X) + \xi^2 p_2(X)\\ &= 10(2X+5) + 100(X^2+3)\\ &= 8X^2 + 20X + 5\end{aligned}$$

Evaluating at $z = 5$:

$$p(5) = 8\cdot 25 + 20\cdot 5 + 5 = 305 = 6$$

The witness polynomial:

$$\begin{aligned}w(X) = \frac{p(X)-p(5)}{X-5} &= \frac{8X^2+20X+5-6}{X-5}\\ &= \frac{8X^2+20X-1}{X-5} = 8X+14\end{aligned}$$

The proof is $\pi = w(7)G_1 = (8\cdot 7+14)G_1 = 70G_1 = G_1$.

Check The verifier computes:

$$C = \xi c_1 + \xi^2 c_2 = 10 \cdot 19G_1 + 100 \cdot 6G_1 = 8G_1$$
$$v = \xi v_1 + \xi^2 v_2 = 10 \cdot 15 + 100 \cdot 5 = 6$$

where $v_1 = p_1(5) = 15$ and $v_2 = p_2(5) = 28 = 5$.

The verification checks:

$$e(C - vG_1, G_2) \stackrel{?}{=} e(\pi, xG_2 - zG_2)$$
$$e(2G_1, G_2) \stackrel{?}{=} e(G_1, 2G_2)$$

By bilinearity: $e(2G_1, G_2) = e(G_1, G_2)^2 = e(G_1, 2G_2)$

This example illustrates the core mechanism of KZG commitments. The commitment $c_i = p_i(x)G_1$ binds the prover to polynomial p_i without revealing it, as extracting p_i from c_i requires solving the discrete logarithm problem. The proof $\pi = w(x)G_1$ demonstrates knowledge of the committed polynomial's evaluation through polynomial division, that is the existence of $w(X) = \frac{p(X)-p(z)}{X-z}$ as a valid polynomial proves that $p(z) = v$. The verification equation $e(C - vG_1, G_2) = e(\pi, xG_2 - zG_2)$ leverages the bilinearity property to check this polynomial relation without knowing x or the polynomial coefficients.

The following proofs formalize these observations, demonstrating that this construction satisfies the required properties of a polynomial commitment scheme.

Theorem 3.2.2 (Completeness)**.** *The polynomial commitment scheme* PC *constructed above satisfies the completeness property.*

Proof. Assume that $\deg(\boldsymbol{p}) \leq D$ and $\boldsymbol{v} = \boldsymbol{p}(z)$. The goal is to show that $\mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}, \pi, \xi) = 1$. For each $i \in [n]$, the commitment c_i is computed as $c_i = p_i(x)G_1$. When computing the linear combination, $C = \sum_{i=1}^{n} \xi^i c_i = \sum_{i=1}^{n} \xi^i p_i(x)G_1 = p(x)G_1$, where $p(X) = \sum_{i=1}^{n} \xi^i p_i(X)$. Similarly, $v = \sum_{i=1}^{n} \xi^i v_i = \sum_{i=1}^{n} \xi^i p_i(z) = p(z)$. The witness polynomial $w(X) = \frac{p(X)-p(z)}{X-z}$ is a polynomial because $X - z$ divides $p(X) - p(z)$ by Corollary 2.3.4.

Now, verifying the pairing equation:

$$\begin{aligned}
e(C - vG_1, G_2) &= e(p(x)G_1 - p(z)G_1, G_2) \\
&= e((p(x) - p(z))G_1, G_2) \\
&= e\left(\frac{p(x) - p(z)}{x - z}(x - z)G_1, G_2\right) \\
&= e(w(x)(x - z)G_1, G_2) \\
&= e(w(x)G_1, (x - z)G_2) \\
&= e(\pi, xG_2 - zG_2)
\end{aligned}$$

Thus, the verification equation is satisfied, and PC.Check outputs 1, proving completeness. □

Theorem 3.2.3 (Succinctness)**.** *The polynomial commitment scheme* PC *constructed above satisfies the succinctness property.*

Proof. For a list of n polynomials, the scheme requires:

- Commitment size: n elements from $\mathbb{G}_1$, so $|\boldsymbol{c}| = n \cdot \mathsf{poly}(\lambda)$
- Evaluation proof size: 1 element from $\mathbb{G}_1$, so $|\pi| = \mathsf{poly}(\lambda)$
- Verification time: Two pairings and one scalar multiplication in $\mathbb{G}_1$ of size n, so $\mathsf{time}(\mathsf{Check}) = n \cdot \mathsf{poly}(\lambda)$

None of these quantities depends on the degree D of the committed polynomials, meeting the succinctness requirement. □

Now the construction satisfies the extractability property. First, an intermediary property called evaluation binding is introduced, then it is shown how this implies extractability in the algebraic group model.

Definition 3.2.4 (Evaluation Binding)**.** *A polynomial commitment scheme* PC *satisfies evaluation binding if for every maximum degree bound $D \in \mathbb{N}$ and efficient adversary*

$\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *the following probability is negligible in the security parameter* λ*:*

$$\Pr\left[\begin{array}{c|r} \boldsymbol{v} \neq \boldsymbol{v}' & \\ \wedge & (\mathsf{ck}, \mathsf{rk}) \leftarrow \mathsf{PC.Setup}(1^\lambda, D) \\ \mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}, \pi, \xi) = 1 & (\boldsymbol{c}, z, \boldsymbol{v}, \boldsymbol{v}', \cdot \leftarrow \mathcal{A}_1(\mathsf{ck}, \mathsf{rk}) \\ \wedge & \textit{Sample opening challenge } \xi \\ \mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}', \pi', \xi) = 1 & (\pi, \pi') \leftarrow \mathcal{A}_2(\xi) \end{array}\right]$$

Lemma 3.2.5 (Evaluation Binding)**.** *If the bilinear group sampler* BP *satisfies the SDH assumption, the KZG polynomial commitment scheme constructed above achieves evaluation binding.*

Proof. Suppose for contradiction that there exists an efficient adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ that breaks evaluation binding with non-negligible probability. An adversary $\mathcal{B}$ that breaks the SDH assumption can be constructed. The concrete construction can be found in [MBKM19, Lemma B.9].

$\mathcal{B}$ receives as input a bilinear group bp and a SDH challenge $\boldsymbol{\Sigma} = \{\{x^i G_1\}_{i=0}^{D}, xG_2\}$. It runs $\mathcal{A}_1(\mathsf{ck}, \mathsf{rk})$ where $\mathsf{ck} = (bp, \{x^i G_1\}_{i=0}^{D})$ and $\mathsf{rk} = (bp, xG_2)$ to obtain $(\boldsymbol{c}, z, \boldsymbol{v}, \boldsymbol{v}', \cdot$. Then it samples ξ and runs $\mathcal{A}_2(\xi)$ to obtain (π, π').

Let $C = \sum_{i=1}^{n} \xi^i c_i$, $v = \sum_{i=1}^{n} \xi^i v_i$, and $v' = \sum_{i=1}^{n} \xi^i v_i'$. Since $\boldsymbol{v} \neq \boldsymbol{v}'$, with high probability $v \neq v'$ for a random ξ. If both proofs pass verification, then:

$$e(C - vG_1, G_2) = e(\pi, xG_2 - zG_2)$$
$$e(C - v'G_1, G_2) = e(\pi', xG_2 - zG_2)$$

If $z = x$, then $\mathcal{B}$ can arbitrarily break the SDH assumption by outputting $(a, \frac{1}{z+a}G_1)$ for $a \in \mathbb{F}_q \setminus \{-z\}$. Otherwise, if $\pi \neq \pi'$, rearranging the equations:

$$e((v' - v)G_1, G_2) = e(\pi - \pi', xG_2 - zG_2)$$

This implies $(v' - v)G_1 = (x - z)(\pi - \pi')$, and thus:

$$\frac{1}{x - z} G_1 = \frac{\pi - \pi'}{v' - v}$$

Now, $\mathcal{B}$ output $(-z, \frac{\pi-\pi'}{v'-v})$, which breaks the SDH assumption.

If $\pi = \pi'$ but $v \neq v'$, the verification equations are contradictory, so this case cannot occur. Therefore, $\mathcal{B}$ successfully breaks the SDH assumption whenever $\mathcal{A}$ breaks evaluation binding, completing the proof. □

Theorem 3.2.6 (Extractability). *If the KZG polynomial commitment scheme satisfies evaluation binding, then it also satisfies extractability in the algebraic group model.*

Proof. The proof shows that for any algebraic adversary $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$ that extracts the committed polynomials. In the algebraic group model, when $\mathcal{A}$ outputs a group element $Y \in \mathbb{G}_1$, it must also provide the representation of Y in terms of previously seen group elements. For each commitment c_i output by $\mathcal{A}$, since $\mathcal{A}$ is algebraic, it also outputs coefficients $\{a_{i,j}\}_{j=0}^{D}$ such that:

$$c_i = \sum_{j=0}^{D} a_{i,j} \cdot (x^j G_1)$$

The extractor $\mathcal{E}$ defines the polynomial $p_i(X) = \sum_{j=0}^{D} a_{i,j} X^j$ for each commitment c_i. This ensures that $c_i = p_i(x)G_1$, which means p_i is a valid polynomial corresponding to the commitment c_i. Now it must be shown that if $\mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}, \pi, \xi) = 1$, then $\boldsymbol{v} = \boldsymbol{p}(z)$ with overwhelming probability. Suppose for contradiction that $\boldsymbol{v} \neq \boldsymbol{p}(z)$ with non-negligible probability. Let $p(X) = \sum_{i=1}^{n} \xi^i p_i(X)$ and $v = \sum_{i=1}^{n} \xi^i v_i$. The adversary can compute the honest proof $\pi' = w'(x)G_1$ where $w'(X) = \frac{p(X)-p(z)}{X-z}$, which would satisfy:

$$e(C - p(z)G_1, G_2) = e(\pi', xG_2 - zG_2)$$

However, by assumption, the adversary produced a proof π that satisfies:

$$e(C - vG_1, G_2) = e(\pi, xG_2 - zG_2)$$

Where $v \neq p(z)$ with high probability. This yields two valid proofs for different evaluations at the same point, contradicting the evaluation binding property. Therefore, with overwhelming probability, $\boldsymbol{v} = \boldsymbol{p}(z)$ whenever verification passes. Since both $\deg(\boldsymbol{p}) \leq D$ (by construction) and $\boldsymbol{v} = \boldsymbol{p}(z)$, the extractability property holds. □

A concrete implementation of the KZG polynomial commitment scheme in SageMath is provided in Appendix 1, demonstrating the practical construction of the algorithms described in this section.

The KZG Trapdoor The KZG commitment scheme's security relies critically on the secrecy of the trapdoor $x \in \mathbb{F}_q$ used in generating the SRS $\{x^i G_1\}_{i=0}^{D}$. Knowledge of x allows an adversary to break the binding property by creating valid opening proofs for arbitrary evaluations.

Specifically, given commitment $c = p(x)G_1$ for any polynomial p, an adversary knowing x can claim any evaluation $p(z) = v'$ (even when $p(z) \neq v'$) by constructing:

$$\pi' = \frac{p(x) - v'}{x - z} G_1$$

This proof satisfies the verification equation $e(c - v'G_1, G_2) = e(\pi', xG_2 - zG_2)$ despite the false claim. This demonstrates how trapdoor knowledge violates knowledge soundnessthe adversary can "prove" false statements without knowing a valid witness.

On the other hand, this same trapdoor enables the zero-knowledge property in zk-SNARKs. A simulator with access to x can generate valid proofs without knowing the witness, producing proofs indistinguishable from honestly generated ones.

3.3 Optimization

The KZG polynomial commitment scheme offers significant opportunities for optimization, particularly in scenarios involving multiple polynomial evaluations or verification of polynomial relations. These optimizations are crucial for practical zk-SNARKs constructions where verification efficiency is paramount.

Batching Pairing Equations The verification procedure $\mathsf{PC.Check}$ requires computing a pairing equation of the form $e(C - vG_1, G_2) = e(\pi, xG_2 - zG_2)$ for each polynomial evaluation. When a protocol requires verifying multiple such equations, the computational cost can become significant, as pairing operations are relatively expensive.

To reduce this cost, the pairing equation can be transformed to ensure that the $\mathbb{G}_2$ argument in the right-hand side pairing remains constant:

$$\begin{aligned} e(C - vG_1, G_2) &= e(\pi, xG_2 - zG_2) \\ &= e(\pi, xG_2) \cdot e(\pi, -zG_2) \\ &= e(\pi, xG_2) \cdot e(-z\pi, G_2) \end{aligned}$$

This transformation allows rewriting the verification equation as:

$$e(C - vG_1 + z\pi, G_2) = e(\pi, xG_2)$$

Now suppose n equations of this form need verification:

$$e(C_i - v_iG_1 + z_i\pi_i, G_2) = e(\pi_i, xG_2) \quad \text{for } i \in [n]$$

Instead of computing $2n$ pairings, these equations can be batched together to compute just 2 pairings. The verifier samples a random field element $r \leftarrow\$ \mathbb{F}_q$ and uses the identity $\prod_i e(G_i, H)^{r^i} = e(\sum_i r^i G_i, H)$ to check a single equation:

$$e\left(\sum_{i=1}^{n} r^i(C_i - v_iG_1 + z_i\pi_i), G_2\right) = e\left(\sum_{i=1}^{n} r^i\pi_i, xG_2\right)$$

This batching technique maintains security because if any individual equation $e(C_i - v_iG_1 + z_i\pi_i, G_2) \neq e(\pi_i, xG_2)$ for some $i \in [n]$, then the batched verification accepts with probability at most $\frac{n}{|\mathbb{F}_q|}$, which is negligible. Figure 3.2 summarizes the batch verification construction.

```
$\mathsf{PC.BatchCheck}(\mathsf{rk}, [\boldsymbol{c}_i]_{i=1}^n, [z_i]_{i=1}^n, [\boldsymbol{v}_i]_{i=1}^n, [\pi_i]_{i=1}^n, [\xi_i]_{i=1}^n) \to 0/1$

Sample randomness $r \leftarrow\$ \mathbb{F}_q$;   $L \leftarrow 0 \in \mathbb{G}_1$;   $R \leftarrow 0 \in \mathbb{G}_1$
For $i = 1, \ldots, n$ :
    $C_i \leftarrow \sum_{j=1}^{|\boldsymbol{c}_i|} \xi_i^j \cdot c_{i,j}$;   $v_i \leftarrow \sum_{j=1}^{|\boldsymbol{v}_i|} \xi_i^j \cdot v_{i,j}$
    $L \leftarrow L + r^i \cdot (C_i - v_i \cdot G_1 + z_i \cdot \pi_i)$;   $R \leftarrow R + r^i \cdot \pi_i$
check $e(L, G_2) \stackrel{?}{=} e(R, xG_2)$
return 1 if check passes, 0 otherwise
```

Figure 3.2: KZG batch verification

Opening Linear Combinations of Polynomials In zk-SNARKs constructions, the verifier often needs to check non-linear combinations of polynomials, for example:

$$p_1(X) + p_2(X)p_3(X) = p_4(X)$$

A standard approach is to evaluate each polynomial at a random point $z \in \mathbb{F}_q$ and verify $p_1(z) + p_2(z)p_3(z) = p_4(z)$ (which will be explained in detail in Section 5.2). This would require the prover to evaluate and open all four polynomials at the point z, sending four field elements to the verifier.

The optimization leverages the linearity of polynomial commitments and their *homomorphic* properties. That is, operations on commitments correspond to the same operations on the committed values. For KZG commitments, this means that for polynomials $f(X)$ and $g(X)$ with commitments c_f and c_g, and any scalar $a, b \in \mathbb{F}_q$, the commitment to $a \cdot f(X) + b \cdot g(X)$ equals $a \cdot c_f + b \cdot c_g$. This property enables computing commitments to linear combinations of polynomials directly from individual polynomial commitments, without knowing the polynomials themselves. The key insight is that only some polynomials need explicit evaluation, while others can be represented as linear combinations.

First, the prover evaluates just p_2 at point z to obtain $v_2 = p_2(z) \in \mathbb{F}_q$. Then, a derived polynomial is defined:

$$p_5(X) := p_1(X) + v_2 p_3(X) - p_4(X)$$

The original relation holds if and only if $p_5(z) = 0$. Now, the prover can execute a single batched opening:

$$\pi \leftarrow \mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p}, z, \xi)$$

where $\boldsymbol{p} = [p_2, p_5]$ and ξ a random challenge.

On the verification side, the verifier computes the commitment to p_5 as $c_5 = c_1 + v_2 c_3 - c_4$, and performs a single verification:

$$\mathsf{PC.Check}(\mathsf{rk}, \boldsymbol{c}, z, \boldsymbol{v}, \pi, \xi)$$

where $\boldsymbol{c} = [c_2, c_5]$ and $\boldsymbol{v} = [v_2, 0]$.

This approach reduces the communication from four field elements (evaluations of p_1, p_2, p_3, and p_4) to just one (v_2, since $p_5(z) = 0$ is implicit). The technique extends naturally to more complex polynomial relations, allowing significant efficiency gains in zk-SNARKs protocols that involve verifying numerous polynomial constraints.

CHAPTER 4
PREPROCESSING ZK-SNARKS WITH UNIVERSAL SRS

This chapter establishes the theoretical foundations of preprocessing zk-SNARKs with universal structured reference strings (SRS) that underlie the protocols in Chapter 6. The exposition begins with interactive proof systems, which establish the foundational model of a prover convincing a verifier through multi-round communication. It then introduces argument systems, which relax soundness requirements to hold only against computationally bounded adversaries, enabling the use of cryptographic assumptions for efficiency. The concept of preprocessing arguments with universal SRS emerges as a crucial framework that allows a single trusted setup to support multiple circuits. The combination of this argument system framework with polynomial commitment schemes like KZG from Chapter 3 yields the complete zk-SNARKs construction, as formally demonstrated in Marlin and implicitly utilized in Plonk.

4.1 Interactive proof systems

Interactive proof systems [GMR85] enable dynamic message exchange between prover and verifier for statement verification. A *language* L is a set of strings representing decision problems. For instance, PRIMES $= \{x \in \{0,1\}^* \mid x$ encodes a prime number$\}$.

Languages can be defined via relations: for $R \subseteq \{0,1\}^* \times \{0,1\}^*$, the language $L_R = \{x : \exists w$ such that $(x,w) \in R\}$ where x is the *instance* and w the *witness*.

Interactive proof systems extend NP verification by allowing multiple communication rounds with randomized verification. The system (P,V) must satisfy:

Completeness For every $x \in L$, the verifier accepts with high probability when interacting with the honest prover.

Soundness For every $x \notin L$ and any prover strategy P^*, the verifier rejects with high probability.

4.2 Argument systems

Argument systems [BCC88] are a variant of interactive proof systems where soundness holds only against computationally bounded provers. This distinction is crucial in cryptographic applications where adversaries typically have bounded computational resources.

An argument system for a language L is an interactive protocol (P, V) that satisfies:

Completeness For every $x \in L$, when the honest prover P and verifier V interact on input x, the verifier accepts with probability 1.

Computational soundness For every $x \notin L$ and any polynomial-time prover strategy P^*, when P^* and verifier V interact on input x, the verifier rejects with high probability.

Argument systems can be more efficient than proof systems because they leverage cryptographic assumptions to achieve succinctnessthe property that proof size and verification time are significantly smaller than the size of the statement being proven.

4.3 Preprocessing arguments with universal SRS

This section provides semi-formal definitions of preprocessing arguments with universal SRS. For fully formal mathematical definitions, readers are referred to [CHM$^+$20, Section 7].

An *indexed relation* $\mathcal{R}$ is a set of triples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ where $\mathbb{i}$ is the index, $\mathbb{x}$ is the instance, and $\mathbb{w}$ is the witness. The corresponding *indexed language* $\mathcal{L}(\mathcal{R})$ is the set of pairs $(\mathbb{i}, \mathbb{x})$ for which there exists a witness $\mathbb{w}$ such that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$.

For example, the indexed relation for satisfiable boolean circuits consists of triples where $\mathbb{i}$ is the description of a boolean circuit, $\mathbb{x}$ is a partial assignment to its input wires, and $\mathbb{w}$ is an assignment to the remaining wires that makes the circuit evaluate to 1.

Given a size bound $N \in \mathbb{N}$, $\mathcal{R}_N$ denotes the restriction of $\mathcal{R}$ to triples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ with $|\mathbb{i}| \leq N$.

A preprocessing argument with universal SRS is a tuple $\mathsf{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ with the following components:

$\mathsf{ARG}.\mathcal{G}(1^\lambda, N) \to \mathsf{srs}$ The generator algorithm takes a security parameter λ (in unary) and a size bound N, and outputs a SRS srs that supports indices of size up to N.

$\mathsf{ARG}.\mathcal{I}(\mathsf{srs}, \mathbb{i}) \to (\mathsf{ipk}, \mathsf{ivk})$ The indexer is a deterministic preprocessing algorithm that takes the SRS srs and an index $\mathbb{i}$ of size at most N, and outputs an index proving key ipk for the prover and an index verification key ivk for the verifier.

$\mathsf{ARG}.\mathcal{P}(\mathsf{ipk}, \mathbb{x}, \mathbb{w}) \leftrightarrow \mathsf{ARG}.\mathcal{V}(\mathsf{ivk}, \mathbb{x})$ The prover and verifier engage in an interactive protocol where the prover uses the index proving key ipk, an instance $\mathbb{x}$, and a witness $\mathbb{w}$, while the verifier uses the index verification key ivk and the instance $\mathbb{x}$. At the conclusion of the interaction, the verifier outputs a bit indicating acceptance (1) or rejection (0).

A preprocessing argument with universal SRS must satisfy two basic properties:

Completeness For any triple $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}_N$, when the honest prover with input $(\text{ipk}, \mathbb{x}, \mathbb{w})$ interacts with the honest verifier with input $(\text{ivk}, \mathbb{x})$, the verifier outputs 1 with probability 1. This ensures that honest provers with valid witnesses can always convince verifiers through the interactive protocol.

Soundness For any pair $(\mathbb{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R}_N)$ and any efficient adversarial prover strategy, the probability that the prover can convince the verifier to accept is negligible in the security parameter. This ensures that no efficient adversary can convince the verifier of a false statement.

4.4 From arguments to zk-SNARKs

To achieve the full power of preprocessing zk-SNARKs with universal SRS requires augmenting preprocessing arguments with additional properties and transforming them into non-interactive protocols. This section describes this progression and the associated terminology.

Knowledge soundness An argument system satisfies knowledge soundness if for any efficient adversarial prover strategy that causes the verifier to output 1 when interacting on input $(\mathbb{i}, \mathbb{x})$, there exists an efficient extraction algorithm that can extract a witness $\mathbb{w}$ such that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}_N$, except with negligible probability in the security parameter. This ensures that successful provers must actually know a valid witness, rather than simply exploiting the interactive nature of the protocol. This is stronger notion than basic soundness because soundness only guarantees that no prover can convince the verifier of a false statement, while knowledge soundness additionally ensures that any convincing prover possesses the underlying witness. The extraction property is crucial for cryptographic applications where protocols need to guarantee that provers have actual knowledge, not merely the ability to produce valid-looking proofs through other means.

Zero knowledge The interactive protocol reveals nothing about the witness $\mathbb{w}$ beyond the validity of the statement $(\mathbb{i}, \mathbb{x}) \in \mathcal{L}(\mathcal{R})$. Formally, there exists an efficient simulator that, without access to any witness, can simulate the entire transcript of the interaction between prover and verifier that is computationally indistinguishable from a real interaction. This property guarantees that the verifier learns nothing about the witness through the interaction beyond what is implied by the truth of the statement being proven.

Efficiency properties A preprocessing argument with universal SRS should satisfy several efficiency properties:

- **Index efficiency**: The running time of the prover $\mathcal{P}(\mathsf{ipk}, \mathbb{x}, \mathbb{w})$ is $\mathsf{poly}_\lambda(|\mathbb{i}|)$, independent of the size of the universal SRS.
- **Proof succinctness**: The size of the communication transcript between prover and verifier is $\mathsf{poly}(\lambda)$.
- **Verifier succinctness**: The running time of the verifier $\mathcal{V}(\mathsf{ivk}, \mathbb{x})$ is $\mathsf{poly}(\lambda + |\mathbb{x}|)$, independent of the size of the index $\mathbb{i}$.

The Fiat-Shamir transform To convert an interactive argument system into a non-interactive one, the Fiat-Shamir transform [FS86] is employed. This technique replaces the verifier's random challenges with deterministic hash function evaluations of the protocol transcript up to that point, effectively allowing the prover to simulate the entire interaction without requiring any input from the verifier beyond the initial statement.

When an interactive argument system is public-coin (meaning that all messages from the verifier are random challenges), the Fiat-Shamir transform can be applied by modeling the hash function as a random oracle. This transformation preserves completeness, and in many cases, also preserves soundness, knowledge soundness, and zero knowledge properties.

Terminology progression With these properties and transformations, a clear terminology progression emerges:

- An argument system with efficiency properties (succinctness) that is made non-interactive using the Fiat-Shamir transform becomes a *succinct non-interactive Argument* (SNARG).
- A SNARG with knowledge soundness becomes a *succinct non-interactive argument of knowledge* (SNARK).
- A SNARKs with zero knowledge becomes a *zero-knowledge SNARKs* (zk-SNARKs).

Therefore, a preprocessing zk-SNARKs with universal SRS is a non-interactive argument system that has a universal SRS, allows preprocessing of the index, satisfies completeness and knowledge soundness, achieves zero knowledge, and meets the efficiency requirements of succinctness for both proof size and verification time.

4.5 Setup Models for zk-SNARKs

The practical deployment of zk-SNARKs requires careful consideration of how the SRS is generated. While the trusted setup model is conceptually simple, it creates a critical vulnerability: if the single trusted party retains the trapdoor or is compromised, the entire system's soundness fails. This is particularly problematic for decentralized applications like cryptocurrencies where real monetary value is at stake and finding a universally trusted party is nearly impossible. Moreover, protocol upgrades requiring new trusted setups would repeatedly reintroduce these trust assumptions. These practical challenges motivate exploring stronger setup models that distribute or eliminate trust. The setup models for zk-SNARKs as follow:

Trusted Setup In the trusted setup model [CF01], a single party generates the SRS using secret randomness (the trapdoor) and then destroys this randomness. The entire system's security relies on this party's honesty. While simple to implement, this model creates a single point of failure: if the party retains the trapdoor or shares it with others, the soundness of the system is compromised. This weakness makes trusted setup unsuitable for decentralized applications where no single party can be universally trusted.

Subvertible Setup Subvertible setup model [BFS16, ABLZ17] represent the strongest theoretical setup model, maintaining security even when the setup algorithm itself may be maliciously constructed. Bellare et al. [BFS16] proved a fundamental impossibility: no proof system can simultaneously achieve subversion soundness and (even standard) zero-knowledge. This impossibility result shapes the landscape of achievable security properties and motivates the search for practical alternatives.

Updatable Setup The updatable setup model [GKM$^+$18] provides a practical middle ground between the simplicity of trusted setup and the theoretical strength of subvertible setup. Multiple parties sequentially contribute randomness to the SRS, with each party updating the existing SRS and proving they performed the update correctly. Security holds if at least one party in the update sequence is honest and destroys their randomness. The key insight is that certain SRS structures support homomorphic updates without revealing individual contributions. For KZG with SRS $\{x^i G_1\}_{i=0}^{D}$, a party updates x to $x \cdot \delta$ for their secret δ, transforming the SRS to $\{(x\delta)^i G_1\}_{i=0}^{D}$. The final trapdoor is the product of all contributions, unknown to any single party.

The setup models form a hierarchy where stronger models imply weaker ones. As shown by Groth et al. [GKM$^+$18], a proof system satisfying security with subvertible setup also satisfies security with updatable setup (Lemma 2), and one satisfying security with updatable setup also satisfies security with trusted setup (Lemma 1).

Theoretical Landscape and KZG The impossibility of achieving both subversion soundness and zero-knowledge [BFS16] establishes the fundamental limits of what can be achieved in theory. The KZG polynomial commitment scheme, as analyzed in Sonic [MBKM19], achieves subversion zero-knowledge: the zero-knowledge property holds even against adversaries who maliciously generate the SRS. However, KZG can only achieve updatable soundness, not subversion soundness. This positions updatable setups as the strongest practically achievable model: they provide the maximum soundness guarantee possible while preserving zero-knowledge.

Why Updatable Setups Matter The updatable setup model represents the optimal balance between security and practicality. It circumvents the theoretical impossibility while enabling real-world deployment through ceremonies where multiple participants can contribute randomness. The assumption that at least one participant is honest is far more realistic than trusting a single party. Furthermore, the public verifiability of updates provides transparency, and new participants can join without invalidating previous contributions. The KZG polynomial commitment scheme's inherent monomial structure naturally supports these updatable ceremonies, making it the foundation for modern preprocessing zk-SNARKs that achieve both strong security guarantees and practical deployability.

CHAPTER 5
TECHNIQUES IN ZK-SNARKS

The construction of efficient zk-SNARKs relies on several fundamental techniques that transform computational problems into algebraic statements that can be verified succinctly. These techniques serve as building blocks for the practical zk-SNARKs constructions covered in Chapter 6.

This chapter begins with polynomial encoding, which provides a way to represent computational statements as polynomial relations. Next, polynomial identity testing is examined, a technique that enables efficient verification of polynomial relations. The univariate sumcheck that will be used in Marlin [CHM$^+$20] is then explored, which allows for efficient verification of claims about polynomial sums over large domains. Finally, the permutation argument is presented, an essential component for enforcing the ”copy constraint” that will be used in Plonk [GWC19].

5.1 Polynomial encoding

Many zk-SNARKs constructions rely on encoding computational problems into polynomial relations, a critical step that allows leveraging the algebraic properties of polynomials and the efficiency of the KZG commitment scheme. This polynomial encoding transforms computational statements, such as circuit satisfaction or constraint systems, into relationships between polynomials that can be efficiently verified.

The general approach begins with representing a computation as a constraint system over a finite field $\mathbb{F}_q$. Different zk-SNARKs protocols may use different types of constraint systemssuch as Rank-1 Constraint Systems (R1CS) [GGPR13], arithmetic circuits, or custom constraint formats like those in Plonk [GWC19]. But, the fundamental principle remains the same. The computational statement and its witness (the solution or proof) are encoded as polynomials whose properties and relationships capture the original computational problem.

Consider working with multiplicative subgroup $H = \{1, \mathrm{g}, \mathrm{g}^2, \ldots, \mathrm{g}^{n-1}\}$ of $\mathbb{F}_q$, where g is generator of H or equivalently a primitive n-th root of unity. The values in the constraint system (such as witness values, constraint coefficients, or circuit wire values) are mapped to evaluations of polynomials at points in this subgroup. Through interpolation, polynomials are constructed that pass through these specific evaluation points, effectively encoding the entire computation.

The field $\mathbb{F}_q$ and multiplicative subgroup H require specific structural properties to enable FFT-based polynomial operations. As detailed in Section 2.5, q must be chosen such that $q-1$ contains a large power of 2 factor, allowing H to have size $n = 2^k$ for some k. This power-of-2 structure is essential for the FFT algorithm to achieve $O(n \log n)$ complexity for polynomial interpolation, making these encodings computationally feasible even for large-scale constraint systems.

5.2 Polynomial identity testing

Polynomial identity testing enables efficient verification of polynomial relations. Rather than checking if $f(X) = g(X)$ by comparing all coefficients, evaluate both at random $r \leftarrow\$ \mathbb{F}$ and check $f(r) = g(r)$. By Corollary 2.3.7, if $f \neq g$ with deg$(f-g) \leq d$, then Pr$[f(r) = g(r)] \leq \frac{d}{|\mathbb{F}|}$.

In zk-SNARKs, verification often involves relations over domains. For $H = \langle \mathbf{g} \rangle \subseteq \mathbb{F}_q$, to verify $f(a) = g(a)$ for all $a \in H$, use the vanishing polynomial $v_H(X) = X^n - 1$. The relation holds if and only if $v_H(X)$ divides $f(X) - g(X)$, meaning there exists $q(X)$ such that:

$$f(X) - g(X) = v_H(X)q(X)$$

In a zk-SNARK protocol using the KZG commitment scheme, the verification proceeds as follows:

1. The prover commits to polynomials f, g, and q using KZG commitments.
2. The verifier samples a random challenge point $r \leftarrow\$ \mathbb{F}_q$.
3. The prover provides evaluations $f(r)$, $g(r)$, and $q(r)$.
4. The verifier samples an opening challenge $\xi \leftarrow\$ \mathbb{F}_q$.
5. The prover provides KZG opening proofs for the evaluations.
6. The verifier checks that the KZG opening proofs verify correctly and that the

polynomial identity holds at point r:

$$f(r) - g(r) \stackrel{?}{=} v_H(r)q(r)$$

If this check passes, by Lemma 2.3.8, the verifier is convinced with overwhelming probability that the identity holds as a formal polynomial identity, which implies that the original relation holds on all of H. The soundness error is negligible since r is sampled from the exponentially large field $\mathbb{F}_q$.

This approach transforms verification of a computation involving numerous constraints into checking a single equation at a random point. The verifier can compute $v_H(r) = r^n - 1$ efficiently in $O(\log n)$ time using square-and-multiply exponentiation, making the verification process highly efficient compared to the size of H.

Remark 5.2.1. The polynomial identity testing technique naturally extends to more complex polynomial relations involving multiple polynomials. For example, to verify a multiplicative relation $f(a)g(a) = h(a)$ for all $a \in H$, the protocol checks if $v_H(X)$ divides $f(X)g(X) - h(X)$. Similarly, for quadratic relations like $f(a)^2 - g(a)h(a) = 0$ or multi-polynomial constraints like $f(a) + g(a) + h(a) = j(a)$, the same pattern is followed: reformulate as a divisibility check and evaluate at a random point. This flexibility allows polynomial identity testing to handle the diverse constraint types that arise in zk-SNARK constructions, including arithmetic circuit constraints, R1CS relations, and custom constraint systems like those in PLONK.

5.3 Univariate sumcheck

The univariate sumcheck provides an efficient method to verify the sum of a polynomial's evaluations over a large domain. For a polynomial $p(X)$ of degree at most D, the protocol verifies claims about the sum $\sum_{a \in H} p(a)$. The following results are adapted from [Tha22]. First, a key property regarding polynomial sums over multiplicative subgroups is established.

Lemma 5.3.1. *Let H be a multiplicative subgroup of $\mathbb{F}_q$ of size n. For any polynomial $f(X)$ of degree less than n, the following holds:*

$$\sum_{a \in H} f(a) = f(0) \cdot |H|$$

In other words, the sum of evaluations of f over H depends only on the constant term of f.

Proof. By linearity, it suffices to prove the result for monomials. For any monomial X^m with $0 < m < n$, it must be shown that $\sum_{a \in H} a^m = 0$.

Since H is a multiplicative subgroup of order n, it is cyclic. Let g be a generator of H. Then:

$$\sum_{a \in H} a^m = \sum_{j=0}^{n-1} (\mathrm{g}^j)^m = \sum_{j=0}^{n-1} \mathrm{g}^{jm}$$

If $\gcd(m,n) = 1$, then g^m is also a generator of H. As j ranges from 0 to $n-1$, the values g^{jm} simply enumerate all elements of H in a different order. If this sum is multiplied by $\mathrm{g}^m \neq 1$:

$$\mathrm{g}^m \sum_{j=0}^{n-1} \mathrm{g}^{jm} = \sum_{j=0}^{n-1} \mathrm{g}^{(j+1)m} = \sum_{j=0}^{n-1} \mathrm{g}^{jm}$$

where the last equality follows because multiplication by g^m merely permutes the elements of H. This implies $(\mathrm{g}^m - 1) \cdot \sum_{a \in H} a^m = 0$. Since $\mathrm{g}^m \neq 1$ in a field, it must be that $\sum_{a \in H} a^m = 0$.

If $\gcd(m,n) = d > 1$, then g^m generates a proper subgroup of H of order n/d. In this case, as j ranges from 0 to $n-1$, each element in this subgroup appears exactly d times. By a similar argument as above, the sum over this subgroup is zero, so the overall sum is also zero.

For the monomial $X^0 = 1$, clearly $\sum_{a \in H} 1 = n = |H|$.

By linearity, for any polynomial $f(X) = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ with $\deg(f) < n$:

$$\begin{aligned}
\sum_{a \in H} f(a) &= \sum_{a \in H} (f_0 + f_1 a + \cdots + f_{n-1} a^{n-1}) \\
&= f_0 \sum_{a \in H} 1 + f_1 \sum_{a \in H} a + \cdots + f_{n-1} \sum_{a \in H} a^{n-1} \\
&= f_0 \cdot |H| + f_1 \cdot 0 + \cdots + f_{n-1} \cdot 0 \\
&= f_0 \cdot |H| = f(0) \cdot |H|
\end{aligned}$$

□

From this lemma, a key result for the sumcheck protocol that generalizes to arbitrary claimed sums σ can be derived.

Theorem 5.3.2. *Let H be a multiplicative subgroup of $\mathbb{F}_q$ of size n, and let $f(X)$ be a polynomial of degree at most D. Then $\sum_{a\in H} f(a) = \sigma$ if and only if there exist polynomials $h(X) \in \mathbb{F}_q^{<D-n+1}[X]$ and $g(X) \in \mathbb{F}_q^{<n-1}[X]$ satisfying:*

$$f(X) = h(X)\cdot v_H(X) + X\cdot g(X) + \frac{\sigma}{|H|}$$

where $v_H(X) = X^n - 1$ is the vanishing polynomial of H.

Proof. First, suppose the equation holds. Then for any $a \in H$:

$$\begin{aligned} f(a) &= h(a)\cdot v_H(a) + a\cdot g(a) + \frac{\sigma}{|H|} \\ &= h(a)\cdot 0 + a\cdot g(a) + \frac{\sigma}{|H|} \\ &= a\cdot g(a) + \frac{\sigma}{|H|} \end{aligned}$$

Summing over all elements in H:

$$\begin{aligned} \sum_{a\in H} f(a) &= \sum_{a\in H} a\cdot g(a) + \sum_{a\in H} \frac{\sigma}{|H|} \\ &= \sum_{a\in H} a\cdot g(a) + \sigma \end{aligned}$$

The polynomial $a\cdot g(a)$ has no constant term, so by Lemma 5.3.1, $\sum_{a\in H} a\cdot g(a) = 0$. Therefore, $\sum_{a\in H} f(a) = \sigma$.

Conversely, suppose $\sum_{a\in H} f(a) = \sigma$. Define a new polynomial $f'(X) = f(X) - \frac{\sigma}{|H|}$.

Then:

$$\begin{aligned}\sum_{a\in H} f'(a) &= \sum_{a\in H} f(a) - \sum_{a\in H} \frac{\sigma}{|H|} \\ &= \sigma - \frac{\sigma}{|H|}\cdot|H| \\ &= 0\end{aligned}$$

By polynomial division, $f'(X) = h(X)\cdot v_H(X) + r(X)$ where $\deg(r) < n$. Since $v_H(a) = 0$ for all $a \in H$:

$$\begin{aligned}\sum_{a\in H} r(a) &= \sum_{a\in H} f'(a) - \sum_{a\in H} h(a)\cdot v_H(a) \\ &= 0 - 0 = 0\end{aligned}$$

By Lemma 5.3.1, $\sum_{a\in H} r(a) = r(0)\cdot|H|$, so $r(0) = 0$. This means $r(X)$ has no constant term and can be written as $r(X) = X\cdot g(X)$ for some polynomial $g(X)$ with $\deg(g) < n-1$.

Therefore, $f'(X) = h(X)\cdot v_H(X) + X\cdot g(X)$, which implies:

$$f(X) = h(X)\cdot v_H(X) + X\cdot g(X) + \frac{\sigma}{|H|}$$

This completes the proof. $\square$

This characterization serves as the foundation for the univariate sumcheck protocol, which can be efficiently implemented using the KZG polynomial commitment scheme:

1. The prover commits to polynomials $f(X)$, $h(X)$, and $g(X)$ using KZG commitments and claims that $\sum_{a\in H} f(a) = \sigma$.
2. The verifier samples a random challenge point $r \leftarrow\$ \mathbb{F}_q$.
3. The prover provides evaluations $f(r)$, $h(r)$, and $g(r)$.
4. The verifier samples an opening challenge $\xi \leftarrow\$ \mathbb{F}_q$.
5. The prover provides KZG opening proofs for the evaluations.

6. The verifier checks that the KZG opening proofs verify correctly and that the polynomial identity holds at point r:

$$f(r) \stackrel{?}{=} h(r)v_H(r) + r \cdot g(r) + \frac{\sigma}{|H|}$$

If this check passes, the verifier is convinced with overwhelming probability by Lemma 2.3.8 that the claimed sum is correct. The verifier gains efficiency by reducing verification of a sum with potentially many terms to checking a single identity at a random point, coupled with the succinctness properties of the KZG commitment scheme.

5.4 Permutation Argument

This section explores the technique introduced in Plonk that uses a permutation argument operating on univariate polynomials evaluated over multiplicative subgroups. First, the meaning of "extended permutations" that work across multiple polynomials is defined. Consider a multiplicative subgroup $H = \langle \mathrm{g} \rangle$ of order n in $\mathbb{F}_q$, and a permutation $\sigma : [kn] \to [kn]$ for some positive integer k. For polynomials $f_1, \ldots, f_k, g_1, \ldots, g_k \in \mathbb{F}_q^{<n}[X]$, the notation $(g_1, \ldots, g_k) = \sigma(f_1, \ldots, f_k)$ means the following:

Define the sequences $(f_{(1)}, \ldots, f_{(kn)})$ and $(g_{(1)}, \ldots, g_{(kn)})$ in $\mathbb{F}_q^{kn}$ by:

$$f_{((j-1)\cdot n+i)} := f_j(\mathrm{g}^i), \quad g_{((j-1)\cdot n+i)} := g_j(\mathrm{g}^i)$$

for each $j \in [k]$, $i \in [n]$. Then $(g_1, \ldots, g_k) = \sigma(f_1, \ldots, f_k)$ if and only if $g_{(i)} = f_{(\sigma(i))}$ for each $i \in [kn]$.

In other words, the extended permutation relates the evaluations of multiple polynomials at points in H according to the permutation σ. This generalization allows capturing more complex relationships between polynomials, which is necessary for representing the wiring constraints in arithmetic circuits.

Before presenting the main result, a key lemma that provides the foundation for testing whether two sequences follow a permutation relationship is first established.

Lemma 5.4.1. *Fix any permutation σ of $[n]$, and any sequences $(a_1, \ldots, a_n)$, $(b_1, \ldots, b_n) \in$*

$\mathbb{F}_q^n$. If

$$\prod_{i\in[n]}(a_i+\beta\cdot i+\gamma)=\prod_{i\in[n]}(b_i+\beta\cdot\sigma(i)+\gamma)$$

with probability larger than $\frac{n}{|\mathbb{F}_q|}$ over uniform $\beta,\gamma\in\mathbb{F}_q$, then $b_i=a_{\sigma(i)}$ for all $i\in[n]$.

Proof. By Lemma 2.3.8 and the assumption, the following equality of polynomials holds in $\mathbb{F}_q[X,Y]$:

$$\prod_{i=1}^{n}(a_i+iX+Y)\equiv\prod_{i=1}^{n}(b_i+\sigma(i)X+Y)$$

Because $\mathbb{F}_q[X,Y]$ is a unique factorization domain[1] where the invertible elements are exactly $\mathbb{F}_q^*$. The linear factors in the above products are irreducible. Thus, there must be a one-to-one map between the factors of each side, such that each factor on the left side equals a constant multiple of a factor on the right side.

Moreover, since the coefficient of Y in all terms on both sides is 1, these constant multiples must all be 1. Therefore, this mapping must match each factor on the left side to one on the right side with the same coefficient for X. In summary, for all $i\in[n]$:

$$a_{\sigma(i)}+\sigma(i)X+Y\equiv b_i+\sigma(i)X+Y$$

This implies that $b_i=a_{\sigma(i)}$ for all $i\in[n]$. □

Based on this lemma, the main theorem that forms the foundation of the permutation argument in Plonk can now be stated. The notation $L_i(X)$ denotes the i-th Lagrange basis polynomial for H at point g^i, which satisfies $L_i(\mathrm{g}^i)=1$ and $L_i(a)=0$ for all $a\in H\setminus\{\mathrm{g}^i\}$.

Theorem 5.4.2. *Let $H=\langle\mathrm{g}\rangle$ be a multiplicative subgroup of $\mathbb{F}_q$ of order n, and let σ be a permutation on $[kn]$. Let $f_1,\dots,f_k,g_1,\dots,g_k\in\mathbb{F}_q^{<n}[X]$ be polynomials of degree less than n. Define:*

[1]A unique factorization domain (UFD) is a ring in which every non-zero non-unit element can be written as a product of irreducible elements in an essentially unique way. Multivariate polynomial rings over fields are UFDs [MMS96, Corollary 13.2.11].

$$f'_j(X) = f_j(X) + \beta \cdot S_{\mathsf{ID}_j}(X) + \gamma$$
$$g'_j(X) = g_j(X) + \beta \cdot S_{\sigma_j}(X) + \gamma$$

where $S_{\mathsf{ID}_j}(\mathbf{g}^i) = (j-1)\cdot n + i$ *and* $S_{\sigma_j}(\mathbf{g}^i) = \sigma((j-1)\cdot n + i)$ *for each* $i \in [n]$. *Let* $f'(X) = \prod_{j\in[k]} f'_j(X)$ *and* $g'(X) = \prod_{j\in[k]} g'_j(X)$. *If there exists a polynomial* $Z(X)$ *of degree less than* n *such that with non-negligible probability:*

1. $L_1(a)\cdot(Z(a)-1) = 0$ *for all* $a \in H$, *and*
2. $Z(a)\cdot f'(a) = g'(a)\cdot Z(a\cdot\mathbf{g})$ *for all* $a \in H$

then $(g_1,\ldots,g_k) = \sigma(f_1,\ldots,f_k)$.

Proof. The first condition ensures that $Z(\mathbf{g}) = 1$. From the second condition, by induction, for each $i \in [n]$:

$$Z(\mathbf{g}^{i+1}) = \prod_{1\le j\le i} \frac{f'(\mathbf{g}^j)}{g'(\mathbf{g}^j)}$$

Since $\mathbf{g}^{n+1} = \mathbf{g}$:

$$Z(\mathbf{g}) = Z(\mathbf{g}^{n+1}) = \prod_{j\in[n]} \frac{f'(\mathbf{g}^j)}{g'(\mathbf{g}^j)}$$

This implies:

$$1 = \prod_{j\in[n]} \frac{f'(\mathbf{g}^j)}{g'(\mathbf{g}^j)}$$
$$\prod_{j\in[n]} f'(\mathbf{g}^j) = \prod_{j\in[n]} g'(\mathbf{g}^j)$$
$$\prod_{j\in[k]}\prod_{i\in[n]} f'_j(\mathbf{g}^i) = \prod_{j\in[k]}\prod_{i\in[n]} g'_j(\mathbf{g}^i)$$
$$\prod_{j\in[k]}\prod_{i\in[n]} (f_j(\mathbf{g}^i) + \beta\cdot S_{\mathsf{ID}_j}(\mathbf{g}^i) + \gamma) = \prod_{j\in[k]}\prod_{i\in[n]} (g_j(\mathbf{g}^i) + \beta\cdot S_{\sigma_j}(\mathbf{g}^i) + \gamma)$$
$$\prod_{i\in[kn]} (f_{(i)} + \beta\cdot i + \gamma) = \prod_{i\in[kn]} (g_{(i)} + \beta\cdot\sigma(i) + \gamma)$$

By Lemma 5.4.1 this implies $g_{(i)} = f_{(\sigma(i))}$ for all $i \in [kn]$, which means $(g_1, \dots, g_k) = \sigma(f_1, \dots, f_k)$. $\square$

The permutation argument can be efficiently implemented using the KZG polynomial commitment scheme:

1. The prover commits to polynomials $f_1, \dots, f_k$, $g_1, \dots, g_k$ using KZG commitments.
2. The verifier sends random challenges $\beta, \gamma \in \mathbb{F}_q$.
3. The prover computes the polynomials f'_j and g'_j for each $j \in [k]$, as well as their products f' and g'. The prover computes Z and quotient polynomials q_1, q_2 such that:

$$L_1(X)(Z(X) - 1) = v_H(X)q_1(X)$$
$$Z(X)f'(X) - g'(X)Z(X\mathbf{g}) = v_H(X)q_2(X)$$

   The prover commits to Z, q_1, and q_2 using KZG commitments.
4. The verifier samples a random challenge point $r \in \mathbb{F}_q$.
5. The prover provides evaluations $f_j(r)$, $g_j(r)$ for each $j \in [k]$, $Z(r)$, $Z(r \cdot \mathbf{g})$, $q_1(r)$, and $q_2(r)$.
6. The verifier samples an opening challenge $\xi_1, \xi_2 \leftarrow\$ \mathbb{F}_q$ to get opening to two evaluation point r and $r \cdot \mathbf{g}$.
7. The prover provides KZG opening proofs for all evaluations.
8. The verifier uses the homomorphic properties of KZG commitments to verify the polynomial identities without directly computing the products $f'(r)$ and $g'(r)$. This requires linearization techniques as discussed in Section 3.3, with specific implementation details shown in the Plonk protocol in Section 6.2.
9. The verifier checks that the KZG opening proofs verify correctly and that the polynomial identities hold at point r:

$$L_1(r)(Z(r) - 1) \stackrel{?}{=} v_H(r)q_1(r)$$
$$Z(r)f'(r) - g'(r)Z(r\mathbf{g}) \stackrel{?}{=} v_H(r)q_2(r)$$

If these checks pass, the verifier is convinced with overwhelming probability that the permutation relation holds.

CHAPTER 6
MAIN ZK-SNARKS PROTOCOL

This chapter presents two prominent zk-SNARKs protocols that utilize the KZG polynomial commitment scheme: Marlin [CHM$^+$20] and Plonk [GWC19]. Both protocols achieve universal and updatable SRS while taking different approaches to arithmetization and proof construction.

The exposition for each protocol follows a consistent structure: constraint system definition, polynomial encoding strategy, optimization through linearization, and finally the complete non-interactive construction using KZG commitments. This systematic presentation highlights both the modular nature of zk-SNARKs design and the flexibility of polynomial commitment schemes as cryptographic building blocks.

Section 6.1 presents the Marlin protocol, followed by Plonk in Section 6.2. A detailed comparison of their performance characteristics and design trade-offs is provided in Section 6.3.

6.1 Marlin

Marlin is a zk-SNARKs with universal and updatable SRS. It was introduced by Chiesa et al. [CHM$^+$20] as an improvement over previous zk-SNARKs protocols, achieving better prover efficiency, verification time, and proof size. Marlin builds on the algebraic holographic proof (AHP) for Rank-1 Constraint Satisfiability (R1CS) [GGPR13] combined with the KZG polynomial commitment scheme from Chapter 3.

The complete Marlin protocol involves multiple rounds of interaction between the prover and verifier, where polynomial commitments are exchanged and random challenges are generated to ensure soundness. Figure 6.1 presents the complete protocol flow, showing the five rounds of communication. Understanding this interactive structure is essential before seeing how the protocol becomes non-interactive in practice.

Constraint system The R1CS indexed relation is formally defined as the set of all triples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ where $\mathbb{i} = (\mathbb{F}_q, H, K, A, B, C)$ is the index, $\mathbb{F}_q$ is a finite field, H and

K are subsets of $\mathbb{F}$ (typically multiplicative subgroups) of sizes n and m, A, B, C are $H \times H$ matrices over $\mathbb{F}$ with $|K| \geq \max\{\|A\|, \|B\|, \|C\|\}$ where $\|M\|$ denotes the number of non-zero entries in matrix M. Define $z := (x, w)$ as a vector in $\mathbb{F}^H$, where $x := \mathbb{x}$ corresponds to the instance values and $w := \mathbb{w}$ corresponds to the witness values. Here, $\mathbb{F}^H$ denotes the set of vectors indexed by elements in H. The triple $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ satisfies the relation when z fulfills the constraint $Az \circ Bz = Cz$, where $\circ$ denotes the entry-wise product of vectors. For efficiency, the protocol typically assumes that H and K are "FFT-friendly" (having smooth sizes), allowing for fast polynomial operations.

Throughout the protocol assumes that H and K come equipped with bijections $\phi_H : H \to [|H|]$ and $\phi_K : K \to [|K|]$ that are computable in linear time. The protocol uses the structure of the domain H to partition the assignment into instance and witness values. Define $H[\leq k] := \{\kappa \in H : 1 \leq \phi_H(\kappa) \leq k\}$ and $H[> k] := \{\kappa \in H : \phi_H(\kappa) > k\}$ to denote the first k elements in H and the remaining elements, respectively, according to the ordering induced by ϕ_H. The instance x is in $\mathbb{F}^{H[\leq |x|]}$ and the witness w is in $\mathbb{F}^{H[>|x|]}$.

Example 6.1.1 (R1CS for polynomial evaluation)**.** Consider proving knowledge of $X \in \mathbb{F}_{23}$ such that $Y = X^3 + 2X + 5$ where $Y = 15$ is public. The witness is $X = 3$.

The computation uses intermediate values: $w_1 = X = 3$, $w_2 = X^2 = 9$, $w_3 = X^3 = 4$, $w_4 = 2X = 6$. The full assignment vector is $z = (x, w) = [1, Y, w_1, w_2, w_3, w_4] = [1, 15, 3, 9, 4, 6]$.

The R1CS constraints efficiently express the polynomial evaluation:

1. $w_1 \cdot w_1 = w_2$ (computing X^2)
2. $w_2 \cdot w_1 = w_3$ (computing X^3)
3. $2 \cdot w_1 = w_4$ (computing $2X$)
4. $(5 + w_3 + w_4) \cdot 1 = Y$ (final addition)

The constraint matrices are:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Each row corresponds to one constraint, satisfying $(Az)_i \cdot (Bz)_i = (Cz)_i$. For instance, constraint 4: $(Az)_4 = 5+4+6 = 15$ and $(Bz)_4 = 1$, giving $15 \cdot 1 = 15 = (Cz)_4$.

Note that these matrices have dimensions 4×6 (4 constraints, 6 variables), not square as in the general protocol description. In practice, the protocol pads such matrices with zero rows to form square $|H| \times |H|$ matrices, where $|H| \geq \max\{4,6\}$, enabling uniform polynomial encoding over the domain H.

Polynomial encoding Marlin represents the R1CS constraint system as a set of polynomial relations. For efficient encoding, matrices are represented in their sparse form. The indexer processes these matrices as follows: For each matrix $M \in \{A,B,C\}$, three univariate polynomials $\hat{\text{row}}_M$, $\hat{\text{col}}_M$, and $\hat{\text{val}}_M$ of degree less than $|K|$ are constructed such that $\hat{\text{row}}_M(\kappa)$ gives the row index of the κ-th non-zero entry, $\hat{\text{col}}_M(\kappa)$ gives the column index of the κ-th non-zero entry, and $\hat{\text{val}}_M(\kappa)$ gives the value of the κ-th non-zero entry. These polynomials allow efficient representation of the sparse matrices.

To understand the matrix representation, it's essential to first introduce the bivariate polynomial $u_H(X,Y)$ defined as:

$$u_H(X,Y) := \frac{v_H(X) - v_H(Y)}{X - Y}$$

where $v_H(X)$ is the vanishing polynomial of the subgroup H. This polynomial $u_H(X,Y)$ has individual degree $|H|-1$ and has the useful property that it vanishes on the square $H \times H$ except on the diagonal, where it takes non-zero values $(u_H(a,a))_{a\in H}$. For multiplicative subgroups, $u_H(X,Y) = \frac{X^n - Y^n}{X-Y}$ and $u_H(X,X) = nX^{n-1}$.

Using these bivariate polynomials, the low-degree extension of matrix M can be expressed in two equivalent ways. First, as:

$$\hat{M}(X,Y) = \sum_{\kappa\in K} u_H(X, \hat{\text{row}}_M(\kappa))u_H(Y, \hat{\text{col}}_M(\kappa))\hat{\text{val}}_M(\kappa)$$

And this can be equivalently rewritten as:

$$\hat{M}(X,Y) = \sum_{\kappa \in K} \frac{v_H(X)}{(X - \hat{\text{row}}_M(\kappa))} \cdot \frac{v_H(Y)}{(Y - \hat{\text{col}}_M(\kappa))} \cdot \hat{\text{val}}_M(\kappa)$$

where the equivalence follows because $v_H(\hat{\text{row}}_M(\kappa)) = v_H(\hat{\text{col}}_M(\kappa)) = 0$ since $\hat{\text{row}}_M(X)$ and $\hat{\text{col}}_M(X)$ map K to H, and v_H vanishes on H.

The functions that define these polynomials are constructed as follows: For every $\kappa \in K$ with $1 \leq \phi_K(\kappa) \leq \|M\|$, row($\kappa$) gives the row index of the $\phi_K(\kappa)$-th nonzero entry in M, col(κ) gives the column index, and val(κ) is the value of this entry divided by $u_H(\text{row}(\kappa), \text{row}(\kappa)) u_H(\text{col}(\kappa), \text{col}(\kappa))$. By construction, the polynomial $\hat{M}(X,Y)$ agrees with the matrix M everywhere on the domain $H \times H$ and is the unique low-degree extension of M with individual degree less than $|H|$.

The prover begins by encoding the full assignment $z := (x, w)$ as a polynomial. Let $\hat{x}(X)$ be the polynomial of degree less than $|x|$ that agrees with the instance x in the domain $H[\leq |x|]$. The prover computes a "shifted witness" $\overline{w}$ for $H[> |x|]$ according to $\overline{w}(\gamma) := \frac{w(\gamma) - \hat{x}(\gamma)}{v_{H[\leq |x|]}(\gamma)}$, where $v_{H[\leq |x|]}(X)$ is the vanishing polynomial of the subset $H[\leq |x|]$ and $\forall \gamma \in H[> |x|]$. The prover then selects a random polynomial $\hat{w}(X) \in \mathbb{F}_q^{<|w|+b}[X]$ that agrees with $\overline{w}$ on $H[> |x|]$, where b is a parameter for zero-knowledge. The full assignment polynomial is constructed as $\hat{z}(X) := \hat{w}(X) v_{H[\leq |x||]}(X) + \hat{x}(X)$.

The prover computes the three linear combinations $z_A := Az$, $z_B := Bz$, and $z_C := Cz$. The prover then creates random polynomials $\hat{z}_A(X)$, $\hat{z}_B(X)$, $\hat{z}_C(X) \in \mathbb{F}_q^{<|H|+b}[X]$ that agree with z_A, z_B, and z_C on H, respectively.

Polynomial identity testing The Marlin protocol aims to prove the existence of a witness w such that the R1CS relation is satisfied. This is done by encoding the assignment and constraint matrices as polynomials, and proving two key conditions:

1. Entry-wise product condition: $\forall \kappa \in H, \hat{z}_A(\kappa)\hat{z}_B(\kappa) - \hat{z}_C(\kappa) = 0$.
2. Linear relation condition: $\forall M \in \{A, B, C\}, \forall \kappa \in H, \hat{z}_M(\kappa) = \sum_{\iota \in H} \hat{M}(\kappa, \iota)\hat{z}(\iota)$.

To verify the entry-wise product constraint $z_A \circ z_B = z_C$, the prover computes a polynomial $h_0(X)$ such that $\hat{z}_A(X)\hat{z}_B(X) - \hat{z}_C(X) = h_0(X)v_H(X)$. This ensures that the constraint is satisfied on the domain H. The verifier can generate random challenge β_1 and validate

$$\hat{z}_A(\beta_1)\hat{z}_B(\beta_1) - \hat{z}_C(\beta_1) \stackrel{?}{=} h_0(\beta_1)v_H(\beta_1)$$

Then, to verify linear relation constraint the prover samples a random $s(X) \in \mathbb{F}_q^{<2|H|+b-1}[X]$ and computes its sum $\sigma_1 := \sum_{\kappa \in H} s(\kappa)$. To reduce communication complexity, the prover can strategically choose $s(X)$ such that $\sigma_1 = 0$ as the prover doesn't need to send the claimed value of σ_1. Then, to verify the linear relation constraint that z_A, z_B, and z_C are indeed the result of applying matrices A, B, and C to the vector z, the protocol employs a specialized sumcheck protocol based on the techniques from Section 5.3.

The protocol requires a bivariate polynomial $r(X,Y)$ with a critical property: when fixing the second argument to different elements $\kappa \in H$, the resulting univariate polynomials $(r(X,\kappa))_{\kappa \in H}$ must be linearly independent.

The bivariate polynomial $u_H(X,Y) = \frac{v_H(X)-v_H(Y)}{X-Y}$ precisely satisfies this requirement. As noted in the Marlin paper, the univariate polynomials $(u_H(X,\kappa))_{\kappa \in H}$ are linearly independent. Therefore, the protocol can sets $r(X,Y) := u_H(X,Y)$. Defines $r_M(X,Y) = \sum_{\kappa \in H} r(X,\kappa)\hat{M}(\kappa,Y)$ for each matrix $M \in \{A,B,C\}$ and denote $t(X) := \sum_{M \in \{A,B,C\}} \eta_M r_M(\alpha,X)$. The verifier sends random challenges $\alpha, \eta_A, \eta_B, \eta_C \in \mathbb{F}_q$, and the prover needs to prove that

$$q_1(X) := s(X) + r(\alpha,X)\left(\sum_{M \in \{A,B,C\}} \eta_M \hat{z}_M(X)\right) - t(X)\hat{z}(X)$$

sums to 0 over the domain H.

Then, with high probability, the following relationships hold for each matrix M. This probabilistic guarantee forms the foundation of the protocol's soundness:

$$\sum_{\kappa \in H} r(\alpha,\kappa)\hat{z}_M(\kappa) = \sum_{\kappa \in H} r_M(\alpha,\kappa)\hat{z}(\kappa)$$
$$\sum_{\kappa \in H} r(\alpha,\kappa)\hat{z}_M(\kappa) = \sum_{\kappa \in H}\sum_{\iota \in H} r(\alpha,\iota)\hat{M}(\iota,\kappa)\hat{z}(\kappa)$$
$$\Longrightarrow \hat{z}_M(\kappa) = \sum_{\iota \in H} \hat{M}(\kappa,\iota)\hat{z}(\iota)$$

This elegant reduction allows the protocol to employ the univariate sumcheck technique from Section 5.3. Specifically, the prover finds polynomials $g_1(X)$ and $h_1(X)$ such that $q_1(X) = h_1(X)v_H(X) + Xg_1(X)$. Notice that since $\sigma_1 = 0$ was chosen earlier, there's no constant term in this decomposition. The verifier then generates a random challenge β_1 to verify the equation at $X = \beta_1$.

The prover needs to help the verifier compute $t(\beta_1) := \sum_{M \in \{A,B,C\}} \eta_M r_M(\alpha,\beta_1)$, which would otherwise be expensive for the verifier. The Marlin protocol incorporates an optimization that transforms the matrix representation with ideas following from [COS20]. For any matrix $M \in \mathbb{F}_q^{H \times H}$, they define $M^* \in \mathbb{F}_q^{H \times H}$ as $M^*_{a,b} = M_{b,a} \cdot u_H(b,b)$. This transformation allows the key relation $r_M(X,Y) \equiv M^*(Y,X)$ to hold.

Remark 6.1.2 (Matrix transformation optimization)**.** This optimized version of Marlin performs polynomial encoding on the transformed matrices M^* rather than the original matrices M. While the encoding technique from sparse matrix to polynomials $\hat{\text{row}}$, $\hat{\text{col}}$, and $\hat{\text{val}}$ remains unchanged, applying it to M^* instead of M enables more efficient verification. The preprocessing transformation $M \mapsto M^*$ allows the verifier to efficiently compute matrix-related evaluations through the relationship $r_M(X,Y) \equiv M^*(Y,X)$.[2]

Then, the prover defines a new unique polynomial of degree less than $|K|$ such that:

$$\forall \kappa \in K \quad , \quad q_2(\kappa) := \sum_{M \in \{A,B,C\}} \eta_M \frac{v_H(\beta_1)v_H(\alpha)\hat{\mathsf{val}}_{M^*}(\kappa)}{(\beta_1 - \hat{\mathsf{row}}_{M^*}(\kappa))(\alpha - \hat{\mathsf{col}}_{M^*}(\kappa))}$$

This polynomial q_2 essentially encodes the evaluation of the matrix summation at specific

[2]The unoptimized version would encode M directly, requiring more additional complexity in the protocol.

points. To understand why the sum of $q_2(\kappa)$ over K equals $t(\beta_1)$, let's expand the expression step by step:

$$\begin{aligned}
\sum_{\kappa\in K} q_2(\kappa) &= \sum_{\kappa\in K}\sum_{M\in\{A,B,C\}} \eta_M \frac{v_H(\beta_1)v_H(\alpha)\hat{\mathsf{val}}_{M^*}(\kappa)}{(\beta_1-\hat{\mathsf{row}}_{M^*}(\kappa))(\alpha-\hat{\mathsf{col}}_{M^*}(\kappa))} \\
&= \sum_{M\in\{A,B,C\}} \eta_M \sum_{\kappa\in K} \frac{v_H(\beta_1)v_H(\alpha)\hat{\mathsf{val}}_{M^*}(\kappa)}{(\beta_1-\hat{\mathsf{row}}_{M^*}(\kappa))(\alpha-\hat{\mathsf{col}}_{M^*}(\kappa))} \\
&= \sum_{M\in\{A,B,C\}} \eta_M M^*(\beta_1,\alpha) \\
&= \sum_{M\in\{A,B,C\}} \eta_M r_M(\alpha,\beta_1) \\
&= t(\beta_1)
\end{aligned}$$

The prover computes polynomials $g_2(X)$ and $h_2(X)$ such that:

$$q_2(X) = Xg_2(X)+\sigma_2/|K| \quad \text{and} \quad a(X)-b(X)q_2(X) = h_2(X)v_K(X)$$

Where the polynomials $a(X)$ and $b(X)$ are defined as:

$$\begin{aligned}
a(X) &:= \sum_{M\in\{A,B,C\}} \eta_M v_H(\beta_1)v_H(\alpha)\hat{\mathsf{val}}_{M^*}(X) \prod_{N\in\{A,B,C\}\setminus\{M\}} (\beta_1-\hat{\mathsf{row}}_{N^*}(X))(\alpha-\hat{\mathsf{col}}_{N^*}(X)) \\
b(X) &:= \prod_{M\in\{A,B,C\}} (\beta_1-\hat{\mathsf{row}}_{M^*}(X))(\alpha-\hat{\mathsf{col}}_{M^*}(X))
\end{aligned}$$

The first equation demonstrates that f_2 sums to σ_2 over K by Theorem 5.3.2, and the second equation demonstrates that f_2 agrees with the correct evaluations over K. These two equations can be combined into a single equation involving only $g_2(X)$ and $h_2(X)$:

$$a(X)-b(X)(Xg_2(X)+\sigma_2/|K|) = h_2(X)v_K(X)$$

The prover sends just the polynomials $g_2(X)$ and $h_2(X)$. To check this identity, the verifier samples a random element $\beta_2 \in \mathbb{F}_q$ and verifies the equation at $X = \beta_2$.

Through this sophisticated multilevel approach, the protocol efficiently validates the complex relationship between the matrices and witness values:

$$h_2(\beta_2)v_K(\beta_2) \stackrel{?}{=} a(\beta_2) - b(\beta_2)(\beta_2 g_2(\beta_2) + t(\beta_1)/|K|)$$

This check verifies that $f_2(X)$ sums to $t(\beta_1)$ over K, which in turn confirms that $t(\beta_1) = \sum_{M\in\{A,B,C\}} \eta_M r_M(\alpha, \beta_1)$. Once this crucial value has been verified, the protocol can complete the verification of the first sumcheck. This layered approach elegantly reduces the complexity of verifying matrix operations into manageable polynomial checks:

$$\begin{aligned} s(\beta_1) + r(\alpha,\beta_1)\left(\sum_{M\in\{A,B,C\}} \eta_M \hat{z}_M(\beta_1)\right) - t(\beta_1)\hat{z}(\beta_1) \\ \stackrel{?}{=} h_1(\beta_1)v_H(\beta_1) + \beta_1 g_1(\beta_1) \end{aligned}$$

This chain of verifications ensures that the linear relation constraint holds, confirming that z_A, z_B, and z_C are indeed the result of applying matrices A, B, and C to the vector z.

Construction with KZG commitments To transform the interactive polynomial protocol into a non-interactive zk-SNARKs, the construction integrates the KZG polynomial commitment scheme from Chapter 3. This integration is crucial because it allows the prover to commit to polynomials without revealing them, while still enabling the verifier to check polynomial relationships efficiently.

The generator $\mathcal{G}(1^\lambda, D)$ creates a universal SRS that contains KZG commitment parameters $(\mathsf{ck}, \mathsf{rk})$. The commitment key $\mathsf{ck} = (bp, \{x^i G_1\}_{i=0}^{D})$ and verification key $\mathsf{rk} = (bp, xG_2)$ support polynomials of degree at most D, where $D = 6|K| - 6$ is chosen to be at least as large as the maximum degree of any polynomial in the protocol.

The indexer $\mathcal{I}(\mathsf{srs}, \mathbb{i})$ takes the R1CS index $\mathbb{i} = (\mathbb{F}_q, H, K, A, B, C)$ and processes the matrices into polynomial form. For each matrix $M \in \{A, B, C\}$, it computes the polynomials $\hat{\mathsf{row}}_{M^*}$, $\hat{\mathsf{col}}_{M^*}$, and $\hat{\mathsf{val}}_{M^*}$ and creates KZG commitments to these polynomials. The index keys are defined as:

$$\mathsf{ipk} = \left(\mathsf{ck}, \mathbb{i}, [\hat{\mathsf{row}}_{M^*}, \hat{\mathsf{col}}_{M^*}, \hat{\mathsf{val}}_{M^*}]_{M\in\{A,B,C\}}, [c_{\hat{\mathsf{row}}_{M^*}}, c_{\hat{\mathsf{col}}_{M^*}}, c_{\hat{\mathsf{val}}_{M^*}}]_{M\in\{A,B,C\}}\right)$$
$$\mathsf{ivk} = \left(\mathsf{rk}, [c_{\hat{\mathsf{row}}_{M^*}}, c_{\hat{\mathsf{col}}_{M^*}}, c_{\hat{\mathsf{val}}_{M^*}}]_{M\in\{A,B,C\}}\right)$$

The protocol becomes non-interactive using the Fiat-Shamir transform, where verifier challenges are replaced with hash evaluations of the transcript. The prover execution follows the rounds shown in Figure 6.1: encoding the witness, computing derived polynomials, generating quotient polynomials for constraint verification, and producing evaluation proofs.

Linearization and optimization To optimize verification and reduce the communication complexity, Marlin employs linearization polynomials. This technique cleverly reduces the number of field elements in the proof by combining multiple polynomial evaluations into a single linearized polynomial. Without linearization, the prover would need to send individual evaluations of many polynomials at the challenge points. With linearization, several of these evaluations can be ”absorbed” into the polynomial structure itself, significantly reducing the proof size.

The following linearization strategy, while not explicitly detailed in [CHM$^+$20], achieves the stated proof size of 8 field elements in $\mathbb{F}_q$ and 13 group elements in $\mathbb{G}_1$. The highlighted text in the following equations denotes the evaluation parts that need to be transmitted as separate field elements:

$f_1(X) = \hat{z}_A(\beta_1) \cdot \hat{z}_B(X) - \hat{z}_C(X) - h_0(X) \cdot v_H(\beta_1)$ verifies the entry-wise product constraint.

$f_2(X) = s(X) + r(\alpha, \beta_1) \cdot (\eta_A \cdot \hat{z}_A(\beta_1) + \eta_B \cdot \hat{z}_B(X) + \eta_C \cdot \hat{z}_C(X)) - t(\beta_1) \cdot \hat{z}(X) - h_1(X) \cdot v_H(\beta_1) - \beta_1 \cdot g_1(X)$ verifies the first sumcheck relation.

$f_3(X) = h_2(X) \cdot v_K(\beta_2) - \overline{a}(X) + b(\beta_2) \cdot (\beta_2 \cdot g_2(X) + t(\beta_1)/|K|)$ verifies the second sumcheck, where

$$\overline{a}(X) := \sum_{M \in \{A,B,C\}} \eta_M v_H(\beta_1) v_H(\alpha) \hat{\mathsf{val}}_{M^*}(X) \prod_{N \in \{A,B,C\} \setminus \{M\}} (\beta_1 - \hat{\mathsf{row}}_{N^*}(\beta_2))(\alpha - \hat{\mathsf{col}}_{N^*}(\beta_2))$$

The key insight is that the prover only needs to send $\hat{z}_A(\beta_1)$, $t(\beta_1)$, $b(\beta_2)$, and the six evaluations $\{\hat{\mathrm{row}}_{M^*}(\beta_2), \hat{\mathrm{col}}_{M^*}(\beta_2)\}_{M \in \{A,B,C\}}$ as field elements. All other polynomial evaluations such as $v_H(\beta_1)$, $v_K(\beta_2)$, and $r(\alpha, \beta_1)$ can be computed efficiently by the verifier using properties of vanishing polynomials and the bivariate polynomial structure.

The verifier can compute the commitments to these linearization polynomials homomorphically using the KZG commitment properties:

$$\begin{aligned}
c_{f_1} &= \hat{z}_A(\beta_1) \cdot c_{\hat{z}_B} - c_{\hat{z}_C} - v_H(\beta_1) \cdot c_{h_0} \\
c_{f_2} &= c_s + r(\alpha, \beta_1) \cdot (\eta_A \cdot \hat{z}_A(\beta_1) \cdot G_1 + \eta_B \cdot c_{\hat{z}_B} + \eta_C \cdot c_{\hat{z}_C}) \\
&\quad - t(\beta_1) \cdot c_{\hat{z}} - v_H(\beta_1) \cdot c_{h_1} - \beta_1 \cdot c_{g_1} \\
c_{f_3} &= v_K(\beta_2) \cdot c_{h_2} - c_{\overline{a}} + b(\beta_2) \cdot (\beta_2 \cdot c_{g_2} + \frac{t(\beta_1)}{|K|} \cdot G_1)
\end{aligned}$$

where $c_{\overline{a}}$ is computed using the homomorphic properties and the provided evaluations $\hat{\mathrm{row}}_{M^*}(\beta_2)$ and $\hat{\mathrm{col}}_{M^*}(\beta_2)$.

By checking $f_1(\beta_1) = 0$, $f_2(\beta_1) = 0$, and $f_3(\beta_2) = 0$, the verifier ensures the validity of all constraints with high probability. The efficiency of this approach comes from the verifier only needing evaluations at β_1 for $\hat{z}_A$ and t, and at β_2 for $\hat{\mathrm{row}}_{M^*}$ and $\hat{\mathrm{col}}_{M^*}$ where $M \in \{A, B, C\}$, rather than requiring the full polynomials. This linearization strategy transforms verification of complex multilayer sumcheck relations into efficient polynomial commitment verification, achieving both succinctness and practical efficiency.

Summary The complete Marlin protocol achieves a proof size of 8 field elements and 13 group elements in $\mathbb{G}_1$. The field elements consist of: $\hat{z}_A(\beta_1)$, $t(\beta_1)$, and the six evaluations $\{\hat{\mathrm{row}}_{M^*}(\beta_2), \hat{\mathrm{col}}_{M^*}(\beta_2)\}_{M \in \{A,B,C\}}$. The group elements are the commitments: $c_{\hat{w}}$, $c_{\hat{z}_A}$, $c_{\hat{z}_B}$, $c_{\hat{z}_C}$, c_{h_0}, c_s, c_t, c_{g_1}, c_{h_1}, c_{g_2}, c_{h_2}, and the two opening proofs π_1, π_2. The prover performs 11 variable-base multi-scalar multiplications of size m plus $O(m \log m)$ field operations. The verifier requires only 2 pairings and $O(|x| + \log m)$ field operations, achieving the desired succinctness property where verification time is nearly independent of circuit complexity.

```
MarlinP$_1$(ipk, $\mathbb{x}$, $\mathbb{w}$) $\mapsto [c_{\hat{w}}, c_{\hat{z}_A}, c_{\hat{z}_B}, c_{\hat{z}_C}, c_{h_0}, c_s]$ :
    encode $\hat{w}, \hat{z}_A, \hat{z}_B, \hat{z}_C$
    compute $h_0, s$
    $[c_{\hat{w}}, c_{\hat{z}_A}, c_{\hat{z}_B}, c_{\hat{z}_C}, c_{h_0}, c_s] \leftarrow \mathsf{PC.Commit}(ck, [\hat{w}, \hat{z}_A, \hat{z}_B, \hat{z}_C, h_0, s])$
    send $[c_{\hat{w}}, c_{\hat{z}_A}, c_{\hat{z}_B}, c_{\hat{z}_C}, c_{h_0}, c_s]$
MarlinV$_1$(ivk, $\mathbb{x}, c_{\hat{w}}, c_{\hat{z}_A}, c_{\hat{z}_B}, c_{\hat{z}_C}, c_{h_0}, c_s) \mapsto (\alpha, \eta_A, \eta_B, \eta_C)$ :
    send $\alpha \leftarrow\!\$\, \mathbb{F}_q \setminus H$
    send $\eta_A, \eta_B, \eta_C \leftarrow\!\$\, \mathbb{F}_q$
MarlinP$_2(\alpha, \eta_A, \eta_B, \eta_C) \mapsto (c_t, c_{g_1}, c_{h_1})$ :
    compute $t, g_1, h_1$
    $[c_t, c_{g_1}, c_{h_1}] \leftarrow \mathsf{PC.Commit}(ck, [t, g_1, h_1])$
    send $(c_t, c_{g_1}, c_{h_1})$
MarlinV$_2(c_t, c_{g_1}, c_{h_1}) \mapsto \beta_1$ :
    send $\beta_1 \leftarrow\!\$\, \mathbb{F}_q \setminus H$
MarlinP$_3(\beta_1) \mapsto (c_{g_2}, c_{h_2})$ :
    compute $g_2, h_2$
    $[c_{g_2}, c_{h_2}] \leftarrow \mathsf{PC.Commit}(ck, [g_2, h_2])$
    send $(c_{g_2}, c_{h_2})$
MarlinV$_3(c_{g_2}, c_{h_2}) \mapsto \beta_2$ :
    send $\beta_2 \leftarrow\!\$\, \mathbb{F}_q$
MarlinP$_4(\beta_2) \mapsto (\boldsymbol{v_1}, \boldsymbol{v_2})$ :
    $\boldsymbol{p_1}, \boldsymbol{p_2} \leftarrow [\hat{z}_A, t], [\mathsf{poly}_{M^*}]_{\mathsf{poly} \in \{\hat{\mathsf{row}}, \hat{\mathsf{col}}\}, M* \in \{A^*, B^*, C^*\}}$
    $\boldsymbol{v_1}, \boldsymbol{v_2} \leftarrow \boldsymbol{p_1}(\beta_1), \boldsymbol{p_2}(\beta_2)$
    send $(\boldsymbol{v_1}, \boldsymbol{v_2})$
MarlinV$_4(\boldsymbol{v_1}, \boldsymbol{v_2}) \mapsto (\xi_1, \xi_2)$ :
    send $\xi_1, \xi_2 \leftarrow\!\$\, \mathbb{F}_q$
MarlinP$_5(\xi_1, \xi_2) \mapsto (\pi_1, \pi_2)$ :
    compute $f_1, f_2, f_3$
    $\boldsymbol{p_1}, \boldsymbol{p_2} \leftarrow [f_1, f_2] + \boldsymbol{p_1}, [f_3] + \boldsymbol{p_2}$
    $\pi_1, \pi_2 \leftarrow \mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p_1}, \beta_1, \xi_1), \mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p_2}, \beta_2, \xi_2)$
    send $(\pi_1, \pi_2)$
MarlinV$_5(\pi_1, \pi_2) \mapsto 0/1$ :
    compute $c_{f_1}, c_{f_2}, c_{f_3}$
    $\boldsymbol{c_1}, \boldsymbol{c_2} \leftarrow [c_{f_1}, c_{f_2}, c_{\hat{z}_A}, c_t], [c_{f_3}] + [c_{\mathsf{poly}_{M^*}}]_{\mathsf{poly} \in \{\hat{\mathsf{row}}, \hat{\mathsf{col}}\}, M* \in \{A^*, B^*, C^*\}}$
    $\boldsymbol{v_1}, \boldsymbol{v_2} \leftarrow [0, 0] + \boldsymbol{v_1}, [0] + \boldsymbol{v_2}$
    return $\mathsf{PC.BatchCheck}(\mathsf{rk}, [\boldsymbol{c_i}]_{i=1}^2, [\beta_i]_{i=1}^2, [\boldsymbol{v_i}]_{i=1}^2, [\pi_i]_{i=1}^2, [\xi_i]_{i=1}^2])$
```

Figure 6.1: Marlin protocol

6.2 Plonk

Plonk is a universal zk-SNARKs construction with fully succinct verification introduced by Gabizon, Williamson, and Ciobotaru [GWC19]. Building on ideas from Sonic [MBKM19], Plonk achieves significant efficiency improvements through a fundamentally different approach to polynomial representation. Instead of working with polynomial coefficients as in previous protocols, Plonk focuses on polynomial evaluations over a multiplicative subgroup. This design choice enables a more direct arithmetization where circuit gates are represented through evaluations at specific roots of unity, eliminating intermediate polynomial manipulations. Furthermore, the multiplicative subgroup structure naturally aligns with the permutation argument for checking wire constraints, resulting in a simpler and more efficient protocol overall. These optimizations make Plonk particularly attractive for practical implementations while maintaining the same universal and updatable SRS properties.

The Plonk protocol proceeds through a carefully orchestrated sequence of commitments and challenges that build up a complete proof of circuit satisfiability. Figure 6.2 illustrates the complete protocol flow, showing how the prover and verifier exchange information across five rounds. In the first round, the prover commits to the wire polynomials encoding the circuit's execution. The second round introduces the permutation polynomial after receiving randomness for the copy constraint check. The third round commits to the quotient polynomial that combines all constraints. The fourth round provides polynomial evaluations at a random point, and the final round delivers the opening proofs. Each round's challenges depend on previous commitments, creating a sound protocol where cheating would require predicting the verifier's randomness. This interactive structure provides the foundation for understanding how Plonk achieves both efficiency and security before it becomes non-interactive through the Fiat-Shamir transform.

Constraint system The Plonk indexed relation is formally defined as the set of all triples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ where $\mathbb{i} = (\mathbb{F}_q, H, m, n, \mathcal{V}, \mathcal{Q})$ is the index. Here $\mathbb{F}_q$ is a finite field, $H = \langle \mathrm{g} \rangle$ is a multiplicative subgroup of $\mathbb{F}_q$ of size n where g is a primitive n-th root of unity where n is also the number of gates, m is the number of wires, $\mathcal{V} = (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in [m]^n$ represent the left input, right input, and output wire indices for each gate, and $\mathcal{Q} = (\boldsymbol{q_L}, \boldsymbol{q_R}, \boldsymbol{q_O}, \boldsymbol{q_M}, \boldsymbol{q_C}) \in (\mathbb{F}_q^n)^5$ are selector vectors determining the functionality of each gate.

For a triple $(\mathbb{i}, \mathbb{x}, \mathbb{w})$, let $z := (\mathbb{x}, \mathbb{w})$ be the combined assignment in $\mathbb{F}_q^m$, where $\mathbb{x}$ denotes the public input values and $\mathbb{w}$ denotes the witness values. The protocol design assumes without loss of generality that the public inputs correspond to the first ℓ wires, meaning that $\mathbb{x} \in \mathbb{F}_q^\ell$ and $\mathbb{w} \in \mathbb{F}_q^{m-\ell}$. This organization simplifies the handling of public and private values in the circuit. The relation is satisfied if the following two conditions hold:

1. For each $i \in [n]$, the gate constraint is satisfied:

$$(\boldsymbol{q_L})_i \cdot z_{\boldsymbol{a}_i} + (\boldsymbol{q_R})_i \cdot z_{\boldsymbol{b}_i} + (\boldsymbol{q_O})_i \cdot z_{\boldsymbol{c}_i} + (\boldsymbol{q_M})_i \cdot (z_{\boldsymbol{a}_i} \cdot z_{\boldsymbol{b}_i}) + (\boldsymbol{q_C})_i = 0$$

2. The copy constraints are satisfied. The wiring pattern defines a permutation $\sigma : [3n] \to [3n]$ where σ groups positions that reference the same wire. For a valid assignment $z \in \mathbb{F}_q^m$, if we define the value vector $v \in \mathbb{F}_q^{3n}$ as:

$$\begin{aligned} v_i &= z_{\boldsymbol{a}_i} \text{ for } i \in [n] \\ v_{n+i} &= z_{\boldsymbol{b}_i} \text{ for } i \in [n] \\ v_{2n+i} &= z_{\boldsymbol{c}_i} \text{ for } i \in [n] \end{aligned}$$

   Then the copy constraint requires that $v_i = v_{\sigma(i)}$ for all $i \in [3n]$.

The second constraint captures the circuit's wiring pattern. In a circuit, the same wire often connects multiple gates. For example, one gate's output might feed into several other gates inputs. Let $[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{c}] \in [m]^{3n}$ denote the concatenation of wire indices. Whenever positions in $[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{c}]$ contain the same wire index, the actual wire values at those positions must be equal.

To illustrate this concretely: suppose gate 1 outputs to wire 5 (so $\boldsymbol{c}_1 = 5$), gate 3's left input reads from wire 5 (so $\boldsymbol{a}_3 = 5$), and gate 7's right input also reads from wire 5 (so $\boldsymbol{b}_7 = 5$). This means positions $2n+1$, 3, and $n+7$ in the concatenated vector $[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{c}]$ all contain the value 5. The permutation σ creates a cycle connecting these three positions, enforcing that z_5 (the value on wire 5) must be consistent across all three uses.

Each gate in Plonk can function as an addition gate, multiplication gate, or a combination of both by setting different selectors to appropriate values. For example, setting $(\boldsymbol{q_L})_i =$

$(\boldsymbol{q_R})_i = 1$, $(\boldsymbol{q_O})_i = -1$, and $(\boldsymbol{q_M})_i = 0$ creates an addition gate, while setting $(\boldsymbol{q_L})_i = (\boldsymbol{q_R})_i = 0$, $(\boldsymbol{q_O})_i = -1$, and $(\boldsymbol{q_M})_i = 1$ creates a multiplication gate. Plonk also provides a direct mechanism for enforcing constant constraints: to enforce that wire j equals a constant $a \in \mathbb{F}_q$, set for gate i: $\boldsymbol{a}_i = j$, $(\boldsymbol{q_L})_i = 1$, $(\boldsymbol{q_M})_i = (\boldsymbol{q_R})_i = (\boldsymbol{q_O})_i = 0$, and $(\boldsymbol{q_C})_i = -a$, giving the constraint $1 \cdot z_j + 0 + 0 + 0 - a = 0$ which forces $z_j = a$.

Example 6.2.1 (Plonk circuit for polynomial evaluation)**.** Consider the same problem: proving knowledge of $X \in \mathbb{F}_{23}$ such that $Y = X^3 + 2X + 5$ where $Y = 15$ is public and $X = 3$ is the witness.

The circuit uses 9 wires with 1-indexed assignment $z = [1, 2, 5, 15, 3, 9, 4, 6, 10]$ where:

- Wires 1-4: public inputs $[1, 2, 5, Y]$
- Wires 5-9: witness values $[X, X^2, X^3, 2X, X^3 + 2X]$

The 9 gates enforce the computation:

1. $1 \cdot z_1 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot (z_1 \cdot 0) - 1 = 0$ (enforce wire 1 = 1)
2. $1 \cdot z_2 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot (z_2 \cdot 0) - 2 = 0$ (enforce wire 2 = 2)
3. $1 \cdot z_3 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot (z_3 \cdot 0) - 5 = 0$ (enforce wire 3 = 5)
4. $1 \cdot z_4 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot (z_4 \cdot 0) - 15 = 0$ (enforce wire 4 = Y)
5. $0 \cdot z_5 + 0 \cdot z_5 - 1 \cdot z_6 + 1 \cdot (z_5 \cdot z_5) + 0 = 0$ (compute X^2)
6. $0 \cdot z_6 + 0 \cdot z_5 - 1 \cdot z_7 + 1 \cdot (z_6 \cdot z_5) + 0 = 0$ (compute X^3)
7. $0 \cdot z_5 + 0 \cdot z_2 - 1 \cdot z_8 + 1 \cdot (z_5 \cdot z_2) + 0 = 0$ (compute $2X$)
8. $1 \cdot z_7 + 1 \cdot z_8 - 1 \cdot z_9 + 0 \cdot (z_7 \cdot z_8) + 0 = 0$ (compute $X^3 + 2X$)
9. $1 \cdot z_9 + 1 \cdot z_3 - 1 \cdot z_4 + 0 \cdot (z_9 \cdot z_3) + 0 = 0$ (compute final sum)

The selector vectors and wire indices are:

$$\boldsymbol{q_L} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \boldsymbol{q_R} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \boldsymbol{q_O} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}, \boldsymbol{q_M} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \boldsymbol{q_C} = \begin{bmatrix} -1 \\ -2 \\ -5 \\ -15 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\boldsymbol{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 5 \\ 7 \\ 9 \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 5 \\ 5 \\ 2 \\ 8 \\ 3 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 6 \\ 7 \\ 8 \\ 9 \\ 4 \end{bmatrix}$$

The copy constraint permutation $\sigma : [27] \to [27]$ on the concatenated vector $[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{c}]$ creates cycles:

- $(10, 11, 12, 13, 19, 20, 21, 22)$: non-active wires (value 0)
- $(2, 16)$: wire 2 appears at positions 2 and 16
- $(3, 18)$: wire 3 appears at positions 3 and 18
- $(4, 27)$: wire 4 appears at positions 4 and 27
- $(5, 7, 14, 15)$: wire 5 appears at positions 5, 7, 14, and 15
- $(6, 23)$: wire 6 appears at positions 6 and 23
- $(8, 24)$: wire 7 appears at positions 8 and 24
- $(17, 25)$: wire 8 appears at positions 17 and 25
- $(9, 26)$: wire 9 appears at positions 9 and 26

This permutation ensures consistent values across wire reuses.

Polynomial encoding Plonk encodes the circuit and its witness as polynomials evaluated over the multiplicative subgroup $H = \langle \mathrm{g} \rangle$. The prover constructs three wire polynomials $a(X), b(X), c(X) \in \mathbb{F}_q^{<|H|+b}[X]$ such that for each $i \in [n]$:

$$a(\mathrm{g}^i) = z_{\boldsymbol{a}_i}, \quad b(\mathrm{g}^i) = z_{\boldsymbol{b}_i}, \quad c(\mathrm{g}^i) = z_{\boldsymbol{c}_i}$$

The selector polynomials $q_L, q_R, q_O, q_M, q_C \in \mathbb{F}_q^{<|H|+b}[X]$ are fixed for a specific circuit and define the function of each gate. These polynomials are determined during the preprocessing phase and satisfy for each $i \in [n]$:

$$q_L(\mathrm{g}^i) = (\boldsymbol{q_L})_i,\ q_R(\mathrm{g}^i) = (\boldsymbol{q_R})_i,\ q_O(\mathrm{g}^i) = (\boldsymbol{q_O})_i,\ q_M(\mathrm{g}^i) = (\boldsymbol{q_M})_i,\ q_C(\mathrm{g}^i) = (\boldsymbol{q_C})_i$$

For the public input, Plonk uses a special encoding. The protocol assumes the circuit is "prepared" for ℓ public inputs, meaning that for each $i \in [\ell]$:

$$\boldsymbol{a}_i = i, \quad (\boldsymbol{q_L})_i = 1, \quad (\boldsymbol{q_M})_i = (\boldsymbol{q_R})_i = (\boldsymbol{q_O})_i = (\boldsymbol{q_C})_i = 0$$

The public input polynomial is then defined as:

$$\mathsf{PI}(X) = \sum_{i \in [\ell]} -x_i \cdot L_i(X)$$

The encoding of the permutation σ requires mapping the abstract permutation on indices $[3n]$ to concrete field elements that can be interpolated into polynomials. This encoding process involves several steps that connect the circuit's wiring pattern to evaluable polynomials.

First, three disjoint cosets of the multiplicative subgroup H are constructed. Let $k_1, k_2 \in \mathbb{F}_q$ be chosen such that the sets H, $k_1 \cdot H$, and $k_2 \cdot H$ are pairwise disjoint. For example, when g is a quadratic residue in $\mathbb{F}_q$, k_1 can be chosen as any quadratic non-residue, and k_2 as a quadratic non-residue not contained in $k_1 \cdot H$. This gives the extended domain

$H' := H \cup (k_1 \cdot H) \cup (k_2 \cdot H)$ containing $3n$ distinct elements.

Next, a bijection between the abstract index set $[3n]$ and the concrete field elements in H' is established. This mapping connects positions in the concatenated wire vector $[\boldsymbol{a};\boldsymbol{b};\boldsymbol{c}]$ to specific field elements:

$$\begin{aligned}
i &\mapsto \mathbf{g}^i \text{ for } i \in [n] \text{ (left wire positions)} \\
n+i &\mapsto k_1 \cdot \mathbf{g}^i \text{ for } i \in [n] \text{ (right wire positions)} \\
2n+i &\mapsto k_2 \cdot \mathbf{g}^i \text{ for } i \in [n] \text{ (output wire positions)}
\end{aligned}$$

The abstract permutation $\sigma : [3n] \to [3n]$ is then lifted to a mapping $\sigma^* : [3n] \to H'$ by composing σ with the bijection above. For any $j \in [3n]$, $\sigma^*(j)$ gives the field element in H' corresponding to the position that j maps to under the permutation.

Finally, the three permutation polynomials are defined using Lagrange interpolation over the subgroup H:

$$\begin{aligned}
S_{\sigma_1}(X) &:= \sum_{i=1}^{n} \sigma^*(i) L_i(X) \\
S_{\sigma_2}(X) &:= \sum_{i=1}^{n} \sigma^*(n+i) L_i(X) \\
S_{\sigma_3}(X) &:= \sum_{i=1}^{n} \sigma^*(2n+i) L_i(X)
\end{aligned}$$

These polynomials encode where each wire position maps under the permutation: $S_{\sigma_1}(\mathbf{g}^i)$ gives the field element that position i (in the left wire segment) maps to, and similarly for S_{σ_2} and S_{σ_3}.

For efficiency, the identity permutation polynomials are represented directly as degree-1 polynomials: $S_{ID_1}(X) = X$, $S_{ID_2}(X) = k_1 X$, and $S_{ID_3}(X) = k_2 X$. Since these are low-degree polynomials, their evaluations can be computed directly by the verifier without requiring the prover to send them, saving communication cost.

Polynomial identity testing The Plonk protocol verifies the constraint system by combining all constraints into a single quotient polynomial. The prover must prove two key conditions:

1. Gate constraints: For all $\mathbf{g}^i \in H$:

$$q_L(\mathbf{g}^i)\cdot a(\mathbf{g}^i)+q_R(\mathbf{g}^i)\cdot b(\mathbf{g}^i)+q_O(\mathbf{g}^i)\cdot c(\mathbf{g}^i)+q_M(\mathbf{g}^i)\cdot a(\mathbf{g}^i)\cdot b(\mathbf{g}^i)+q_C(\mathbf{g}^i)+\mathsf{PI}(\mathbf{g}^i)=0$$

2. Permutation constraint: $(a,b,c)=\sigma(a,b,c)$, verified through the conditions from Theorem 5.4.2 For all $\mathbf{g}^i \in H$:

$$L_1(\mathbf{g}^i)(Z(\mathbf{g}^i)-1)=0$$
$$Z(\mathbf{g}^i)f'(\mathbf{g}^i)-g'(X)Z(\mathbf{g}^{i+1})=0$$

where $f'(X)$ and $g'(X)$ are the polynomials from the permutation argument:

$$f'(X)=(a(X)+\beta X+\gamma)(b(X)+\beta k_1 X+\gamma)(c(X)+\beta k_2 X+\gamma)$$
$$g'(X)=(a(X)+\beta S_{\sigma_1}(X)+\gamma)(b(X)+\beta S_{\sigma_2}(X)+\gamma)(c(X)+\beta S_{\sigma_3}(X)+\gamma)$$

These constraints are combined using a random challenge $\alpha \in \mathbb{F}_q$. The prover constructs a quotient polynomial $t(X)$ that encodes all constraints:

$$\begin{aligned}t(X)=\frac{1}{v_H(X)}\Big[&q_L(X)a(X)+q_R(X)b(X)+q_O(X)c(X)+q_M(X)a(X)b(X)+q_C(X)+\mathsf{PI}(X)\\&+\alpha\left(Z(X)f'(X)-g'(X)Z(X\mathbf{g})\right)\\&+\alpha^2L_1(X)(Z(X)-1)\Big]\end{aligned}$$

The polynomial $t(X)$ is well-defined (has no remainder when dividing by $v_H(X)$) if and only if all constraints are satisfied on H. Since $t(X)$ can have degree up to $3n+5$ (accounting for blinding factors), the prover splits it into three polynomials

$t'_{\mathsf{lo}}(X), t'_{\mathsf{mid}}(X), t'_{\mathsf{hi}}(X)$ of degree at most $n+5$ such that:

$$t(X) = t'_{\mathsf{lo}}(X) + X^n \cdot t'_{\mathsf{mid}}(X) + X^{2n} \cdot t'_{\mathsf{hi}}(X)$$

To achieve zero-knowledge, the prover adds blinding factors by choosing random scalars $b_{10}, b_{11} \in \mathbb{F}_q$ and defines:

$$\begin{aligned} t_{\mathsf{lo}}(X) &= t'_{\mathsf{lo}}(X) + b_{10}X^n \\ t_{\mathsf{mid}}(X) &= t'_{\mathsf{mid}}(X) - b_{10} + b_{11}X^n \\ t_{\mathsf{hi}}(X) &= t'_{\mathsf{hi}}(X) - b_{11} \end{aligned}$$

Note that the blinding cancels out when reconstructing $t(X) = t_{\mathsf{lo}}(X) + X^n t_{\mathsf{mid}}(X) + X^{2n} t_{\mathsf{hi}}(X)$, maintaining the original polynomial while hiding information about individual components.

The verifier generates a random challenge $\zeta \in \mathbb{F}_q$ and checks that the polynomial identity holds at this point. Specifically, the verifier checks:

$$\begin{aligned} & q_L(\zeta)a(\zeta) + q_R(\zeta)b(\zeta) + q_O(\zeta)c(\zeta) + q_M(\zeta)a(\zeta)b(\zeta) + q_C(\zeta) + \mathsf{PI}(\zeta) \\ & + \alpha\left(Z(\zeta)f'(\zeta) - g'(\zeta)Z(\zeta\mathbf{g})\right) \\ & + \alpha^2 L_1(\zeta)(Z(\zeta) - 1) \\ & \stackrel{?}{=} \left(t_{\mathsf{lo}}(\zeta) + \zeta^n t_{\mathsf{mid}}(\zeta) + \zeta^{2n} t_{\mathsf{hi}}(\zeta)\right) \cdot v_H(\zeta) \end{aligned}$$

where:

$$\begin{aligned} f'(\zeta) &= (a(\zeta) + \beta\zeta + \gamma)(b(\zeta) + \beta k_1\zeta + \gamma)(c(\zeta) + \beta k_2\zeta + \gamma) \\ g'(\zeta) &= (a(\zeta) + \beta S_{\sigma_1}(\zeta) + \gamma)(b(\zeta) + \beta S_{\sigma_2}(\zeta) + \gamma)(c(\zeta) + \beta S_{\sigma_3}(\zeta) + \gamma) \end{aligned}$$

By Lemma 2.3.8, if this identity holds at ζ with high probability over the random choice, then with overwhelming probability all constraints are satisfied on H.

Construction with KZG commitments To transform the interactive polynomial protocol into a non-interactive zk-SNARKs, Plonk integrates the KZG polynomial commitment scheme with the constraint system and permutation argument. This integration achieves a preprocessing zk-SNARKs with universal and updatable SRS, making it highly practical for real-world applications.

The generator $\mathcal{G}(1^\lambda, D)$ creates a universal SRS containing KZG commitment parameters $(\mathsf{ck}, \mathsf{rk})$. The commitment key $\mathsf{ck} = (bp, \{x^i G_1\}_{i=0}^{D})$ and verification key $\mathsf{rk} = (bp, xG_2)$ support polynomials of degree at most D, where $D = n + 5$ is chosen to accommodate the highest-degree polynomial in the protocol (accounting for blinding factors).

The indexer $\mathcal{I}(\mathsf{srs}, \mathbb{i})$ takes the Plonk index $\mathbb{i} = (\mathbb{F}_q, H, m, n, \mathcal{V}, \mathcal{Q})$ and preprocesses the circuit-specific polynomials. It first constructs the selector polynomials $q_L, q_R, q_O, q_M, q_C \in \mathbb{F}_q^{<n}[X]$ from the selector vectors by interpolation over H. Then it constructs the permutation polynomials $S_{\sigma_1}, S_{\sigma_2}, S_{\sigma_3} \in \mathbb{F}_q^{<n}[X]$ that encode the copy constraints. The indexer creates KZG commitments to all these polynomials. The index keys are:

$$\begin{aligned}
\mathsf{ipk} &= \Big(\mathsf{ck}, \mathbb{i}, [q_L, q_R, q_O, q_M, q_C, S_{\sigma_1}, S_{\sigma_2}, S_{\sigma_3}], [c_{q_L}, c_{q_R}, c_{q_O}, c_{q_M}, c_{q_C}, c_{S_{\sigma_1}}, c_{S_{\sigma_2}}, c_{S_{\sigma_3}}]\Big) \\
\mathsf{ivk} &= \Big(\mathsf{rk}, [c_{q_L}, c_{q_R}, c_{q_O}, c_{q_M}, c_{q_C}, c_{S_{\sigma_1}}, c_{S_{\sigma_2}}, c_{S_{\sigma_3}}]\Big)
\end{aligned}$$

The protocol becomes non-interactive through the Fiat-Shamir transform, where verifier challenges are replaced with hash evaluations of the transcript. The prover execution follows the rounds shown in Figure 6.2: encoding the witness as wire polynomials, constructing the permutation polynomial, building the quotient polynomial that combines all constraints, and producing evaluation proofs.

Linearization and optimization To optimize verification, Plonk uses linearization to reduce the number of polynomial evaluations the prover needs to send. This technique cleverly reduces the communication complexity by combining multiple polynomial evaluations into a single linearized polynomial, similar to the approach used in Marlin but adapted for Plonk's constraint structure.

After the verifier chooses a random evaluation point $\zeta \in \mathbb{F}_q$, the prover constructs linearization polynomials that absorb most evaluations into the polynomial structure itself. The highlighted text in the following equations denotes the evaluation parts that would otherwise need to be transmitted as separate field elements:

The linearization polynomial $r(X)$ combines all constraint checks:

$$\begin{aligned} r(X) = {} & a(\zeta)b(\zeta)\cdot q_M(X) + a(\zeta)\cdot q_L(X) + b(\zeta)\cdot q_R(X) + c(\zeta)\cdot q_O(X) + \mathsf{PI}(\zeta) + q_C(X) \\ & + \alpha\left[(a(\zeta)+\beta\zeta+\gamma)(b(\zeta)+\beta k_1\zeta+\gamma)(c(\zeta)+\beta k_2\zeta+\gamma)\cdot z(X)\right. \\ & \quad \left. -(a(\zeta)+\beta S_{\sigma_1}(\zeta)+\gamma)(b(\zeta)+\beta S_{\sigma_2}(\zeta)+\gamma)(c(\zeta)+\beta\cdot S_{\sigma_3}(X)+\gamma)z(\zeta\mathbf{g})\right] \\ & + \alpha^2\left[(z(X)-1)L_1(\zeta)\right] \\ & - v_H(\zeta)\cdot\left(t_{\mathsf{lo}}(X)+\zeta^n t_{\mathsf{mid}}(X)+\zeta^{2n}t_{\mathsf{hi}}(X)\right) \end{aligned}$$

The key insight is that most polynomial evaluations can be absorbed into the linearization polynomial itself rather than being sent as separate field elements. The prover only needs to send the evaluations $a(\zeta)$, $b(\zeta)$, $c(\zeta)$, $S_{\sigma_1}(\zeta)$, $S_{\sigma_2}(\zeta)$, and $z(\zeta\mathbf{g})$ as field elements. All other evaluations such as $q_L(\zeta)$, $q_R(\zeta)$, $q_O(\zeta)$, $q_M(\zeta)$, $q_C(\zeta)$, $L_1(\zeta)$, $v_H(\zeta)$, $\mathsf{PI}(\zeta)$, and $S_{\sigma_3}(\zeta)$ are computed by the verifier from the preprocessed polynomial commitments or direct calculation, since these are either circuit-dependent polynomials known during preprocessing or efficiently computable values.

The verifier can compute the commitment to $r(X)$ homomorphically using the KZG commitment properties:

$$\begin{aligned} c_r = {} & a(\zeta)b(\zeta)\cdot c_{q_M} + a(\zeta)\cdot c_{q_L} + b(\zeta)\cdot c_{q_R} + c(\zeta)\cdot c_{q_O} + \mathsf{PI}(\zeta)\cdot G_1 + c_{q_C} \\ & + \alpha\left[(a(\zeta)+\beta\zeta+\gamma)(b(\zeta)+\beta k_1\zeta+\gamma)(c(\zeta)+\beta k_2\zeta+\gamma)\cdot c_z\right. \\ & \left. -(a(\zeta)+\beta S_{\sigma_1}(\zeta)+\gamma)(b(\zeta)+\beta S_{\sigma_2}(\zeta)+\gamma)\cdot\Big(\beta z(\zeta\mathbf{g})\cdot c_{S_{\sigma_3}} + (c(\zeta)+\gamma)z(\zeta\mathbf{g})\cdot G_1\Big)\right] \\ & + \alpha^2 L_1(\zeta)\cdot c_z - v_H(\zeta)\cdot\left(c_{t_{\mathsf{lo}}} + \zeta^n c_{t_{\mathsf{mid}}} + \zeta^{2n} c_{t_{\mathsf{hi}}}\right) \end{aligned}$$

The linearization technique enables batch verification of all polynomial relations through a single KZG opening proof that verifies $r(\zeta) = 0$. This approach reduces the proof from potentially dozens of field elements to just 6 field elements while maintaining the security guarantees of the protocol.

Summary The complete Plonk protocol achieves a proof size of 6 field elements and 9 group elements in $\mathbb{G}_1$. The field elements consist of the evaluations: $a(\zeta)$, $b(\zeta)$, $c(\zeta)$, $S_{\sigma_1}(\zeta)$, $S_{\sigma_2}(\zeta)$, and $z(\zeta\mathbf{g})$. The group elements are the commitments: c_a, c_b, c_c, c_z, $c_{t_{\mathsf{lo}}}$, $c_{t_{\mathsf{mid}}}$, $c_{t_{\mathsf{hi}}}$, and the two opening proofs π_1, π_2. The prover performs 7 variable-base multi-scalar multiplications of size n plus $O(n \log n)$ field operations. The verifier requires only 2 pairings and $O(\ell + \log n)$ field operations, achieving remarkable efficiency where verification time depends logarithmically on circuit size and linearly only on the number of public inputs.

```
$\text{PlonkP}_1(\mathsf{ipk}, \mathbb{x}, \mathbb{w}) \mapsto (c_a, c_b, c_c)$ :
    encode $a, b, c$
    compute $h_0, s$
    $[c_a, c_b, c_c] \leftarrow \mathsf{PC.Commit}(\mathsf{ck}, [a, b, c])$
    send $(c_a, c_b, c_c)$
$\text{PlonkV}_1(\mathsf{ivk}, \mathbb{x}, c_a, c_b, c_c) \mapsto (\beta, \gamma)$ :
    send $\beta, \gamma \leftarrow\!\!\$\, \mathbb{F}_q$
$\text{PlonkP}_2(\beta, \gamma) \mapsto c_z$ :
    compute $z$
    $[c_z] \leftarrow \mathsf{PC.Commit}(\mathsf{ck}, [t, g_1, h_1])$
    send $c_z$
$\text{PlonkV}_2(c_z) \mapsto \alpha$ :
    send $\alpha \leftarrow\!\!\$\, \mathbb{F}_q$
$\text{PlonkP}_3(\alpha) \mapsto (c_{t_{lo}}, c_{t_{mid}}, c_{t_{hi}})$ :
    compute $t_{lo}, t_{mid}, t_{hi}$
    $[c_{t_{lo}}, c_{t_{mid}}, c_{t_{hi}}] \leftarrow \mathsf{PC.Commit}(\mathsf{ck}, [t_{lo}, t_{mid}, t_{hi}])$
    send $(c_{t_{lo}}, c_{t_{mid}}, c_{t_{hi}})$
$\text{PlonkV}_3(c_{t_{lo}}, c_{t_{mid}}, c_{t_{hi}}) \mapsto \mathfrak{z}$ :
    send $\mathfrak{z} \leftarrow\!\!\$\, \mathbb{F}_q$
$\text{PlonkP}_4(\beta_2) \mapsto (\boldsymbol{v_1}, \boldsymbol{v_2})$ :
    $\boldsymbol{p_1}, \boldsymbol{p_2} \leftarrow [a, b, c, S_{\sigma_1}, S_{\sigma_2}], [z]$
    $\boldsymbol{v_1}, \boldsymbol{v_2} \leftarrow \boldsymbol{p_1}(\mathfrak{z}), \boldsymbol{p_2}(\mathfrak{z}g)$
    send $(\boldsymbol{v_1}, \boldsymbol{v_2})$
$\text{PlonkV}_4(\boldsymbol{v_1}, \boldsymbol{v_2}) \mapsto (\xi_1, \xi_2)$ :
    send $\xi_1, \xi_2 \leftarrow\!\!\$\, \mathbb{F}_q$
$\text{PlonkP}_5(\xi_1, \xi_2) \mapsto (\pi_1, \pi_2)$ :
    compute $r$
    $\boldsymbol{p_1} \leftarrow [r] + \boldsymbol{p_1}$
    $\pi_1, \pi_2 \leftarrow \mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p_1}, \mathfrak{z}, \xi_1), \mathsf{PC.Open}(\mathsf{ck}, \boldsymbol{p_2}, \mathfrak{z}g, \xi_2)$
    send $(\pi_1, \pi_2)$
$\text{PlonkV}_5(\pi_1, \pi_2) \mapsto 0/1$ :
    compute $c_r$
    $\boldsymbol{c_1}, \boldsymbol{c_2} \leftarrow [c_r, c_a, c_b, c_c, c_{S_{\sigma_1}}, c_{S_{\sigma_2}}], [c_z]$
    $\boldsymbol{v_1} \leftarrow [0] + \boldsymbol{v_1}$
    return $\mathsf{PC.BatchCheck}(\mathsf{rk}, [\boldsymbol{c_i}]_{i=1}^{2}, [\beta_i]_{i=1}^{2}, [\boldsymbol{v_i}]_{i=1}^{2}, [\pi_i]_{i=1}^{2}, [\xi_i]_{i=1}^{2})$
```

Figure 6.2: Plonk protocol

6.3 Comparison of Marlin and Plonk

Both Marlin and Plonk utilize the KZG polynomial commitment scheme but achieve different performance characteristics through distinct design choices. Table 6.1 summarizes their key metrics.

Table 6.1: Performance comparison of Marlin and Plonk

Metric	Marlin	Plonk
Constraint system	R1CS	Custom gates
SRS degree	$6m$	n
Proof size		
Field elements ($\mathbb{F}_q$)	8	6
Group elements ($\mathbb{G}_1$)	13	9
Prover complexity		
v-MSM operations	11 of size m	7 of size n
Field operations	$O(m \log m)$	$O(n \log n)$
Verifier complexity		
Pairings	2	2
Field operations	$O(\ell + \log m)$	$O(\ell + \log n)$

Marlin: n = constraints, m = sparse matrix domain. Plonk: n = gates, ℓ = public inputs.

Design trade-offs Marlin's R1CS representation goods at high fan-in addition gates. A single constraint can express $\sum_{i=1}^{k} a_i x_i = b$ regardless of k, while Plonk requires $O(k)$ gates.

Plonk's uniform gate structure simplifies arithmetization and enables direct polynomial encoding. Every gate supports both addition and multiplication, eliminating the need for complex matrix representations.

Practical considerations Plonk's 30% smaller proof size (6+9 vs 8+13 elements) reduces on-chain storage costs in blockchain applications. Its simpler structure typically yields faster proving times for general-purpose circuits. However, for circuits originally designed for R1CS assuming ”free” additions, Marlin may perform better.

Both protocols achieve universal and updatable SRS through KZG commitments. The choice depends primarily on circuit structure: Marlin for addition-heavy computations, Plonk for balanced arithmetic circuits requiring simplicity and smaller proofs.

CHAPTER 7
CONCLUSION

This final project has presented a comprehensive exposition of the KZG polynomial commitment scheme and its fundamental role in zk-SNARKs constructions, specifically examining its implementation in the Marlin and Plonk protocols. Through systematic development of mathematical foundations, detailed proofs, extensive examples, and complete SageMath implementations, this work has aimed to bridge the gap between abstract cryptographic theory and concrete understanding for undergraduate students.

7.1 Conclusion

The key insights and accomplishments of this educational exposition can be summarized as follows:

First, this project has demonstrated that the KZG polynomial commitment scheme serves as a powerful cryptographic primitive that enables the construction of efficient ZKPs. By building from fundamental algebraic structures through to bilinear pairings, the exposition shows how mathematical elegance translates into practical cryptographic protocols. The constant-size commitments and evaluation proofs, regardless of polynomial degree, make KZG particularly suitable for achieving the succinctness property essential to zk-SNARKs.

Second, the detailed examination of both Marlin and Plonk reveals how a single polynomial commitment scheme can enable diverse protocol designs with different performance characteristics. Marlin's use of algebraic holographic proofs and its R1CS arithmetization provides optimal prover efficiency, while Plonk's custom gates and permutation arguments offer a more flexible constraint system. This comparison illustrates the modular nature of modern cryptographic constructions, where foundational primitives like KZG can be composed in various ways to achieve different design goals.

Third, the extensive numerical examples throughout this work serve to demystify operations that often remain abstract in research papers. By computing actual polynomial evaluations over small finite fields, showing explicit pairing calculations, and tracing through complete protocol executions, these examples transform theoretical concepts into tangible computations that students can verify and understand.

Fourth, the SageMath implementations provide a practical bridge between theory and application. These implementations closely follow the theoretical constructions, with extensive documentation explaining each step. Students can modify parameters, experiment with different field sizes, and observe how changes affect performance and security properties. This hands-on approach reinforces theoretical understanding through practical experimentation.

Finally, this project has shown that the apparent complexity of zk-SNARKs protocols stems not from inherently difficult concepts, but from the layering of multiple mathematical and cryptographic ideas. By carefully unpacking each layer, from groups and fields through polynomial commitments to complete protocols, this exposition demonstrates that these sophisticated constructions are accessible to motivated undergraduate students.

The educational value of this work extends beyond the specific protocols examined. By understanding how KZG enables Marlin and Plonk, students gain insight into the broader principles of cryptographic protocol design. They learn how information-theoretic protocols combine with cryptographic commitments to achieve computational security, how polynomial encodings transform computational statements into algebraic relations, and how careful protocol design achieves seemingly contradictory properties like proving knowledge without revealing information.

7.2 Future Directions

Based on the foundation established in this project, several directions for continued study and exploration are suggested:

1. **Exploring ZKP constructions beyond zk-SNARKs**: While this final project focused on zk-SNARKs using KZG commitments, the ZKPs landscape encompasses diverse constructions with different trade-offs. According to a comprehensive survey of ZKP frameworks [SAKK25], four major taxonomies exist: zk-SNARKs, zk-STARKs (Scalable Transparent Arguments of Knowledge), MPC-in-the-Head (MPCitH), and VOLE-based ZK (Vector Oblivious Linear Evaluation). Each approach offers distinct advantages in terms of trusted setup requirements, proof size, post-quantum security,

and computational efficiency. Understanding these alternative constructions would provide students with a comprehensive view of the zero-knowledge landscape and the diverse approaches to achieving privacy-preserving computation.

2. **Alternative polynomial commitment schemes**: Building on the understanding of different ZKP constructions, students could explore polynomial commitment schemes beyond KZG. FRI (Fast Reed-Solomon Interactive Oracle Proofs of Proximity) [BSBHR18], used in zk-STARKs, achieves transparency and post-quantum security through error-correcting codes. Bulletproofs [BBB$^+$18] eliminate trusted setup using inner product arguments, though with logarithmic proof sizes. IPA (Inner Product Arguments) and other schemes offer different trade-offs between setup requirements, proof sizes, and verification complexity. Comparing these alternatives would deepen understanding of how the choice of polynomial commitment scheme fundamentally shapes the properties of the resulting ZKPs.
3. **Implementation optimizations**: The SageMath implementations prioritize clarity over performance and assume constraint systems are provided directly rather than compiled from higher-level programs. Students could implementing the protocols in lower-level languages like Rust or C++ and investigating how production implementations achieve practical performance while maintaining security would bridge the gap between educational code and real-world systems. Second, developing a compiler that transforms programs written in a high-level language into the constraint systems required by Marlin (R1CS) or Plonk (custom gates) would make the implementations more practical. Such a compiler would handle variable assignment, constraint generation, and witness computation automatically, bridging the gap between abstract protocol implementations and usable proof systems. Investigating how production frameworks like Circom [ide22], Leo [Ale21], or Noir [Azt22] achieve both efficient compilation and optimized proving would provide valuable insights into building practical zero-knowledge applications.
4. **Applications and use cases**: Moving beyond the protocols themselves, students could implement concrete applications using the zk-SNARKs constructions studied. Examples include privacy-preserving voting systems, anonymous credentials, or verifiable computation outsourcing. Building actual applications would reinforce understanding of how theoretical properties translate into practical privacy and verifiability guarantees.

5. **Security analysis and attacks**: While this project presented security proofs in idealized models, students interested in cryptanalysis could explore potential attacks, implementation vulnerabilities, or the implications of different security assumptions. Understanding how side-channel attacks might compromise implementations, or how the security of KZG relates to progress in solving discrete logarithm problems, would develop critical security thinking.

The rapidly evolving field of ZKPs offers numerous opportunities for continued learning. The foundation provided by understanding KZG and its role in Marlin and Plonk equips students to engage with ongoing research, contribute to open-source implementations, or apply these techniques in novel contexts. As ZKPs find increasing application in blockchain systems, privacy-preserving technologies, and verifiable computation, the knowledge gained from this exposition provides a solid starting point for future exploration and contribution to this exciting field.

REFERENCES

[ABLZ17] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, and Michał Zajac. A subversion-resistant snark. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 3–33, Cham, 2017. Springer International Publishing.

[Ale21] Aleo Systems. Leo: A programming language for formally verified, zero-knowledge applications. https://developer.aleo.org/leo/, 2021. The Aleo Network.

[Azt22] Aztec Network. Noir: A domain specific language for zero knowledge proofs. https://noir-lang.org/, 2022. Version 0.19.0.

[BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 56–73, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BBB$^+$18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[BCG$^+$22] K. A. Bamberger, R. Canetti, S. Goldwasser, R. Wexler, and E. J. Zimmerman. Verification dilemmas in law and the promise of zero-knowledge proofs. *Berkeley Technology Law Journal*, 37:1, 2022.

[BFS16] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. Nizks with an untrusted crs: Security in the face of parameter subversion. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 777–804, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BLS03] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communication Networks*, pages 257–267, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. Association for Computing Machinery.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Matteo Colombo and Mikhail Chertkov, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[BSCG$^+$14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium*, pages 781–796, 2014.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 499–530, Cham, 2023. Springer Nature Switzerland.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 19–40, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[CHM$^+$20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 738–768, Cham, 2020. Springer International Publishing.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 769–793, Cham, 2020. Springer International Publishing.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 33–62, Cham, 2018. Springer International Publishing.

[FMMO19] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019*, pages 649–678, Cham, 2019. Springer International Publishing.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas

Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GHK+22] G. S. Gaba, M. Hedabou, P. Kumar, A. Braeken, M. Liyanage, and M. Alazab. Zero knowledge proofs based authenticated key agreement protocol for sustainable healthcare. *Sustainable Cities and Society*, 80:103766, 2022.

[GKM+18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 698–728, Cham, 2018. Springer International Publishing.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291304, New York, NY, USA, 1985. Association for Computing Machinery.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[HPS14] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer, 2014.

[ide22] iden3. Circom: Circuit compiler for zero-knowledge proofs. https://docs.circom.io/, 2022. Version 2.1.0.

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 3 edition, December 2020.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. volume 48, pages 203–209, 1987.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Lip24] Helger Lipmaa. Polymath: Groth16 is not the limit. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 457–493, Cham, 2024. Springer Nature Switzerland.

[LLH$^+$22] Chao Lin, Min Luo, Xinyi Huang, Kim-Kwang Raymond Choo, and Debiao He. An efficient privacy-preserving credit score system based on noninteractive zero-knowledge proof. *IEEE Systems Journal*, 16(1):1592–1601, 2022.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 21112128, New York, NY, USA, 2019. Association for Computing Machinery.

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. pages 417–426, 1986.

[MMS96] D. S. Malik, John M. Mordeson, and M. K. Sen. *Fundamentals of Abstract Algebra*. McGraw-Hill College, 1996.

[Nat15a] National Institute of Standards and Technology. FIPS PUB 180-4: Secure Hash Standard (SHS). Technical report, U.S. Department of Commerce, August 2015.

[Nat15b] National Institute of Standards and Technology. FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical report, U.S. Department of Commerce, August 2015.

[RMMY12] Michael O. Rabin, Yishay Mansour, S. Muthukrishnan, and Moti Yung. Strictly-black-box zero-knowledge and efficient validation of financial transactions. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, pages 738–749, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SAKK25] Nojan Sheybani, Anees Ahmed, Michel Kinsy, and Farinaz Koushanfar. Zero-knowledge proof frameworks: A survey. *arXiv preprint arXiv:2502.07063*, 2025.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, pages 256–266, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[TDNHDS20] A. E. B. Tomaz, J. C. Do Nascimento, A. S. Hafid, and J. N. De Souza. Preserving privacy in mobile health systems using non-interactive zero-knowledge proof and blockchain. *IEEE Access*, 8:204441–204458, 2020.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundations and Trends in Privacy and Security*, 4(24):117–660, 2022.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (EUROSAM '79)*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer-Verlag, 1979.

The full code can be seen in the following link: https://github.com/swusjask/kzg-snark

APPENDIX 1: KZG

```python
from sage.all import GF, PolynomialRing

class KZG:

    def __init__(self, curve_type="bn254"):
        # Set up the curve operations based on the specified curve type
        if curve_type == "bn254":
            from py_ecc.optimized_bn128 import (
                G1, G2, multiply, add, curve_order, pairing,
                neg, Z1, Z2, eq
            )
        elif curve_type == "bls12_381":
            from py_ecc.optimized_bls12_381 import (
                G1, G2, multiply, add, curve_order, pairing,
                neg, Z1, Z2, eq
            )
        else:
            raise ValueError(f"Unsupported curve type: {curve_type}")

        # Store curve operations
        self.G1 = G1
        self.G2 = G2
        self.Z1 = Z1  # Zero point in G1
        self.Z2 = Z2  # Zero point in G2
        self.multiply = multiply
        self.add = add
        self.neg = neg
        self.pairing = pairing
        self.eq = eq
        self.curve_order = curve_order

        # Set up the finite field and polynomial ring
        self.Fq = GF(curve_order)
        self.R = PolynomialRing(self.Fq, "X")
        self.X = self.R.gen()

    def setup(self, max_degree):
        # Sample a random secret tau  Fq
        tau = self.Fq.random_element()

        # Generate commitment key: [G, G, G, ..., G]
        powers_of_tau_G1 = [self.G1]
        for i in range(1, max_degree + 1):
            powers_of_tau_G1.append(self.multiply(self.G1, int(tau**i)))

        # Generate verification key: G
        tau_G2 = self.multiply(self.G2, int(tau))

        # Return the key pair
        return (powers_of_tau_G1, tau_G2)

    def commit(self, ck, polynomials):
        # Ensure all inputs are SageMath polynomials
        sage_polynomials = []
        for poly in polynomials:
            if isinstance(poly, list):
                sage_polynomials.append(self.R(poly))
            else:
                sage_polynomials.append(poly)

        max_degree = len(ck) - 1
        commitments = []

        for poly in sage_polynomials:
            if poly.degree() > max_degree:
                raise ValueError(
                    f"exceeds maximum allowed degree {max_degree}"
                )

            # Compute the commitment: p()G =  p(G)
            commitment = self.Z1  # Zero point in G1
            coeffs = poly.list()  # Get coefficients [p, p, ...]

            for i, coeff in enumerate(coeffs):
                if coeff == 0:
                    continue
                term = self.multiply(ck[i], int(coeff))
                commitment = self.add(commitment, term)

            commitments.append(commitment)

        return commitments

    def open(self, ck, polynomials, z, xi):
        # Ensure all inputs are SageMath polynomials
        sage_polynomials = []
        for poly in polynomials:
            if isinstance(poly, list):
                sage_polynomials.append(self.R(poly))
            else:
                sage_polynomials.append(poly)

        # Convert z and challenge to field elements
        z = self.Fq(z)
        xi = self.Fq(xi)

        # Compute the batch polynomial: p(X) =  p(X)
        combined_poly = self.R(0)  # Zero polynomial
```

```python
        for i, poly in enumerate(sage_polynomials):
            combined_poly += xi ** (i + 1) * poly

        # Compute witness polynomial w(X) = (p(X) - p(z))/(X - z)
        X = self.X
        witness_poly = (combined_poly - combined_poly(z)) // (X - z)

        # Commit to the witness polynomial
        proof = self.commit(ck, [witness_poly])[0]

        return proof

    def check(self, rk, commitments, z, evaluations, proof, xi):
        # Extract the verification key component
        tau_G2 = rk

        # Convert z and challenge to field elements
        z = self.Fq(z)
        xi = self.Fq(xi)

        # Compute the batch commitment: C =  C
        combined_commitment = self.Z1
        for i, comm in enumerate(commitments):
            challenge_power = int(xi ** (i + 1))
            term = self.multiply(comm, challenge_power)
            combined_commitment = self.add(combined_commitment, term)

        # Compute the batch evaluation: v =  v
        combined_evaluation = self.Fq(0)
        for i, eval_i in enumerate(evaluations):
            combined_evaluation += xi ** (i + 1) * self.Fq(eval_i)

        # Convert combined evaluation to integer for curve operations
        eval_int = int(combined_evaluation)
        z_int = int(z)

        # Compute C - vG
        v_G1 = self.multiply(self.G1, eval_int)
        C_minus_v = self.add(combined_commitment, self.neg(v_G1))

        # Compute G - zG
        z_G2 = self.multiply(self.G2, z_int)
        tauG2_minus_z = self.add(tau_G2, self.neg(z_G2))

        # Check the pairing equation: e(C - vG, G) = e(, G - zG)
        left_pairing = self.pairing(self.G2, C_minus_v)
        right_pairing = self.pairing(tauG2_minus_z, proof)

        return left_pairing == right_pairing

    def batch_check(self, rk, commitments_list, z_list, evaluations_list, proof_list, xi_list, r=None):
        # Extract the verification key component
        tau_G2 = rk

        # Sample random field element r for batching
        if r is None:
            r = self.Fq.random_element()

        # Initialize accumulators for the batched equation
        left_acc = self.Z1  # Zero point in G1
        right_acc = self.Z1  # Zero point in G1

        # Process each verification instance
        for i, (commitments, z, evaluations, proof, xi) in enumerate(
            zip(commitments_list, z_list, evaluations_list, proof_list, xi_list)
        ):
            # Convert z and challenge to field elements
            z = self.Fq(z)
            xi = self.Fq(xi)

            # Compute the batch commitment and evaluation
            combined_commitment = self.Z1
            combined_evaluation = self.Fq(0)

            for j, comm in enumerate(commitments):
                xi_power = xi ** (j + 1)
                combined_commitment = self.add(
                    combined_commitment, self.multiply(comm, int(xi_power))
                )
                combined_evaluation += xi_power * self.Fq(evaluations[j])

            # Convert to integers for curve operations
            eval_int = int(combined_evaluation)
            z_int = int(z)

            # Transform the verification equation
            # From: e(C - vG, G) = e(, G - zG)
            # To:   e(C - vG + z, G) = e(, G)
            v_G1 = self.multiply(self.G1, eval_int)
            C_minus_v = self.add(combined_commitment, self.neg(v_G1))
            z_pi = self.multiply(proof, z_int)
            term_left = self.add(C_minus_v, z_pi)

            # Apply random power r^i to this verification instance
            r_power = int(r ** (i + 1))  # Use r^(i+1) for security
            term_left = self.multiply(term_left, r_power)
            term_right = self.multiply(proof, r_power)

            # Accumulate terms for the batched equation
            left_acc = self.add(left_acc, term_left)
            right_acc = self.add(right_acc, term_right)

        # Check the batched equation:
        # e((r^i(C_i - v_iG + z_i_i)), G) = e((r^i_i), G)
        left_pairing = self.pairing(self.G2, left_acc)
        right_pairing = self.pairing(tau_G2, right_acc)

        return left_pairing == right_pairing
```

Code 1: KZG polynomial commitment scheme

APPENDIX 2: MARLIN

```python
from sage.all import vector, prod, PolynomialRing, GF

from fft_ff import fft_ff_interpolation

class Encoder:

    def __init__(self, q):
        self.Fq = GF(q)  # Finite field GF(curve_order)
        self.R = PolynomialRing(self.Fq, 'X')
        self.X = self.R.gen()

    def update_state(self, A, B, C):
        self.A = A
        self.B = B
        self.C = C

        # Calculate appropriate subgroup size for domain H (must be power of 2)
        self.n = self.find_subgroup_size(max(A.nrows(), A.ncols()))

        # Calculate appropriate subgroup size for domain K (must be power of 2)
        num_nonzero_A = len(A.nonzero_positions())
        num_nonzero_B = len(B.nonzero_positions())
        num_nonzero_C = len(C.nonzero_positions())
        self.m = self.find_subgroup_size(
            max(num_nonzero_A, num_nonzero_B, num_nonzero_C)
        )

        # Generate multiplicative subgroups H and K
        self.g_H = self.Fq(1).nth_root(self.n)
        self.g_K = self.Fq(1).nth_root(self.m)
        self.H = [self.g_H**i for i in range(self.n)]
        self.K = [self.g_K**i for i in range(self.m)]

        # Compute vanishing polynomials
        self.v_H = self.X**self.n - 1
        self.v_K = self.X**self.m - 1

    def find_subgroup_size(self, n):
        return 2 ** ((n - 1).bit_length())

    def u_H(self, a, b):
        if a == b:
            # When a=b, u_H(a,a) is the formal derivative of v_H at a
            return self.v_H.derivative()(a)
        else:
            return (self.v_H(a) - self.v_H(b)) / (a - b)

    def encode_matrices(self):
        # Precompute u_H values for efficiency
        u_H_diag = {h: self.u_H(h, h) for h in self.H}

        # Dictionary to store all encoded polynomials
        encoded = {}

        # Encode each constraint matrix (A, B, C)
        for name, M in [("A", self.A), ("B", self.B), ("C", self.C)]:
            # Get non-zero positions and values
            nonzero_positions = list(M.nonzero_positions())

            # Prepare arrays for row, column, and values
            row_values = [self.Fq(0)] * self.m
            col_values = [self.Fq(0)] * self.m
            val_values = [self.Fq(0)] * self.m

            # Prepare data for K-interpolation
            for k, (i, j) in enumerate(nonzero_positions):
                row_values[k] = self.H[i]
                col_values[k] = self.H[j]
                # Adjust value by u_H factors as required by Marlin protocol
                val_values[k] = self.Fq(M[i, j]) / (
                    u_H_diag[self.H[i]] * u_H_diag[self.H[j]]
                )

            # Use FFT interpolation to get polynomials
            row_poly = fft_ff_interpolation(row_values, self.g_K, self.Fq)
            col_poly = fft_ff_interpolation(col_values, self.g_K, self.Fq)
            val_poly = fft_ff_interpolation(val_values, self.g_K, self.Fq)

            # Store in dictionary with descriptive keys
            encoded[f"row_{name}"] = row_poly
            encoded[f"col_{name}"] = col_poly
            encoded[f"val_{name}"] = val_poly

        return encoded

    def encode_witness(self, z, x_size):
        # Convert z to field elements
        z = [self.Fq(zi) for zi in z]

        # Split z into x (public input) and w (private witness)
        x, w = z[:x_size], z[x_size:]

        X = self.X

        # Create Lagrange polynomial for x (public input)
        x_points = [(self.H[i], x[i]) for i in range(len(x))]
        x_poly = self.R.lagrange_polynomial(x_points)

        # Calculate vanishing polynomial for x positions
        v_H_x = prod([X - self.H[i] for i in range(len(x))])

        # Encode w using the optimization approach
        # First, create values array with zeros for x positions
        values = [self.Fq(0)] * len(x)

        # Add witness values adjusted by public input polynomial
        for i, wi in enumerate(w):
            values.append(wi - x_poly(self.H[i + len(x)]))

        # Pad values to match H size if needed
        padding_size = self.n - len(values)
        if padding_size > 0:
            values.extend([self.Fq(0)] * padding_size)

        # Interpolate to get polynomial f
        f = fft_ff_interpolation(values, self.g_H, self.Fq)

        # Calculate w_poly = f / v_H_x
        w_poly = f // v_H_x
        assert w_poly * v_H_x == f, "w_poly is not well-defined"

        # Reconstruct z_poly = w_poly * v_H_x + x_poly
        z_poly = w_poly * v_H_x + x_poly

        return {
            "x_poly": x_poly,
            "w_poly": w_poly,
            "z_poly": z_poly,
            "x": x,
            "w": w,
        }

    def encode_linear_combinations(self, z):
        # Convert z to a vector of field elements
        z_vector = vector(self.Fq, [self.Fq(zi) for zi in z])

        # Compute the linear combinations
        zA = self.A * z_vector
        zB = self.B * z_vector
        zC = self.C * z_vector

        # Convert to lists
        zA_list = list(zA)
        zB_list = list(zB)
        zC_list = list(zC)

        # Pad lists to match subgroup size
        if len(zA_list) < self.n:
            zA_list.extend([self.Fq(0)] * (self.n - len(zA_list)))
        if len(zB_list) < self.n:
            zB_list.extend([self.Fq(0)] * (self.n - len(zB_list)))
        if len(zC_list) < self.n:
            zC_list.extend([self.Fq(0)] * (self.n - len(zC_list)))

        # Use FFT interpolation to get polynomials
        zA_poly = fft_ff_interpolation(zA_list, self.g_H, self.Fq)
        zB_poly = fft_ff_interpolation(zB_list, self.g_H, self.Fq)
        zC_poly = fft_ff_interpolation(zC_list, self.g_H, self.Fq)

        return {
            "zA_poly": zA_poly,
            "zB_poly": zB_poly,
            "zC_poly": zC_poly,
            "zA": zA_list,
            "zB": zB_list,
            "zC": zC_list,
        }
```

Code 2: Marlin encoder

```
from kzg import KZG
from marlin.encoder import Encoder

class Indexer:

    def __init__(self, curve_type="bn254"):
        self.kzg = KZG(curve_type=curve_type)
        self.encoder = Encoder(self.kzg.curve_order)

    def preprocess(self, A, B, C, max_degree):
        # Setup KZG commitment scheme
        ck, rk = self.kzg.setup(max_degree)

        # Update encoder state with matrices
        self.encoder.update_state(A, B, C)

        # Create star matrices for more efficient R1CS representation
        A_star, B_star, C_star = A.T, B.T, C.T
        for i in range(A.ncols()):
            A_star[:, i] *= self.encoder.u_H(self.encoder.H[i], self.encoder.H[i])
            B_star[:, i] *= self.encoder.u_H(self.encoder.H[i], self.encoder.H[i])
            C_star[:, i] *= self.encoder.u_H(self.encoder.H[i], self.encoder.H[i])

        self.encoder.update_state(A_star, B_star, C_star)

        # Encode matrices into polynomials
        encoded_matrices = self.encoder.encode_matrices()

        # Organize indexer polynomials in a dictionary
        indexer_polys = {}
        for matrix in ["A", "B", "C"]:
            for poly_type in ["row", "col", "val"]:
                key = f"{poly_type}_{matrix}"
                indexer_polys[key] = encoded_matrices[key]

        # Create a list version for commitment (to be consistent with paper)
        indexer_polys_list = []
        for matrix in ["A", "B", "C"]:
            for poly_type in ["row", "col", "val"]:
                key = f"{poly_type}_{matrix}"
                indexer_polys_list.append(encoded_matrices[key])

        # Commit to the indexer polynomials
        index_commitments = self.kzg.commit(ck, indexer_polys_list)

        # Organize commitments in a dictionary
        commitments = {}
        i = 0
        for matrix in ["A", "B", "C"]:
            for poly_type in ["row", "col", "val"]:
                key = f"{poly_type}_{matrix}"
                commitments[key] = index_commitments[i]
                i += 1

        # Create index proving key
        ipk = {
            "ck": ck,
            "A": A,
            "B": B,
            "C": C,
            "polynomials": indexer_polys,
            "commitments": commitments,
            # Additional data needed by the prover
            "subgroups": {
                "H": self.encoder.H,
                "K": self.encoder.K,
                "g_H": self.encoder.g_H,
                "g_K": self.encoder.g_K,
                "n": self.encoder.n,
                "m": self.encoder.m
            },
            "vanishing_polys": {
                "v_H": self.encoder.v_H,
                "v_K": self.encoder.v_K,
            }
        }

        # Create index verification key - only contains what verifier needs
        ivk = {
            "rk": rk,
            "commitments": commitments,
            "subgroups": {
                "n": self.encoder.n,
                "m": self.encoder.m,
                "g_H": self.encoder.g_H,
            },
            "vanishing_polys": {
                "v_H": self.encoder.v_H,
                "v_K": self.encoder.v_K,
            }
        }

        return ipk, ivk
```

Code 3: Marlin indexer

```
from sage.all import prod

from kzg import KZG
from fft_ff import fft_ff, fft_ff_interpolation
from transcript import Transcript
from marlin.encoder import Encoder

class Prover:

    def __init__(self, curve_type="bn254"):
        self.kzg = KZG(curve_type=curve_type)

    def prove(self, ipk, x, w, zero_knowledge_bound=2):
        # Extract data from index proving key
        ck = ipk["ck"]
        A, B, C = ipk["A"], ipk["B"], ipk["C"]
        polynomials = ipk["polynomials"]
        H, K = ipk["subgroups"]["H"], ipk["subgroups"]["K"]
        n, m = ipk["subgroups"]["n"], ipk["subgroups"]["m"]
        g_K = ipk["subgroups"]["g_K"]
        v_H, v_K = ipk["vanishing_polys"]["v_H"], ipk["vanishing_polys"]["v_K"]
        R = self.kzg.R
        X = self.kzg.X
        Fq = self.kzg.Fq

        # Create an encoder with the same field
        self.encoder = Encoder(self.kzg.curve_order)
        self.encoder.update_state(A, B, C)

        # Create a transcript for the Fiat-Shamir transform
        transcript = Transcript("marlin-proof", Fq)
        transcript.append_message("public-inputs", x)

        # Phase 1: Encode witness and linear combinations
        z = list(x) + list(w)  # Full variable assignment
        x_size = len(x)

        # Compute vanishing polynomials for x and w
        v_H_x = prod([X - h for h in H[:x_size]])
        v_H_w = prod([X - h for h in H[x_size:]])

        # Encode witness
        encoded_witness = self.encoder.encode_witness(z, x_size)

        # Encode linear combinations
        encoded_combinations = self.encoder.encode_linear_combinations(z)

        # Extract polynomials
        w_poly = encoded_witness["w_poly"]
        x_poly = encoded_witness["x_poly"]
        zA_poly = encoded_combinations["zA_poly"]
        zB_poly = encoded_combinations["zB_poly"]
        zC_poly = encoded_combinations["zC_poly"]

        # Add randomness for zero-knowledge (bounded independence)
        b = zero_knowledge_bound

        # Random polynomials of degree < b
        w_random = sum(Fq.random_element() * X**i for i in range(b))
        zA_random = sum(Fq.random_element() * X**i for i in range(b))
        zB_random = sum(Fq.random_element() * X**i for i in range(b))
        zC_random = sum(Fq.random_element() * X**i for i in range(b))

        # Mask the polynomials with randomness
        w_masked = w_poly + w_random * v_H_w
        zA_masked = zA_poly + zA_random * v_H
        zB_masked = zB_poly + zB_random * v_H
        zC_masked = zC_poly + zC_random * v_H
        z_masked = w_masked * v_H_x + x_poly

        # Compute h_0 for the first check: zAzB - zC = h_0v_H
        h_0 = (zA_masked * zB_masked - zC_masked) // v_H
        assert h_0 * v_H == zA_masked * zB_masked - zC_masked, "h_0 polynomial is not well-defined"

        # Generate random polynomial s(X) such that sum(s(H)) = 0
        s_random = sum(Fq.random_element() * X**i for i in range(2*n+b-1))
        s_sum = sum(s_random(h) for h in H)
        s = s_random - s_sum/len(H)  # Adjust to ensure sum over H is zero

        # First round commitments
        first_round_polys = [w_masked, zA_masked, zB_masked, zC_masked, h_0, s]
        first_round_commitments = self.kzg.commit(ck, first_round_polys)

        # Add commitments to transcript
        transcript.append_message("round1-commitments", first_round_commitments)

        # Get first round challenges
        eta_A = transcript.get_challenge("eta_A")
        eta_B = transcript.get_challenge("eta_B")
        eta_C = transcript.get_challenge("eta_C")
        alpha = transcript.get_challenge("alpha")

        # Ensure alpha is not in H (as required by protocol)
        while alpha in H:
            alpha = transcript.get_challenge("alpha-retry")

        # Compute t(X) - the combined polynomial for _M *  r_M(, X)
        t = self._compute_t_polynomial(
            polynomials, eta_A, eta_B, eta_C, alpha, v_H, K, R
        )

        # Compute first sumcheck polynomial
        r = lambda a, b: self.encoder.u_H(a, b)  # Helper function for u_H

        # Compute the sum of s(X) + r(, X) * ( _M * z_M(X)) - t(X) * z(X) over H
        poly = R(s + r(alpha, X)*(eta_A * zA_masked + eta_B * zB_masked + eta_C * zC_masked) - t * z_masked)

        # Divide by vanishing polynomial v_H to get h_1 and g_1
        h_1 = poly // v_H
        g_1 = poly % v_H

        assert g_1.constant_coefficient() == 0, "Sum over H is not 0"
        g_1 = g_1 // X
        assert h_1*v_H + X * g_1 == poly, "h_1 and g_1 are not well-defined"

        # Second round commitments
        second_round_polys = [t, g_1, h_1]
        second_round_commitments = self.kzg.commit(ck, second_round_polys)

        transcript.append_message("round2-commitments", second_round_commitments)

        # Get second round challenge
        beta_1 = transcript.get_challenge("beta_1")

        # Ensure beta_1 is not in H
        while beta_1 in H:
            beta_1 = transcript.get_challenge("beta_1-retry")

        # Calculate a(X) and b(X) polynomials for the third sumcheck
        a, b = self._compute_a_b_polynomials(
            polynomials, eta_A, eta_B, eta_C, beta_1, alpha, v_H, R
        )

        # Calculate sumcheck for the value of t()
        t_beta1 = t(beta_1)

        # Calculate f_2, g_2, and h_2 polynomials for the third sumcheck
        f_2 = self._compute_f2_polynomial(
            polynomials, eta_A, eta_B, eta_C, beta_1, alpha, v_H, v_K, K, g_K, Fq, R
        )

        # Verify f_2(0) = t()/m
        assert f_2.constant_coefficient() == t_beta1 / m, "f_2 polynomial is incorrect"

        # Compute g_2 and h_2
        g_2 = f_2 // X
        h_2 = (a - b*f_2) // v_K
        assert h_2 * v_K == a - b * (X * g_2 + t_beta1 / m), "h_2 and g_2 are not well-defined"

        # Third round commitments
        third_round_polys = [g_2, h_2]
        third_round_commitments = self.kzg.commit(ck, third_round_polys)

        transcript.append_message("round3-commitments", third_round_commitments)

        # Get third round challenge
        beta_2 = transcript.get_challenge("beta_2")

        # f1(x) = zA()*zB(x) - zC(x) - h_0(x)*v_H()
        f_1 = zA_masked(beta_1) * zB_masked - zC_masked - h_0 * v_H(beta_1)

        # f2(x) = s(x) + r(, )*(_A*zA() + _B*zB(x) + _C*zC(x)) - t()*z(x) - h_1(x)*v_H() - g_1(x)
        z = w_masked * v_H_x(beta_1) + x_poly(beta_1)
        f_2 = s + r(alpha, beta_1) * (eta_A * zA_masked(beta_1) + eta_B * zB_masked + eta_C * zC_masked) - t_beta1 * z - h_1 *

        # f3(x) = h_2(x)*v_K() - a(x) + b()*(g_2(x) + t()/m)
        # a(x) =  _M * v_H() * v_H() * val_M(x) _{NM} ( - row_N())( - col_N())

        a_lin, b_lin = self._compute_a_b_linear_polynomials(
            polynomials, eta_A, eta_B, eta_C, beta_1, beta_2, alpha, v_H, R, Fq
        )

        f_3 = h_2 * v_K(beta_2) - a_lin + b_lin * (beta_2 * g_2 + t_beta1 / m)

        assert f_1(beta_1) == 0, "f_1 polynomial is not well-defined"
        assert f_2(beta_1) == 0, "f_2 polynomial is not well-defined"
        assert f_3(beta_2) == 0, "f_3 polynomial is not well-defined"

        # Evaluate polynomials at the challenge points
        polys_beta1 = [zA_masked, t]
        evals_beta1 = [p(beta_1) for p in polys_beta1]

        # Polynomials for beta_2 evaluation
        polys_beta2 = []
        for matrix in ["A", "B", "C"]:
            for poly_type in ["row", "col"]:
                key = f"{poly_type}_{matrix}"
                polys_beta2.append(polynomials[key])

        evals_beta2 = [p(beta_2) for p in polys_beta2]

        # Add evaluations to transcript
        transcript.append_message("evaluations-beta1", evals_beta1)
        transcript.append_message("evaluations-beta2", evals_beta2)

        xi_1 = transcript.get_challenge("xi_1")
        xi_2 = transcript.get_challenge("xi_2")

        # Generate KZG batch proofs
        polys_beta1 = [f_1, f_2] + polys_beta1
        polys_beta2 = [f_3] + polys_beta2
        proof_beta1 = self.kzg.open(ck, polys_beta1, beta_1, xi_1)
        proof_beta2 = self.kzg.open(ck, polys_beta2, beta_2, xi_2)

        # Assemble the final proof
        proof = {
            "commitments": {
                "first_round": first_round_commitments,
                "second_round": second_round_commitments,
                "third_round": third_round_commitments
            },
            "evaluations": {
                "beta1": evals_beta1,
                "beta2": evals_beta2
            },
            "kzg_proofs": {
                "beta1": proof_beta1,
                "beta2": proof_beta2
            }
        }

        return proof

    def _compute_t_polynomial(self, polynomials, eta_A, eta_B, eta_C, alpha, v_H, K, R):
        # Extract row, col, val polynomials for each matrix
        row_A = polynomials["row_A"]
        col_A = polynomials["col_A"]
        val_A = polynomials["val_A"]

        row_B = polynomials["row_B"]
        col_B = polynomials["col_B"]
        val_B = polynomials["val_B"]

        row_C = polynomials["row_C"]
        col_C = polynomials["col_C"]
        val_C = polynomials["val_C"]

        X = R.gen()
        t_poly = R(0)

        # Compute t(X) =  _M  _K [v_H(X)v_H()val_M*() / ((X - row_M*())( - col_M*()))]
        for kappa in K:
            # Term for matrix A
            denom_A = (X - row_A(kappa)) * (alpha - col_A(kappa))
            if denom_A != 0:
                term_A = (v_H(X) * v_H(alpha) * val_A(kappa)) / denom_A
                t_poly += eta_A * term_A

            # Term for matrix B
            denom_B = (X - row_B(kappa)) * (alpha - col_B(kappa))
            if denom_B != 0:
                term_B = (v_H(X) * v_H(alpha) * val_B(kappa)) / denom_B
                t_poly += eta_B * term_B

            # Term for matrix C
            denom_C = (X - row_C(kappa)) * (alpha - col_C(kappa))
            if denom_C != 0:
                term_C = (v_H(X) * v_H(alpha) * val_C(kappa)) / denom_C
                t_poly += eta_C * term_C

        return R(t_poly)

    def _compute_a_b_polynomials(self, polynomials, eta_A, eta_B, eta_C, beta_1, alpha, v_H, R):
        # Extract row, col, val polynomials for each matrix
        row_A = polynomials["row_A"]
        col_A = polynomials["col_A"]
        val_A = polynomials["val_A"]

        row_B = polynomials["row_B"]
        col_B = polynomials["col_B"]
        val_B = polynomials["val_B"]

        row_C = polynomials["row_C"]
        col_C = polynomials["col_C"]
        val_C = polynomials["val_C"]

        a = R(0)
        b = R(1)  # Start with 1 for the product

        # Process each matrix
        for matrix_idx, (eta, row, col, val) in enumerate([
            (eta_A, row_A, col_A, val_A),
            (eta_B, row_B, col_B, val_B),
            (eta_C, row_C, col_C, val_C)
        ]):
            # Calculate the product term for other matrices
            other_product = R(1)
            for other_idx, (other_row, other_col) in enumerate([
                (row_A, col_A), (row_B, col_B), (row_C, col_C)
            ]):
                if other_idx != matrix_idx:
                    other_product *= (beta_1 - other_row) * (alpha - other_col)

            # Add term to a(X)
            a += eta * v_H(beta_1) * v_H(alpha) * val * other_product

            # Update b(X) with this matrix's factors
            b *= (beta_1 - row) * (alpha - col)

        return a, b

    def _compute_a_b_linear_polynomials(self, polynomials, eta_A, eta_B, eta_C, beta_1, beta_2, alpha, v_H, R, Fq):
        row_A = polynomials["row_A"]
        col_A = polynomials["col_A"]
        val_A = polynomials["val_A"]

        row_B = polynomials["row_B"]
        col_B = polynomials["col_B"]
        val_B = polynomials["val_B"]

        row_C = polynomials["row_C"]
        col_C = polynomials["col_C"]
        val_C = polynomials["val_C"]
        a = R(0)
        b = Fq(1)

        for matrix_idx, (eta, row, col, val) in enumerate([
            (eta_A, row_A, col_A, val_A),
            (eta_B, row_B, col_B, val_B),
            (eta_C, row_C, col_C, val_C)
        ]):
            other_product = Fq(1)
            for other_idx, (other_row, other_col) in enumerate([
                (row_A, col_A), (row_B, col_B), (row_C, col_C)
            ]):
                if other_idx != matrix_idx:
                    other_product *= (beta_1 - other_row(beta_2)) * (alpha - other_col(beta_2))

            a += eta * v_H(beta_1) * v_H(alpha) * val * other_product
            b *= (beta_1 - row(beta_2)) * (alpha - col(beta_2))

        return a, b

    def _compute_f2_polynomial(self, polynomials, eta_A, eta_B, eta_C, beta_1, alpha, v_H, v_K, K, g_K, Fq, R):
```

```python
        # Extract row, col, val polynomials for each matrix
        row_A = polynomials["row_A"]
        col_A = polynomials["col_A"]
        val_A = polynomials["val_A"]

        row_B = polynomials["row_B"]
        col_B = polynomials["col_B"]
        val_B = polynomials["val_B"]

        row_C = polynomials["row_C"]
        col_C = polynomials["col_C"]
        val_C = polynomials["val_C"]

        # Pre-compute v_H values
        v_H_beta1 = v_H(beta_1)
        v_H_alpha = v_H(alpha)

        # Pre-compute evaluations at points in K for efficiency
        row_A_evals = fft_ff(list(row_A), g_K, Fq)
        col_A_evals = fft_ff(list(col_A), g_K, Fq)
        val_A_evals = fft_ff(list(val_A), g_K, Fq)

        row_B_evals = fft_ff(list(row_B), g_K, Fq)
        col_B_evals = fft_ff(list(col_B), g_K, Fq)
        val_B_evals = fft_ff(list(val_B), g_K, Fq)

        row_C_evals = fft_ff(list(row_C), g_K, Fq)
        col_C_evals = fft_ff(list(col_C), g_K, Fq)
        val_C_evals = fft_ff(list(val_C), g_K, Fq)

        # Compute f2() for each   K
        f2_evals = []

        for i in range(len(K)):
            # Calculate denominators for each matrix
            denom_A = (beta_1 - row_A_evals[i]) * (alpha - col_A_evals[i])
            denom_B = (beta_1 - row_B_evals[i]) * (alpha - col_B_evals[i])
            denom_C = (beta_1 - row_C_evals[i]) * (alpha - col_C_evals[i])

            # Calculate individual terms with safeguards for division by zero
            term_A = v_H_beta1 * v_H_alpha * val_A_evals[i] / denom_A if denom_A != 0 else 0
            term_B = v_H_beta1 * v_H_alpha * val_B_evals[i] / denom_B if denom_B != 0 else 0
            term_C = v_H_beta1 * v_H_alpha * val_C_evals[i] / denom_C if denom_C != 0 else 0

            # Combine terms with challenge values
            f2_evals.append(eta_A * term_A + eta_B * term_B + eta_C * term_C)

        # Interpolate to get the polynomial
        f_2 = fft_ff_interpolation(f2_evals, g_K, Fq)

        return f_2
```

Code 4: Marlin prover

```
from sage.all import prod

from kzg import KZG
from transcript import Transcript

class Verifier:

    def __init__(self, curve_type="bn254"):
        # Initialize the KZG polynomial commitment scheme with specified curve
        self.kzg = KZG(curve_type=curve_type)

    def verify(self, ivk, x, proof):
        # Extract data from verification key and proof
        rk = ivk["rk"]
        index_commitments = ivk["commitments"]
        n, m = ivk["subgroups"]["n"], ivk["subgroups"]["m"]
        g_H = ivk["subgroups"]["g_H"]
        v_H, v_K = ivk["vanishing_polys"]["v_H"], ivk["vanishing_polys"]["v_K"]

        # Extract proof components
        first_round_commitments = proof["commitments"]["first_round"]
        second_round_commitments = proof["commitments"]["second_round"]
        third_round_commitments = proof["commitments"]["third_round"]
        evals_beta1 = proof["evaluations"]["beta1"]
        evals_beta2 = proof["evaluations"]["beta2"]
        kzg_proof_beta1 = proof["kzg_proofs"]["beta1"]
        kzg_proof_beta2 = proof["kzg_proofs"]["beta2"]

        # Create a transcript for the Fiat-Shamir transform
        R = self.kzg.R
        Fq = self.kzg.Fq
        transcript = Transcript("marlin-proof", Fq)
        transcript.append_message("public-inputs", x)

        # Recreate the transcript to generate the same challenges as the prover
        transcript.append_message("round1-commitments", first_round_commitments)

        # Get first round challenges
        eta_A = transcript.get_challenge("eta_A")
        eta_B = transcript.get_challenge("eta_B")
        eta_C = transcript.get_challenge("eta_C")
        alpha = transcript.get_challenge("alpha")

        # Add second round commitments and get challenge
        transcript.append_message("round2-commitments", second_round_commitments)
        beta_1 = transcript.get_challenge("beta_1")

        # Add third round commitments and get challenge
        transcript.append_message("round3-commitments", third_round_commitments)
        beta_2 = transcript.get_challenge("beta_2")

        # Add evaluations to transcript
        transcript.append_message("evaluations-beta1", evals_beta1)
        transcript.append_message("evaluations-beta2", evals_beta2)

        # Get opening challenges for batch verification
        xi_1 = transcript.get_challenge("xi_1")
        xi_2 = transcript.get_challenge("xi_2")

        # Extract evaluations from the proof
        [zA_beta1, t_beta1] = evals_beta1

        # Get commitments
        [w_comm, zA_comm, zB_comm, zC_comm, h0_comm, s_comm] = first_round_commitments
        [t_comm, g1_comm, h1_comm] = second_round_commitments
        [g2_comm, h2_comm] = third_round_commitments

        # Compute linearization polynomial f1, f2, f3 commitments

        # f1(x) = zA()*zB(x) - zC(x) - h_0(x)*v_H()
        f1_comm = self.kzg.multiply(zB_comm, int(zA_beta1))
        f1_comm = self.kzg.add(f1_comm, self.kzg.neg(zC_comm))
        f1_comm = self.kzg.add(f1_comm, self.kzg.multiply(h0_comm, int(-v_H(beta_1))))

        # f2(x) = s(x) + r(, )*(_A*zA() + _B*zB(x) + _C*zC(x)) - t()*z(x) - h_1(x)*v_H() - g_1(x)
        H_x = [g_H**i for i in range(len(x))]
        v_H_x_beta1 = prod([(beta_1 - H_x[i]) for i in range(len(x))])
        x_points = [(H_x[i], x[i]) for i in range(len(x))]
        x_poly = R.lagrange_polynomial(x_points)
        x_beta1 = x_poly(beta_1)

        z_comm = self.kzg.multiply(w_comm, int(v_H_x_beta1))
        z_comm = self.kzg.add(z_comm, self.kzg.multiply(self.kzg.G1, int(x_beta1)))

        r_alpha_beta1 = (alpha**n - beta_1**n) / (alpha - beta_1)

        f2_comm = s_comm
        temp = self.kzg.multiply(self.kzg.G1, int(eta_A * zA_beta1))
        temp = self.kzg.add(temp, self.kzg.multiply(zB_comm, int(eta_B)))
        temp = self.kzg.add(temp, self.kzg.multiply(zC_comm, int(eta_C)))
        temp = self.kzg.multiply(temp, int(r_alpha_beta1))
        f2_comm = self.kzg.add(f2_comm, temp)
        f2_comm = self.kzg.add(f2_comm, self.kzg.multiply(z_comm, int(-t_beta1)))
        f2_comm = self.kzg.add(f2_comm, self.kzg.multiply(h1_comm, int(-v_H(beta_1))))
        f2_comm = self.kzg.add(f2_comm, self.kzg.multiply(g1_comm, int(-beta_1)))

        # f3(x) = h_2(x)*v_K() - a(x) + b()*(g_2(x) + t()/m)

        a_comm, b_lin = self._compute_a_b_linear(index_commitments, evals_beta2, beta_1, alpha, eta_A, eta_B, eta_C, v_H, Fq)
        f3_comm = self.kzg.multiply(h2_comm, int(v_K(beta_2)))
        f3_comm = self.kzg.add(f3_comm, self.kzg.neg(a_comm))
        temp = self.kzg.multiply(g2_comm, int(beta_2))
        temp = self.kzg.add(temp, self.kzg.multiply(self.kzg.G1, int(t_beta1 / m)))
        temp = self.kzg.multiply(temp, int(b_lin))
        f3_comm = self.kzg.add(f3_comm, temp)

        # batch verify
        beta1_commitments = [f1_comm, f2_comm, zA_comm, t_comm]

        beta2_commitments = [f3_comm]

        for matrix in ["A", "B", "C"]:
            for poly_type in ["row", "col"]:
                key = f"{poly_type}_{matrix}"
                beta2_commitments.append(index_commitments[key])

        beta1_evaluations = [0] * 2 + evals_beta1
        beta2_evaluations = [0] + evals_beta2

        commitment_list = [beta1_commitments, beta2_commitments]
        z_values = [beta_1, beta_2]
        evaluation_list = [beta1_evaluations, beta2_evaluations]
        proof_list = [kzg_proof_beta1, kzg_proof_beta2]
        xi_list = [xi_1, xi_2]
        batch_result = self.kzg.batch_check(rk, commitment_list, z_values, evaluation_list, proof_list, xi_list)

        return batch_result

    def _compute_a_b_linear(self, index_commitments, evals_beta2, beta_1, alpha, eta_A, eta_B, eta_C, v_H, Fq):
        val_A_comm, val_B_comm, val_C_comm = (
            index_commitments["val_A"],
            index_commitments["val_B"],
            index_commitments["val_C"],
        )

        [row_A_beta2, col_A_beta2,
         row_B_beta2, col_B_beta2,
         row_C_beta2, col_C_beta2] = evals_beta2

        a = self.kzg.multiply(self.kzg.G1, int(0))
        b = Fq(1)

        for matrix_idx, (eta, row, col, val) in enumerate([
            (eta_A, row_A_beta2, col_A_beta2, val_A_comm),
            (eta_B, row_B_beta2, col_B_beta2, val_B_comm),
            (eta_C, row_C_beta2, col_C_beta2, val_C_comm)
        ]):
            other_product = Fq(1)
            for other_idx, (other_row, other_col) in enumerate([
                (row_A_beta2, col_A_beta2),
                (row_B_beta2, col_B_beta2),
                (row_C_beta2, col_C_beta2)
            ]):
                if other_idx != matrix_idx:
                    other_product *= (beta_1 - other_row) * (alpha - other_col)


            a = self.kzg.add(a, self.kzg.multiply(val, int(eta * v_H(beta_1) * v_H(alpha) * other_product)))
            b *= (beta_1 - row) * (alpha - col)

        return a, b
```

Code 5: Marlin verifier

APPENDIX 3: PLONK

```
from sage.all import PolynomialRing, GF

from fft_ff import fft_ff_interpolation

class Encoder:

    def __init__(self, q):
        self.Fq = GF(q)  # Finite field GF(q)
        self.R = PolynomialRing(self.Fq, 'X')
        self.X = self.R.gen()

    def find_subgroup_size(self, n):
        return 2 ** ((n - 1).bit_length())

    def update_state(self, qM, qL, qR, qO, qC, perm):
        # Calculate appropriate subgroup size (must be power of 2)
        self.n = self.find_subgroup_size(len(qM))

        # Generate subgroup H with generator g
        self.g = self.Fq(1).nth_root(self.n)

        # Store circuit data
        self.qM = qM
        self.qL = qL
        self.qR = qR
        self.qO = qO
        self.qC = qC
        self.perm = perm

        # Generate subgroup H (multiplicative subgroup of order n)
        self.H = [self.g**i for i in range(self.n)]

        # Find suitable multipliers for creating cosets (k1H and k2H)
        self._find_coset_multipliers()

        # Generate cosets
        self.k1H = [self.k1 * h for h in self.H]
        self.k2H = [self.k2 * h for h in self.H]

        # Compute vanishing polynomial for H: Z_H(X) = X^n - 1
        self.v_H = self.X**self.n - 1

    def _find_coset_multipliers(self):
        n = self.n

        # Keep trying random values until we find suitable ones
        while True:
            k1 = self.Fq.random_element()
            k2 = self.Fq.random_element()

            # Check that k1 and k2 satisfy the required conditions:
            # 1. k1ˆn  1 (k1 is not in H)
            # 2. k2ˆn  1 (k2 is not in H)
            # 3. (k1/k2)ˆn  1 (k1H and k2H are disjoint)
            # 4. k1, k2  0 (non-zero)
            if (k1**n != 1 and
                k2**n != 1 and
                (k1/k2)**n != 1 and
                k1 != 0 and k2 != 0):
                self.k1 = k1
                self.k2 = k2
                return

    def encode_selectors(self):
        # Ensure the state has been initialized
        if not hasattr(self, 'H'):
            raise ValueError("Call update_state before encoding selectors")

        # Interpolate the selector polynomials
        qM_poly = fft_ff_interpolation(self.qM, self.g, self.Fq)
        qL_poly = fft_ff_interpolation(self.qL, self.g, self.Fq)
        qR_poly = fft_ff_interpolation(self.qR, self.g, self.Fq)
        qO_poly = fft_ff_interpolation(self.qO, self.g, self.Fq)
        qC_poly = fft_ff_interpolation(self.qC, self.g, self.Fq)

        return {
            "qM": qM_poly,
            "qL": qL_poly,
            "qR": qR_poly,
            "qO": qO_poly,
            "qC": qC_poly
        }

    def encode_permutation(self):
        if not hasattr(self, 'H') or not hasattr(self, 'k1') or not hasattr(self, 'k2'):
            raise ValueError("Call update_state before encoding permutation")

        n = self.n

        # Function to map position indices to elements in H ł k1H ł k2H
        def index_to_element(i):
            if 0 <= i < n:
                return self.H[i]  # Left wires mapped to H
            elif n <= i < 2*n:
                return self.k1H[i-n]  # Right wires mapped to k1H
            elif 2*n <= i < 3*n:
                return self.k2H[i-2*n]  # Output wires mapped to k2H
            else:
                raise ValueError(f"Index {i} out of range [0, {3*n-1}]")

        # Compute permutation star values for each group
        S_sigma1_values = [index_to_element(self.perm[i]) for i in range(n)]
        S_sigma2_values = [index_to_element(self.perm[i+n]) for i in range(n)]
        S_sigma3_values = [index_to_element(self.perm[i+2*n]) for i in range(n)]

        # Interpolate to get the permutation polynomials
        S_sigma1_poly = fft_ff_interpolation(S_sigma1_values, self.g, self.Fq)
        S_sigma2_poly = fft_ff_interpolation(S_sigma2_values, self.g, self.Fq)
        S_sigma3_poly = fft_ff_interpolation(S_sigma3_values, self.g, self.Fq)

        sigma_star = S_sigma1_values + S_sigma2_values + S_sigma3_values

        return {
            "S_sigma1": S_sigma1_poly,
            "S_sigma2": S_sigma2_poly,
            "S_sigma3": S_sigma3_poly,
            "sigma_star": sigma_star
        }

    def encode_witness(self, w, x_size=0):
        if not hasattr(self, 'H'):
            raise ValueError("Call update_state before encoding witness")

        n = self.n

        # Split witness into a (left), b (right), c (output) values
        a_values = w[:n]
        b_values = w[n:2*n]
        c_values = w[2*n:3*n]

        # Interpolate to get witness polynomials
        a_poly = fft_ff_interpolation(a_values, self.g, self.Fq)
        b_poly = fft_ff_interpolation(b_values, self.g, self.Fq)
        c_poly = fft_ff_interpolation(c_values, self.g, self.Fq)

        # Extract public inputs if specified
        x = w[:x_size] if x_size > 0 else []

        # Compute public input polynomial if there are public inputs
        PI = self.compute_public_input_poly(x) if x_size > 0 else self.R(0)

        return {
            "a": a_poly,
            "b": b_poly,
            "c": c_poly,
            "x": x,
            "PI": PI
        }

    def compute_lagrange_basis(self, i):
        if not hasattr(self, 'H'):
            raise ValueError("Call update_state before computing Lagrange basis")

        n = self.n
        g = self.g
        X = self.X

        # Compute the i-th Lagrange basis polynomial for H
        numerator = g**i * (X**n - 1)
        denominator = n * (X - g**i)
        L_i = numerator // denominator

        return L_i

    def compute_public_input_poly(self, x):
        if not hasattr(self, 'H'):
            raise ValueError("Call update_state before computing public input poly")

        PI = self.R(0)
        for i, x_i in enumerate(x):
            L_i = self.compute_lagrange_basis(i)
            PI -= x_i * L_i

        return PI
```

Code 6: Plonk encoder

```
from kzg import KZG
from plonk.encoder import Encoder

class Indexer:

    def __init__(self, curve_type="bn254"):
        self.kzg = KZG(curve_type=curve_type)
        self.encoder = Encoder(self.kzg.curve_order)

    def preprocess(self, qM, qL, qR, qO, qC, perm, max_degree):
        # Setup KZG commitment scheme
        ck, rk = self.kzg.setup(max_degree)

        # Update encoder state with circuit description
        self.encoder.update_state(qM, qL, qR, qO, qC, perm)

        # Encode circuit into polynomials
        selector_polys = self.encoder.encode_selectors()
        permutation_polys = self.encoder.encode_permutation()

        # Organize polynomials into ordered dictionary
        indexer_polys = {
            "qM": selector_polys["qM"],
            "qL": selector_polys["qL"],
            "qR": selector_polys["qR"],
            "qO": selector_polys["qO"],
            "qC": selector_polys["qC"],
            "S_sigma1": permutation_polys["S_sigma1"],
            "S_sigma2": permutation_polys["S_sigma2"],
            "S_sigma3": permutation_polys["S_sigma3"]
        }

        # Create ordered list for commitment
        poly_list = [
            indexer_polys["qM"],
            indexer_polys["qL"],
            indexer_polys["qR"],
            indexer_polys["qO"],
            indexer_polys["qC"],
            indexer_polys["S_sigma1"],
            indexer_polys["S_sigma2"],
            indexer_polys["S_sigma3"]
        ]

        # Commit to the indexer polynomials
        commitments_list = self.kzg.commit(ck, poly_list)

        # Organize commitments in a dictionary
        indexer_commitments = {
            "qM": commitments_list[0],
            "qL": commitments_list[1],
            "qR": commitments_list[2],
            "qO": commitments_list[3],
            "qC": commitments_list[4],
            "S_sigma1": commitments_list[5],
            "S_sigma2": commitments_list[6],
            "S_sigma3": commitments_list[7]
        }

        # Create index proving key - everything the prover needs
        ipk = {
            "ck": ck,
            "polynomials": indexer_polys,
            "commitments": indexer_commitments,
            # Additional data needed by the prover
            "subgroups": {
                "H": self.encoder.H,
                "n": self.encoder.n,
                "g": self.encoder.g,
                "k1": self.encoder.k1,
                "k2": self.encoder.k2
            },
            "vanishing_poly": self.encoder.v_H,
            "sigma_star": permutation_polys["sigma_star"],
        }

        # Create index verification key - only what the verifier needs
        ivk = {
            "rk": rk,
            "commitments": indexer_commitments,
            "subgroups": {
                "n": self.encoder.n,
                "g": self.encoder.g,
                "k1": self.encoder.k1,
                "k2": self.encoder.k2
            }
        }

        return ipk, ivk
```

Code 7: Plonk indexer

```
from kzg import KZG
from fft_ff import fft_ff_interpolation
from transcript import Transcript
from plonk.encoder import Encoder

class Prover:

    def __init__(self, curve_type="bn254"):
        self.kzg = KZG(curve_type=curve_type)

    def prove(self, ipk, x, w):
        # Extract data from index proving key
        ck = ipk["ck"]
        polynomials = ipk["polynomials"]
        H = ipk["subgroups"]["H"]
        n = ipk["subgroups"]["n"]
        g = ipk["subgroups"]["g"]
        k1 = ipk["subgroups"]["k1"]
        k2 = ipk["subgroups"]["k2"]
        v_H = ipk["vanishing_poly"]
        sigma_star = ipk["sigma_star"]
        Fq = self.kzg.Fq
        R = self.kzg.R
        X = self.kzg.X

        # Create an encoder with the same field
        self.encoder = Encoder(self.kzg.curve_order)

        # Create a transcript for the Fiat-Shamir transform
        transcript = Transcript("plonk-proof", Fq)

        # Add public inputs to the transcript
        transcript.append_message("public-inputs", x)

        # Compute the full witness
        full_witness = x + w

        # Initialize the encoder with a temporary empty permutation (for PI computation)
        empty_perm = [0] * (3 * n)
        empty_selectors = [Fq(0)] * n
        self.encoder.update_state(empty_selectors, empty_selectors, empty_selectors, empty_selectors, empty_selectors, empty_p

        # Compute public input polynomial
        PI = self.encoder.compute_public_input_poly(x)

        # ----- Round 1: Wire polynomials -----
        # Generate random blinding scalars
        b1, b2 = Fq.random_element(), Fq.random_element()
        b3, b4 = Fq.random_element(), Fq.random_element()
        b5, b6 = Fq.random_element(), Fq.random_element()
        b7, b8, b9 = Fq.random_element(), Fq.random_element(), Fq.random_element()

        # Split witness into a, b, c values
        a_values = full_witness[:n]
        b_values = full_witness[n:2*n]
        c_values = full_witness[2*n:3*n]

        # Compute wire polynomials a(X), b(X), c(X) with blinding factors for zero-knowledge
        a_poly = (b1 * X + b2) * v_H + fft_ff_interpolation(a_values, g, Fq)
        b_poly = (b3 * X + b4) * v_H + fft_ff_interpolation(b_values, g, Fq)
        c_poly = (b5 * X + b6) * v_H + fft_ff_interpolation(c_values, g, Fq)

        # Commit to wire polynomials
        wire_polys = [a_poly, b_poly, c_poly]
        wire_commitments = self.kzg.commit(ck, wire_polys)
        a_commit, b_commit, c_commit = wire_commitments

        # Add commitments to transcript
        transcript.append_message("round1-commitments", wire_commitments)

        # ----- Round 2: Permutation polynomial -----
        # Get permutation challenges
        beta = transcript.get_challenge("beta")
        gamma = transcript.get_challenge("gamma")

        # Compute permutation polynomial z(X)
        z_poly = self._compute_permutation_polynomial(
            a_values, b_values, c_values,
            sigma_star,
            beta, gamma, g, k1, k2, n, H, v_H, X, Fq,
            b7, b8, b9
        )

        # Verify first Lagrange polynomial condition: L_1(X)(z(X) - 1) = 0 over H
        L1 = R((X**n - 1) / (n * (X - 1)))
        assert L1 * (z_poly - 1) % v_H == 0, "z_poly does not satisfy L1 condition"

        # Commit to permutation polynomial
        z_commit = self.kzg.commit(ck, [z_poly])[0]

        # Add commitment to transcript
        transcript.append_message("round2-commitment", z_commit)

        # ----- Round 3: Quotient polynomial -----
        # Get quotient challenge
        alpha = transcript.get_challenge("alpha")

        # Compute quotient polynomial t(X)
        t_poly = self._compute_quotient_polynomial(
            a_poly, b_poly, c_poly, z_poly,
            polynomials["qM"], polynomials["qL"], polynomials["qR"],
            polynomials["qO"], polynomials["qC"],
            polynomials["S_sigma1"], polynomials["S_sigma2"], polynomials["S_sigma3"],
            alpha, beta, gamma, PI, v_H, H, n, g, k1, k2, R, X
        )

        # Split t(X) into degree < n polynomials
        t_lo, t_mid, t_hi = self._split_quotient_polynomial(t_poly, n, X, R, Fq)

        # Commit to the parts of t(X)
        t_polys = [t_lo, t_mid, t_hi]
        t_commitments = self.kzg.commit(ck, t_polys)
        t_lo_commit, t_mid_commit, t_hi_commit = t_commitments

        # Add commitments to transcript
        transcript.append_message("round3-commitments", t_commitments)

        # ----- Round 4: Evaluation point -----
        # Get evaluation challenge
        zeta = transcript.get_challenge("zeta")

        # Compute opening evaluations
        a_zeta = a_poly(zeta)
        b_zeta = b_poly(zeta)
        c_zeta = c_poly(zeta)
        s_sigma1_zeta = polynomials["S_sigma1"](zeta)
        s_sigma2_zeta = polynomials["S_sigma2"](zeta)
        z_omega_zeta = z_poly(zeta * g)  # z()

        # Add evaluations to transcript
        evaluations = [a_zeta, b_zeta, c_zeta, s_sigma1_zeta, s_sigma2_zeta, z_omega_zeta]
        transcript.append_message("round4-evaluations", evaluations)

        # ----- Round 5: Opening proof -----
        # Get opening challenge
        v = transcript.get_challenge("v")

        # Compute linearization polynomial r(X)
        r_poly = self._compute_linearization_polynomial(
            a_zeta, b_zeta, c_zeta, s_sigma1_zeta, s_sigma2_zeta, z_omega_zeta,
            polynomials["qM"], polynomials["qL"], polynomials["qR"],
            polynomials["qO"], polynomials["qC"], polynomials["S_sigma3"],
            z_poly, t_lo, t_mid, t_hi, alpha, beta, gamma, zeta, PI, n, k1, k2, R
        )

        # Verify linearization polynomial r() = 0
        assert r_poly(zeta) == 0, "r() should be zero"

        # First batch: Polynomials to be opened at zeta
        zeta_polys = [
            r_poly,
            a_poly,
            b_poly,
            c_poly,
            polynomials["S_sigma1"],
            polynomials["S_sigma2"]
        ]

        # Get the opening proofs
        W_z = self.kzg.open(ck, zeta_polys, zeta, v)
        W_zw = self.kzg.open(ck, [z_poly], zeta * g, v)

        # Assemble the final proof
        proof = {
            "commitments": {
                "a": a_commit,
                "b": b_commit,
                "c": c_commit,
                "z": z_commit,
                "t_lo": t_lo_commit,
                "t_mid": t_mid_commit,
                "t_hi": t_hi_commit,
            },
            "evaluations": {
                "a": a_zeta,
                "b": b_zeta,
                "c": c_zeta,
                "s_sigma1": s_sigma1_zeta,
                "s_sigma2": s_sigma2_zeta,
                "z_omega": z_omega_zeta
            },
            "kzg_proofs": {
                "W_z": W_z,
                "W_zw": W_zw
            }
        }

        return proof

    def _compute_permutation_polynomial(self, a_values, b_values, c_values,
                                        sigma_star,
                                        beta, gamma, g, k1, k2, n, H, v_H, X, Fq,
                                        b7, b8, b9):
        # Add blinding factors for zero-knowledge
        z_poly = (b7 * X**2 + b8 * X + b9) * v_H

        # Compute the permutation accumulator at each point
        z_values = [Fq(1)]  # z() = 1

        for i in range(n - 1):
            # Compute numerator: (a() +  + )(b() + k + )(c() + k + )
            num = ((a_values[i] + beta * H[i] + gamma) *
                   (b_values[i] + beta * k1 * H[i] + gamma) *
                   (c_values[i] + beta * k2 * H[i] + gamma))

            # Compute denominator: (a() + () + )(b() + () + )(c() + () + )
            den = ((a_values[i] + beta * sigma_star[i] + gamma) *
                   (b_values[i] + beta * sigma_star[i + n] + gamma) *
                   (c_values[i] + beta * sigma_star[i + 2*n] + gamma))

            if den == 0:
                # This should be extremely rare and would indicate an issue with the circuit
                raise ValueError("Denominator is zero in permutation polynomial calculation")

            # Update the accumulator: z() = z()num/den
            z_values.append(z_values[-1] * (num / den))

        # Interpolate the accumulator values to get the rest of z(X)
        z_interp = fft_ff_interpolation(z_values, g, Fq)

        # Combine with the blinding part
        z_poly += z_interp

        return z_poly

    def _compute_quotient_polynomial(self, a_poly, b_poly, c_poly, z_poly,
                                     qM, qL, qR, qO, qC,
                                     S_sigma1, S_sigma2, S_sigma3,
                                     alpha, beta, gamma, PI, v_H, H, n, g, k1, k2, R, X):
        # Term 1: Gate constraints
        term1 = R((a_poly * b_poly * qM + a_poly * qL + b_poly * qR + c_poly * qO + PI + qC) / v_H)

        # Term 2: Permutation constraints (part 1)
        term2 = alpha * (z_poly * (a_poly + beta * X + gamma) *
                         (b_poly + beta * k1 * X + gamma) *
                         (c_poly + beta * k2 * X + gamma)) / v_H

        # Term 3: Permutation constraints (part 2)
        z_poly_shifted = R(z_poly(g * X))  # z(X)
        term3 = -alpha * ((a_poly + beta * S_sigma1 + gamma) *
                          (b_poly + beta * S_sigma2 + gamma) *
                          (c_poly + beta * S_sigma3 + gamma) *
                          z_poly_shifted) / v_H

        # Term 4: Copy constraints (first Lagrange basis condition)
        L1 = R((X**n - 1) / (n * (X - 1)))
        term4 = R(alpha**2 * (z_poly - 1) * L1 / v_H)

        # Combine all terms
        t_poly = R(term1 + term2 + term3 + term4)

        return t_poly

    def _split_quotient_polynomial(self, t_poly, n, X, R, Fq):
        # Get coefficients of t_poly
        t_coeffs = t_poly.list()
        t_coeffs.extend([Fq(0)] * (3 * n - len(t_coeffs)))  # Pad to ensure 3n coefficients

        # Split into three parts
        t_lo_coeffs = t_coeffs[:n]
        t_mid_coeffs = t_coeffs[n:2*n]
        t_hi_coeffs = t_coeffs[2*n:]

        # Add random blinding factors for zero-knowledge
        b10, b11 = Fq.random_element(), Fq.random_element()

        # Create the polynomials with blinding
        t_lo = R(t_lo_coeffs) + b10 * X**n
        t_mid = R(t_mid_coeffs) - b10 + b11 * X**n
        t_hi = R(t_hi_coeffs) - b11

        # Verify that t(X) = t_lo + Xˆn * t_mid + Xˆ(2n) * t_hi
        assert t_poly == t_lo + X**n * t_mid + X**(2*n) * t_hi, "t(X) does not equal the sum of its parts"

        return t_lo, t_mid, t_hi

    def _compute_linearization_polynomial(self, a_zeta, b_zeta, c_zeta,
                                          s_sigma1_zeta, s_sigma2_zeta, z_omega_zeta,
                                          qM, qL, qR, qO, qC, S_sigma3,
                                          z_poly, t_lo, t_mid, t_hi, alpha, beta, gamma, zeta, PI, n, k1, k2, R):
        # Compute z_H() = ˆn - 1
```

```python
        z_H_zeta = zeta**n - 1

        # Compute L1()
        L1_zeta = R((z_H_zeta) / (n * (zeta - 1)))

        # Evaluate PI()
        PI_zeta = PI(zeta)

        # Compute the individual terms
        # Gate constraints
        term1 = a_zeta * b_zeta * qM + a_zeta * qL + b_zeta * qR + c_zeta * qO + PI_zeta + qC

        # Permutation constraints (part 1)
        term2 = alpha * ((a_zeta + beta * zeta + gamma) *
                         (b_zeta + beta * k1 * zeta + gamma) *
                         (c_zeta + beta * k2 * zeta + gamma) * z_poly)

        # Permutation constraints (part 2)
        term3 = -alpha * ((a_zeta + beta * s_sigma1_zeta + gamma) *
                          (b_zeta + beta * s_sigma2_zeta + gamma) *
                          (c_zeta + beta * S_sigma3 + gamma) * z_omega_zeta)

        # Copy constraints
        term4 = alpha**2 * (z_poly - 1) * L1_zeta

        # Compute the final linearization polynomial
        # Subtract the quotient polynomial terms
        r_poly = term1 + term2 + term3 + term4 - z_H_zeta * (
            t_lo + zeta**n * t_mid + zeta**(2*n) * t_hi
        )

        return r_poly
```

Code 8: Plonk prover

```python
from kzg import KZG
from transcript import Transcript
from plonk.encoder import Encoder

class Verifier:

    def __init__(self, curve_type="bn254"):
        self.kzg = KZG(curve_type=curve_type)

    def verify(self, ivk, x, proof):
        # Extract data from verification key
        rk = ivk["rk"]
        commitments = ivk["commitments"]
        n = ivk["subgroups"]["n"]
        g = ivk["subgroups"]["g"]
        k1 = ivk["subgroups"]["k1"]
        k2 = ivk["subgroups"]["k2"]

        # Extract proof components
        wire_commitments = [
            proof["commitments"]["a"],
            proof["commitments"]["b"],
            proof["commitments"]["c"]
        ]
        z_comm = proof["commitments"]["z"]
        quotient_commitments = [
            proof["commitments"]["t_lo"],
            proof["commitments"]["t_mid"],
            proof["commitments"]["t_hi"]
        ]
        W_z = proof["kzg_proofs"]["W_z"]
        W_zw = proof["kzg_proofs"]["W_zw"]

        # Extract evaluations
        a_zeta = proof["evaluations"]["a"]
        b_zeta = proof["evaluations"]["b"]
        c_zeta = proof["evaluations"]["c"]
        s_sigma1_zeta = proof["evaluations"]["s_sigma1"]
        s_sigma2_zeta = proof["evaluations"]["s_sigma2"]
        z_omega_zeta = proof["evaluations"]["z_omega"]

        # Extract selector and permutation commitments
        qM_comm = commitments["qM"]
        qL_comm = commitments["qL"]
        qR_comm = commitments["qR"]
        qO_comm = commitments["qO"]
        qC_comm = commitments["qC"]
        s_sigma1_comm = commitments["S_sigma1"]
        s_sigma2_comm = commitments["S_sigma2"]
        s_sigma3_comm = commitments["S_sigma3"]

        # Set up field
        Fq = self.kzg.Fq

        # Create encoder for public input polynomial
        self.encoder = Encoder(self.kzg.curve_order)
        empty_perm = [0] * (3 * n)
        empty_selectors = [Fq(0)] * n
        self.encoder.update_state(empty_selectors, empty_selectors, empty_selectors, empty_selectors, empty_selectors, empty_p

        # Compute public input polynomial
        PI = self.encoder.compute_public_input_poly(x)

        # Create a transcript for the Fiat-Shamir transform
        transcript = Transcript("plonk-proof", Fq)

        # Add public inputs to the transcript
        transcript.append_message("public-inputs", x)

        # Recreate the transcript to recover the same challenges as the prover
        transcript.append_message("round1-commitments", wire_commitments)

        # Get permutation challenges
        beta = transcript.get_challenge("beta")
        gamma = transcript.get_challenge("gamma")

        # Add permutation commitment
        transcript.append_message("round2-commitment", z_comm)

        # Get quotient challenge
        alpha = transcript.get_challenge("alpha")

        # Add quotient commitments
        transcript.append_message("round3-commitments", quotient_commitments)

        # Get evaluation challenge
        zeta = transcript.get_challenge("zeta")

        # Add evaluations
        evaluations = [a_zeta, b_zeta, c_zeta, s_sigma1_zeta, s_sigma2_zeta, z_omega_zeta]
        transcript.append_message("round4-evaluations", evaluations)

        # Get opening challenge
        v = transcript.get_challenge("v")

        # Get multipoint evaluation challenge (for batch verification)
        u = transcript.get_challenge("u")

        # Calculate ZH(zeta) = zeta^n - 1
        ZH_zeta = zeta**n - 1

        # Calculate L1(zeta) = (zeta^n - 1)/(n*(zeta - 1))
        L1_zeta = (zeta**n - 1)/(n * (zeta - 1))

        # Calculate PI(zeta) - public input polynomial evaluation
        PI_zeta = PI(zeta)

        # Compute linearization polynomial r(X) commitment
        # First term: a_zeta * b_zeta * qM(X) + a_zeta * qL(X) + b_zeta * qR(X) + c_zeta * qO(X) + PI(zeta) + qC(X)
        r_comm = self.kzg.multiply(qM_comm, int(a_zeta * b_zeta))
        r_comm = self.kzg.add(r_comm, self.kzg.multiply(qL_comm, int(a_zeta)))
        r_comm = self.kzg.add(r_comm, self.kzg.multiply(qR_comm, int(b_zeta)))
        r_comm = self.kzg.add(r_comm, self.kzg.multiply(qO_comm, int(c_zeta)))
        r_comm = self.kzg.add(r_comm, self.kzg.multiply(self.kzg.G1, int(PI_zeta)))
        r_comm = self.kzg.add(r_comm, qC_comm)

        # Permutation constraint terms ( term)
        # Step 1: Compute (a_zeta + z + )(b_zeta + k1z + )(c_zeta + k2z + )z(X)
        a_term_1 = a_zeta + beta * zeta + gamma
        b_term_1 = b_zeta + beta * k1 * zeta + gamma
        c_term_1 = c_zeta + beta * k2 * zeta + gamma
        factor_1 = a_term_1 * b_term_1 * c_term_1
        term_1 = self.kzg.multiply(z_comm, int(factor_1))

        # Step 2: Compute (c_zeta + S_3(X) + )
        c_poly_term = self.kzg.multiply(s_sigma3_comm, int(beta))  # S_3(X)
        c_poly_term = self.kzg.add(c_poly_term, self.kzg.multiply(self.kzg.G1, int(c_zeta + gamma)))  # c_zeta + S_3(X) +

        # Step 3: Compute (a_zeta + s_1_zeta + )(b_zeta + s_2_zeta + )(c_zeta + S_3(X) + )z__zeta
        a_term_2 = a_zeta + beta * s_sigma1_zeta + gamma
        b_term_2 = b_zeta + beta * s_sigma2_zeta + gamma
        factor_2 = a_term_2 * b_term_2 * z_omega_zeta
        term_2 = self.kzg.multiply(c_poly_term, int(factor_2))

        # Step 4: Compute term_1 - term_2
        perm_diff = self.kzg.add(term_1, self.kzg.neg(term_2))

        # Step 5: Multiply by
        perm_term = self.kzg.multiply(perm_diff, int(alpha))

        # Add permutation term to r_comm
        r_comm = self.kzg.add(r_comm, perm_term)

        # Add copy constraint term: *[(z(X) - 1)*L1(zeta)]
        factor3 = alpha**2 * L1_zeta
        z_minus_1 = self.kzg.add(z_comm, self.kzg.neg(self.kzg.G1))
        r_comm = self.kzg.add(r_comm, self.kzg.multiply(z_minus_1, int(factor3)))

        # Subtract quotient polynomial terms: -ZH(zeta)*(t_lo(X) + zeta^n*t_mid(X) + zeta^(2n)*t_hi(X))
        t_combined = self.kzg.add(proof["commitments"]["t_lo"],
                                  self.kzg.multiply(proof["commitments"]["t_mid"], int(zeta**n)))
        t_combined = self.kzg.add(t_combined,
                                  self.kzg.multiply(proof["commitments"]["t_hi"], int(zeta**(2*n))))
        r_comm = self.kzg.add(r_comm, self.kzg.neg(self.kzg.multiply(t_combined, int(ZH_zeta))))

        # Prepare polynomials and evaluations for verification
        # First batch: Polynomials evaluated at zeta
        zeta_commitments = [
            r_comm,           # r(X) (linearization polynomial)
            wire_commitments[0],  # a(X)
            wire_commitments[1],  # b(X)
            wire_commitments[2],  # c(X)
            s_sigma1_comm,    # S_sigma1(X)
            s_sigma2_comm     # S_sigma2(X)
        ]

        # Evaluations at zeta
        zeta_evaluations = [
            Fq(0),            # r(zeta) = 0
            a_zeta,           # a(zeta)
            b_zeta,           # b(zeta)
            c_zeta,           # c(zeta)
            s_sigma1_zeta,    # S_sigma1(zeta)
            s_sigma2_zeta     # S_sigma2(zeta)
        ]

        # Second batch: Polynomial z(X) evaluated at zeta * g
        zw_commitments = [z_comm]
        zw_evaluations = [z_omega_zeta]

        # Verify batch opening proofs
        commitment_list = [zeta_commitments, zw_commitments]
        z_values = [zeta, zeta * g]
        evaluations_list = [zeta_evaluations, zw_evaluations]
        proof_list = [W_z, W_zw]
        xi_values = [v] * 2
        batch_result = self.kzg.batch_check(rk, commitment_list, z_values, evaluations_list, proof_list, xi_values, u)

        return batch_result
```

Code 9: Plonk verifier