A Gemini-based Implementation of Bulletproofs: Analyzing Time and Space Tradeoffs

BY

FRANCESCA STELLA
B.S., Politecnico di Torino, Torino, Italy, 2024

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Chicago, 2026

Chicago, Illinois

Defense Committee:
Alexander R. Block, Chair and Advisor
Chris Kanich
Daniele Bringhenti, Politecnico di Torino

Accessibility Statement

An accessible EPUB version of this document can be obtained by contacting fstel@uic.edu.

Acknowledgements

I would like to express my deepest gratitude to Professor Alexander R. Block for his invaluable guidance and support throughout this project; his passion for cryptography has been truly inspiring.

I would also like to thank my parents, Marco and Cristina, for supporting me and giving me the opportunity to have this important experience. In difficult moments, they always listened to me and showed they were proud of me regardless of the results.

I also want to thank my brother Alessandro and his wife Cristina for always be there; no matter how much we grow up, we will always be "ciuchi".

A special thank you goes to my love, Christian. He always believed in me and gave me courage when I needed it most. He shared the joy of every goal I reached and always supported me in my choices.

I want to thank my grandparents, who always valued my studies and were happy for every result I achieved, whether big or small.

I am also grateful to my friends, who stayed by my side during the hard times. We shared travels and happy moments that were essential during these challenging years. A special thank you to my best friend, Isabella, who always put me first and listened to me, even from a distance. I also want to thank Aurora, a childhood friend that I am so happy to have found again after many years.

Finally, a huge thank you to my dog, Potter. No matter how my day went, he always welcomed me home with a wagging tail and made me smile even when no one else could.

Thank you to all of you for helping me make this dream come true.

FS

Contents

Accessibility Statement ii

Acknowledgements iii

List of Figures vi

List of Tables vii

List of Abbreviations viii

Summary ix

Chapter 1. Introduction 1
1. Motivation 2
2. Our Contributions 2
3. Thesis Organization 3

Chapter 2. Background 4
1. Zero-Knowledge Proofs 4
2. ZK-SNARKs: Succint Non-Interactive Arguments of Knowledge 4
3. Modern SNARK Designs 6
4. The KZG Polynomial Commitment Scheme 9
5. Bulletproofs 10

Chapter 3. Gemini Framework 13
1. Architectural Limitations of the Original Implementation 13
2. Why KZG is Essential for Gemini 14
3. Generalized API for Polynomial Commitments 14
4. Implementation Issues with Gemini 15

Chapter 4. Time-Optimized Implementation (bp_time) 17
1. Architecture 17
2. Implementation 18

Chapter 5. Space-Efficient Implementation (bp_space) 26
1. Architecture 26
2. Implementation 27

Chapter 6. Experimental Results 51
1. Experimental Setup 51
2. Testing Methodology 53
3. Micro-benchmark Results 54
4. Macro-benchmark Results 57

5. Conclusions 58

Chapter 7. Conclusions and Future Work 60
1. Conclusions 60
2. Future work 61

Appendix A. Software Repository 62

Appendix B. Benchmark Data Tables 63

Bibliography 69

Vita 71
Education 71
Academic Experience 71
Languages 71

List of Figures

1 Interactive zero-knowledge proof 5
2 Non-interactive zero-knowledge proof 5

1 Execution time and memory usage for the commitment phase. 54
2 Execution time and memory usage for the opening phase. 55
3 Execution time and memory usage for the multi-opening phase. 56
4 Execution time for the key generation phase. 57
5 Execution time and memory usage for the complete SNARK proof generation. 57
6 Verification time for the complete SNARK proof generation. 58

List of Tables

1 Correspondence between theoretical notation and implementation variables for `batch_commit`. 19
2 Correspondence between theoretical notation and implementation variables for `open` 20
3 Correspondence between theoretical notation and implementation variables for `open_multi_points` 23

1 Correspondence between theoretical notation and implementation variables for `batch_commit` (Space-Efficient) 28
2 Correspondence between theoretical notation and implementation variables for space-efficient `open` 29
3 Correspondence between theoretical notation and implementation variables for space-efficient `open_multi_points` 31
4 Correspondence between theoretical notation and implementation variables for `compute_z_g` 34
5 Correspondence between theoretical notation and implementation variables for `compute_multi_z_g` 37
6 Correspondence between theoretical notation and implementation variables for space-efficient `compute_final_z_g` 38
7 Correspondence between notation and implementation variables for `compute_final_multi_z_g` 41

1 BP Commit Data 63
2 BP Open Data 64
3 BP Open_Multi Data 64
4 BP SNARK Data 65
5 KZG Commit Data 65
6 KZG Open Data 66
7 KZG Open_Multi Data 66
8 KZG SNARK Data 67
9 SNARK Verifier Data 67
10 Keygen Data 68

List of Abbreviations

API	Application Programming Interface
BP	Bulletproofs
FS	Fiat-Shamir (Heuristic/Transcript)
IOP	Interactive Oracle Proof
IP	Inner Product
IPA	Inner Product Argument
KZG	Kate-Zaverucha-Goldberg (Polynomial Commitment Scheme)
MSM	Multi-Scalar Multiplication
PCS	Polynomial Commitment Scheme
PIOP	Polynomial Interactive Oracle Proof
RAM	Random Access Memory
SNARK	Succinct Non-interactive Argument of Knowledge
ZK	Zero-Knowledge
ZKP	Zero-Knowledge Proof
ZK-SNARK	Zero-Knowledge Succinct Non-interactive Argument of Knowledge

Summary

With the rapid development of verifiable computation and blockchain technologies, Zero-Knowledge Proofs (ZKPs) have emerged as a fundamental tool for ensuring privacy and security. However, one of the main challenges of ZKP is the significant memory overhead required to prove complex statements. The primary focus of this thesis is addressing the challenge of memory-efficient proving using space-optimized architectures for SNARKs.

The research focuses on the Gemini framework, an advanced protocol developed for space efficiency through streaming techniques and the sum-check protocol. A generalized API for the Gemini framework was created and used to define a common interface that can support multiple different commitment schemes.

Using the Gemini architecture, we implement and compare two architectures: `bp_time`, that uses the entire amount of RAM available, and `bp_space`, which is a space-efficient prover that employs even-odd folding and streaming. To validate our contributions, we performed an experimental study, in which we measure execution times and peak memory usage for commitment sizes ranging from 2^{12} to 2^{20}. We compare our transparent Bulletproofs-based implementation against KZG-based baseline, to illustrate the trade-offs between trusted and transparent setups.

Our results clearly show that the KZG-based approach has significantly faster performance compared to the Bulletproofs-based approach. However, Bulletproofs supports a transparent setup that removes the requirement for trusted third parties and cryptographic pairings. Through our benchmarks, we provide a real world study demonstrating the costs and benefits of implementing trustless architecture compared to architectures requiring trust and cryptographic pairings.

CHAPTER 1

Introduction

Zero-Knowledge Proofs are a fundamental tool in modern blockchain applications and verifiable computation systems. One of the main motivations relies on the fact that they enable one party, a computationally powerful prover, such as a cloud computing system, to assure another party, a computationally weak verifier, such as a consumer laptop or mobile device, that a certain statement is true, without exposing any additional information. This property is extremely precious in fields where privacy and security are of primary importance, such as financial transactions, identity verification and secure voting systems.

However, programs that can be proven using zero-knowledge proofs are currently complex and inefficient, requiring a significant amount of computational resources. To prove the correctness of a program, the latter must be converted into a large circuit of logic gates or a set of polynomial constraints and both the Prover and the Verifier need access to this circuit to execute their tasks. The size of the polynomial depends on the complexity of the program: the more complex the program, the larger the polynomial. For this reason, the proving time and the size of the proof can be very large, making the use of Zero-Knowledge proofs challenging in real-world applications.

Historically, the main focus of research in this area has been on the optimization of prover time, verifier time and proof size. A number of significant results have already been achieved in relation to these areas: for a statement of size N, there are protocols that can achieve $O(N)$ prover time, while verifier time and proof size can be reduced to $O(\log N)$ or even $O(1)$, depending on the cryptographic assumptions.

Although time complexity has been greatly improved, space complexity still remains one of the biggest bottlenecks: a program that runs in time T and space S requires the prover to

manifest a circuit of size $\Omega(T)$. This memory requirement quickly becomes very challenging even for medium values of T.

1. Motivation

The motivation of this thesis is the need to bridge the gap between the theoretical advancements in zero-knowledge proofs and their practical implementation. This work takes as its main theoretical reference the paper "Public-Coin Zero-Knowledge Arguments with (almost) Minimal Time and Space Overheads" by Block et al. [**1**], which proposes a novel approach to optimize the memory consumption during the proving process. The opportunity to actually realize this research is offered by the emergence of the Gemini framework [**14**], due to Bootle et al. [**5**], which demonstrates the possibility of managing circuits through a streaming approach. This thesis provides the first concrete implementation of Block et al.'s space-efficient implementation of the *Bulletproofs* [**13**] polynomial commitment scheme. Our implementation is done within the Gemini framework, built upon the `arkworks` [**10**] zkSNARK framework in Rust [**11**].

2. Our Contributions

The major contributions of this thesis include:

- **Implementation of Zero-Knowledge Proofs Within Gemini**: Block et al. [**1**] introduced novel techniques for space-efficient zero-knowledge proofs. In this work, we implement these protocols for the first time.
- **Extension to Univariate Polynomials**: While Block et al.'s work focuses on multilinear polynomials, we extended the implementation to support univariate polynomials.
- **Batch Evaluation**: We implemented an `open_multi_points` function. This new function enables the prover to batch open *multiple* evaluation proofs for a single polynomial. Block et al.'s work did not support this functionality.
- **API and Framework Interaction**: We built an API that allows the interaction between the space-efficient components and the Gemini environment. In building

this API, we discover several aspects of Gemini as implemented that, to the best of our knowledge, has not been documented, and yet is essential to interfacing with the Gemini system.

- **Extensive Benchmarking**: We run extensive micro and macro benchmarks of our implementation. Our benchmarks measure execution time and memory consumption. Moreover, we additionally benchmark the original Gemini system and compare our implementation to the reference implementation of Gemini.

3. Thesis Organization

The research presented in this thesis starts from the discussion of theoretical foundations, in order to provide the context for the main objective, that is to develop and explain practical implementations. Following this introduction, in Chapter 2 theoretical background is discussed, covering Zero-Knowledge Proofs, and Polynomial Commitment Schemes. Chapter 3 introduces the Gemini framework, providing a generalized API designed to facilitate the integration of different commitment schemes. This chapter also describes practical difficulties that arose during the integration of Gemini and provide the technical justification for the new architectures described in subsequent chapters. Chapter 4 describes the `bp_time` implementation, putting the focus on a time-optimized architecture that minimizes prover latency by leveraging full RAM availability. Instead, Chapter 5 presents `bp_space`, a space-efficient implementation with the aim to achieve a logarithmic memory footprint using streaming and even-odd folding techniques. Chapter 6 provides the experimental results and a performance analysis, in order to compare the effect of the polynomial size on the computational time in the different implementations. Finally, Chapter 7 summarizes the research key findings and outlines potential directions for future work.

CHAPTER 2

Background

This chapter provides the theoretical foundations necessary to better understand the subsequent sections of the Thesis. In the first part, we introduce the basic concepts of zero-knowledge proofs and the construction of polynomial commitment schemes. Then, we discuss the role of these schemes in the context of zero-knowledge proofs. Finally, we present an overview of the Gemini framework.

1. Zero-Knowledge Proofs

Zero-Knowledge Proofs (ZKPs) are a cryptographic method that allows the prover to convince the verifier to know a certain piece of information, without revealing the information itself. A solid definition of zero-knowledge proofs can be taken from the research of Shafi Goldwasser, Silvio Micali and Charles Rackoff in the 1980s, who introduced the concept in their paper "The Knowledge Complexity of Interactive Proof Systems" [**15**].

Definition[Informal, [**15**]] 'Zero-knowledge proof systems are those in which the verifier learns nothing from the prover other than the fact that the assertion being proved is true.' Zero-Knowledge proofs can be classified into two main categories:

- Interactive: the prover and the verifier exchange messages.
- Non-interactive: The prover sends only one single message to the verifier, without any other interaction.

In this Thesis, we focus on non-interactive zero-knowledge proofs.

2. ZK-SNARKs: Succint Non-Interactive Arguments of Knowledge

ZK-SNARKs are a specific type of non-interactive zero-knowledge proofs that rapresent a technical implementation needed to allow these proofs to be usable in real-world applications. A ZK-SNARK is a protocol characterized by the following properties:

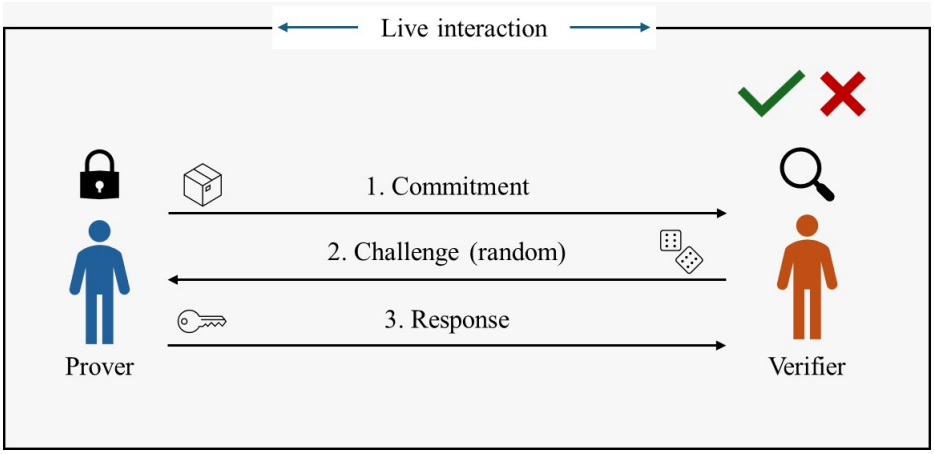


FIGURE 1. Interactive zero-knowledge proof

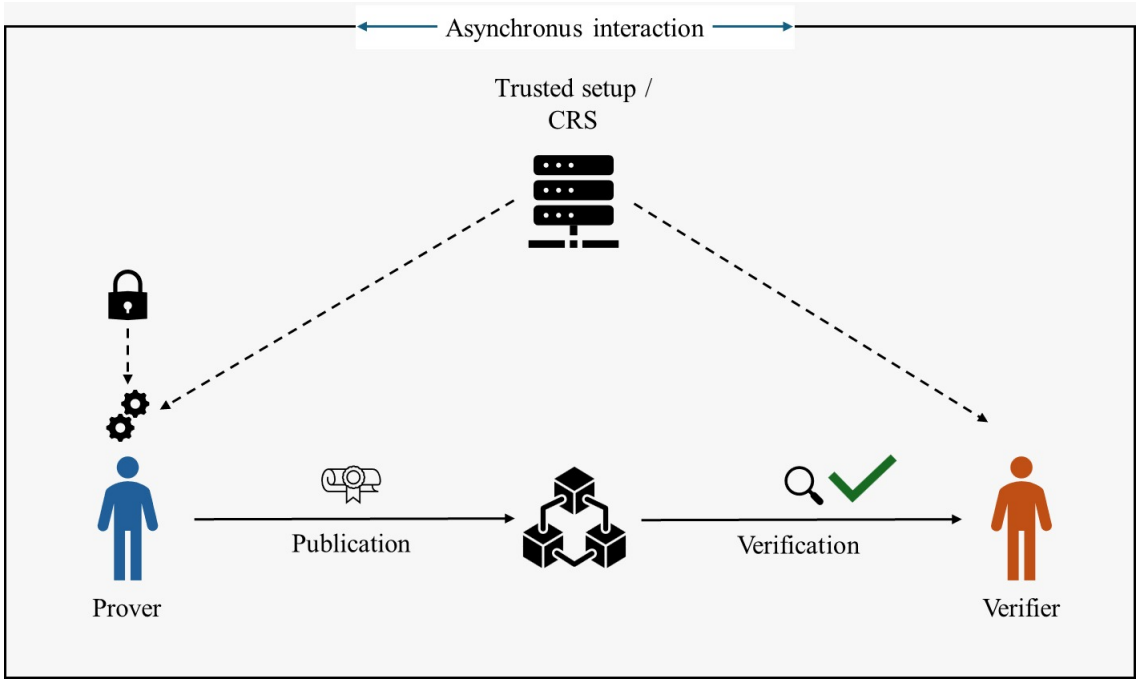


FIGURE 2. Non-interactive zero-knowledge proof

- **S (Succint)**: The size of the proof is extremely small, typically no more than a few hundred bytes.
- **N (Non-interactive)**: The prover can generate the proof without any interaction with the verifier.
- **AR (Argument)**: The soundness of the proof relies on cryptographic assumptions.

- **K (Knowledge)**: The prover must have the knowledge of a witness that satisfies the statement being proved.

Despite the advantages of ZK-SNARKs, in order to generate a SNARK, as described previously, the Prover must construct a (very) large circuit or set of polynomial constraints to prove correctness of the underlying statement. The Gemini framework, that will be discussed in the next sections, is able to manage big polynomials through a streaming approach, allowing to reduce the memory requirements of the Prover.

3. Modern SNARK Designs

In recent years, there has been a significant transition of the design of SNARKs, from monolithic designs to a modular approach. This allows researchers to improve performance of single components, without redesigning the entire protocol. One popular design methodology to construct modern efficient SNARKs is composed of three main components:

- **Polynomial Interactive Oracle Proof (PIOP)** [**16, 6**]: The theoretical basis for the SNARK system.
- **Polynomial Commitment Scheme (PCS)** [**4**]: The cryptographic portion of the SNARK system.
- **Fiat-Shamir Heuristic** [**3**]: This is the mechanism which transforms an interactive PIOP into a non-interactive version.

3.1. (Polynomial) Interactive Oracle Proofs. A new area of research started when Ben-Sasson et al. [**16**] and Reingold, Rothblum, and Rothblum [**17**] formally defined Polynomial Interactive Oracle Proofs (IOPs), creating a framework to describe zero knowledge arguments within the model in which a prover has access to a polynomial oracle. Bünz, Fisch, and Szepieniec [**6**] later extended this model and defined *Polynomial* IOPs (PIOPs).

The PIOP model describes the interaction between a Prover and a Verifier in an abstracted model where the Prover does not send all string data to the Verifier. Instead, it provides the Verifier with access to polynomial oracles. In a PIOP, the Prover "sends" a polynomial $f(x)$ to an ideal third-party (the oracle). The Verifier never actually receives the

coefficients of $f(x)$ directly, however the Verifier can query the oracle at any desired value z, receiving as feedback the result of the evaluation, $y = f(z)$. If the Prover wants to prove that a vector of values satisfies certain constraints, the PIOP determines how those values are represented as polynomials and what identity must be satisfied for the statement to be true.

3.2. Polynomial Commitment Schemes. In order to create a SNARK, a program is converted into a set of polynomial equations (such as R1CS). If the program is complex, the resulting polynomial can have a very high degree, making it difficult to manage and evaluate. A polynomial commitment scheme is the main instrument used for the construction of zero-knowledge proofs. The role of polynomial commitment schemes is to "compress" the polynomial into a single commitment. Kate, Zaverucha, and Goldberg [**4**] introduced the concept of a polynomial commitment scheme in their paper "Constant-Size Commitments to Polynomials and Their Applications."

Definition[E.g., [**1**]] A polynomial commitment scheme (PCS) allows a committer to commit to a polynomial with a short string that can be used by a verifier to confirm claimed evaluations of the committed polynomial.

It consists of the following algorithms:

- **Setup**: Generates the public parameters required for the scheme, based on a security parameter and the maximum degree of the polynomial.
- **Commit**: Takes a polynomial and the public parameters to produce a short commitment string (the "commiment").
- **Open**: Reveals the original polynomial so that the verifier can check it against the commitment.
- **VerifyPoly**: Checks if a given polynomial actually matches the original commitment produced by the committer.
- **CreateWitness**: Generates a small piece of evidence (a witness) to prove that the polynomial, when evaluated at a specific point, results in a specific value.

- **VerifyEval**: Uses the commitment, the evaluation point, the result, and the witness to confirm that the evaluation is correct without needing the full polynomial.

The main properties of a polynomial commitment scheme are:

- **Binding**: Once a commitment is made, the committer cannot change the polynomial
- **Hiding**: The commitment itself does not reveal any information about the polynomial to the verifier before the opening or evaluation phase.
- **Evaluation binding**: It is computationally impossible for a committer to provide a valid witness for a false evaluation of the committed polynomial.

A PCS supports different types of polynomials representations. In the context of the Gemini framework, we primarily focus on Univariate Polynomials of degree $d \leq N-1$, defined as $f(X) = \sum_{i=0}^{N-1} f_i X^i$. Alternatively, other protocols use the Multilinear Polynomials (n-variate), where the degree in each variable is at most 1.

3.3. The Fiat-Shamir Transformation. The protocols described in the PIOP and PCS sections are inherently interactive. That means the Verifier will send random challenges to the Prover, who will then respond. The Fiat-Shamir transformation [**3**] is a method used to convert a public-coin interactive proof to a non-interactive proof. Here, an interactive proof is *public-coin* if all messages sent by the verifier are uniformly random challenges (from some challenge space). The main idea is to replace the public-coin (random challenges) sent by the Verifier to the Prover with the output of a hash function. Thus, instead of having to wait for the prover to receive a new challenge value from an outside source, the Prover generates its own random challenge by hashing its prior messages:

$$challenge_i = H(\text{message}_1, \text{message}_2, \ldots, \text{message}_i) \tag{1}$$

Since the Prover is not able to know what the output of a secure hash function will be (it is completely unpredictable), the Prover can not "cheat" by selecting a random challenge that would favor a false assertion.

4. The KZG Polynomial Commitment Scheme

KZG is a polynomial commitment scheme proposed by Kate, Zaverucha, and Goldberg [**4**], and it is used today in many modern SNARK implementations. KZG can generate constant size commitments and proofs independent of the degree of the polynomial that they are committing. The KZG scheme operates over pairing-based elliptic curve groups. KZG requires a trusted setup to create a set of public parameters. This is generated from a sequence of group elements:

$$(g, g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^{\ell}}) = (g^{\tau^i})_{i\in[0,\ell]} \tag{2}$$

where $g \in \mathbb{G}$ is a group generator and $\tau \in \mathbb{F}$ (the scalar field of $\mathbb{G}$) is a secret value that needs to be destroyed immediately after the trusted setup, in order to maintain the security of the scheme and ℓ is the maximum degree of the polynomials that the scheme can support.

The protocol follows three main steps:

- **Commit**: Given a polynomial $\phi(X) = \sum_{i\in[0,\deg\phi]} \phi_i X^i$, the prover uses the public parameters $(g^{\tau^i})_{i\in[0,\ell]}$ to compute a single group element $c = g^{\phi(\tau)}$, calculated as:

$$c = \prod_{i\in[0,\deg\phi]} (g^{\tau^i})^{\phi_i} \tag{3}$$

- **Open**: In order to show that $\phi(z) = y$ at a particular point z, the prover calculates a "proof" $\pi = g^{q(\tau)}$. This witness is a commitment to a quotient polynomial $q(x)$:

$$q(x) = \frac{\phi(x) - y}{x - z} \tag{4}$$

- **Verify**: The verifier then checks the correctness of the witness using bilinear pairings, but does not know τ. The verification equation shows that:

$$e(\pi, g^{\tau} \cdot g^{-z}) = e(c \cdot g^{-y}, g) \tag{5}$$

  Where e denotes a bilinear map, $z \in \mathbb{F}$ is the evaluation point, and $y = \phi(z)$ is the claimed evaluation.

5. Bulletproofs

Bulletproofs [**12, 13**] have been a major breakthrough in Zero-Knowledge Proof systems. In the context of PCS, Bulletproofs are most commonly used as IPA, that enables a Prover to demonstrate to a Verifier that a vector of coefficients $\mathbf{w}$ satisfies a specific evaluation identity, without exposing the coefficients themselves. The main objective is to prove the following relation:

$$P(z) = \langle \mathbf{w}, \mathbf{z} \rangle = v \tag{6}$$

where $\mathbf{w} = (w_0, \ldots, w_{n-1}) \in \mathbb{F}^n$ is the vector representing the coefficients of some polynomial, and $\mathbf{z} = (1, z, z^2, \ldots, z^{n-1}) \in \mathbb{F}^n$ is the vector of powers of the evaluation point z. The prover commits to the vector $\mathbf{w}$ and the value v with the generated commitment:

$$C = \langle \mathbf{w}, \mathbf{g} \rangle + h \cdot v, \tag{7}$$

where $\mathbf{g} = (g_0, \ldots, g_{n-1})$ and h are generators of a group $\mathbb{G}$. The standard Bulletproofs protocol typically divides a vector into two contiguous parts (left and right). However, as described by [**1**], this methodology is not streaming friendly because the prover will need to store the entire witness in RAM. In this Thesis, we use the even-odd folding strategy, proposed by [**1**], to obtain space efficiency. This methodology divides the vector using the parity of the indices, allowing the Prover to manage the witness in a single streaming pass:

$$\mathbf{w}_e = (w_0, w_2, \ldots, w_{n-2}) \tag{8}$$

$$\mathbf{w}_o = (w_1, w_3, \ldots, w_{n-1}). \tag{9}$$

In every round k (with $k = 0, \ldots, \log_2(n) - 1$), the Prover decreases vectors from size n down to size $n/2$. The current vectors are divided in two parts: "even" and "odd".

$$\mathbf{w} = (\mathbf{w}_e, \mathbf{w}_o), \tag{10}$$

$$\mathbf{g} = (\mathbf{g}_e, \mathbf{g}_o), \tag{11}$$

$$\mathbf{z} = (\mathbf{z}_e, \mathbf{z}_o). \tag{12}$$

The prover calculates two cross-term commitments:

$$L_k = \langle \mathbf{w}_e, \mathbf{g}_o \rangle + h \cdot \langle \mathbf{w}_e, \mathbf{z}_o \rangle, \tag{13}$$

$$R_k = \langle \mathbf{w}_o, \mathbf{g}_e \rangle + h \cdot \langle \mathbf{w}_o, \mathbf{z}_e \rangle. \tag{14}$$

Then, the Prover sends L_k and R_k to the Verifier and receives a challenge x_k. The Prover combines both vectors:

$$\mathbf{w}' = \mathbf{w}_e \cdot x_k + \mathbf{w}_o \cdot x_k^{-1} \tag{15}$$

$$\mathbf{g}' = \mathbf{g}_e \cdot x_k^{-1} + \mathbf{g}_o \cdot x_k \tag{16}$$

$$\mathbf{z}' = \mathbf{z}_e \cdot x_k + \mathbf{z}_o \cdot x_k^{-1} \tag{17}$$

After $\log_2(n)$ rounds, the vectors are reduced to a single element y_{final} and a single generator g_{final}. The final proof contains the pair $(L_k, R_k)_{k=0}^{\log_2 n - 1}$ and the scalar y_{final}.

Algorithm 1: Even-Odd Folding IPA

Input: $\mathbf{w} \in \mathbb{F}^n, z \in \mathbb{F}, \mathbf{g} \in \mathbb{G}^n, h \in \mathbb{G}$

Output: $\pi = \left(\{(L_k, R_k)\}_{k=0}^{\log_2 n - 1}, w_{final}\right)$

// Initialize

$\mathbf{z} \leftarrow (1, z, \ldots, z^{n-1})$;

$v \leftarrow \langle \mathbf{w}, \mathbf{z} \rangle, \quad C \leftarrow \langle \mathbf{w}, \mathbf{g} \rangle + h \cdot v$;

for $k = 0$ **to** $\log_2(n) - 1$ **do**

  // Even-Odd virtual split

  $\mathbf{w}_e, \mathbf{w}_o \leftarrow (w_0, \ldots, w_{n-2}), (w_1, \ldots, w_{n-1})$;

  $\mathbf{g}_e, \mathbf{g}_o \leftarrow (g_0, \ldots, g_{n-2}), (g_1, \ldots, g_{n-1})$;

  $\mathbf{z}_e, \mathbf{z}_o \leftarrow (z_0, \ldots, z_{n-2}), (z_1, \ldots, z_{n-1})$;

  $y_L \leftarrow \langle \mathbf{w}_e, \mathbf{z}_o \rangle, \quad y_R \leftarrow \langle \mathbf{w}_o, \mathbf{z}_e \rangle$;

  $L_k \leftarrow \langle \mathbf{w}_e, \mathbf{g}_o \rangle + h \cdot y_L$;

  $R_k \leftarrow \langle \mathbf{w}_o, \mathbf{g}_e \rangle + h \cdot y_R$;

  // Transcript and challenge

  Append L_k, R_k to Transcript;

  $x_k \leftarrow \text{GetChallenge(Transcript)}, \quad x_k^{-1} \leftarrow \text{inv}(x_k)$;

  // Folding

  $\mathbf{w} \leftarrow \mathbf{w}_e \cdot x_k + \mathbf{w}_o \cdot x_k^{-1}$;

  $\mathbf{g} \leftarrow \mathbf{g}_e \cdot x_k^{-1} + \mathbf{g}_o \cdot x_k$;

  $\mathbf{z} \leftarrow \mathbf{z}_e \cdot x_k^{-1} + \mathbf{z}_o \cdot x_k$;

$w_{final} \leftarrow w_0$;

return $\pi = \left(\{(L_k, R_k)\}_{k=0}^{\log_2(n)-1}, w_{final}\right)$;

CHAPTER 3

Gemini Framework

In classical SNARK constructions, the Prover needs to upload the entire vector of coefficients of the polynomial into memory. Gemini [**5**] is a framework used to manage large polynomials through a streaming approach, avoiding this memory limitation. The streaming model allows the prover to stream the coefficients of a large degree-d polynomial $f = (f_0, \ldots, f_d)$ sequentially, starting with f_0 (or, in some cases, f_d). Then, the prover can either advance the current position of the stream (e.g., move from f_i to f_{i+1}), or *reset* the stream to f_0. Gemini framework divides a SNARK construction into three fundamental components:

- **PIOP (Polynomial Interactive Oracle Proofs)**: an interactive protocol where the Prover creates an oracle and the Verifier can query it.
- **PCS (Polynomial Commitment Scheme)**: a mechanism used to commit to and evaluate polynomials.
- **Fiat-Shamir Heuristic**: a method that allows the transformation of an interactive protocol into a non-interactive one. In the context of Gemini, it is implemented through a Merlin-based transcript [**18**].

1. Architectural Limitations of the Original Implementation

Gemini Framework is based on this theoretical modularity, but the actual implementation was tailored to a specific environment. The original implementation of Gemini is written in order to be compatible with the KZG scheme, that is based on Pairing-friendly groups. Consequently, the entire structure was designed assuming the use of only Pairing-based schemes. A modular framework should be able to support different types of PCS. Since

Bulletproofs does not use Pairings, the implementation of Gemini was adapted to support more general PCS.

2. Why KZG is Essential for Gemini

In the context of Gemini, KZG is essential because, at the cost of a trusted setup, it ensures the following properties:

- Efficient proving,
- *Constant-time* verification, and
- *Constant-sized* proofs.

This is important for many applications where there are limitations due to either bandwidth or on-chain storage constraints. In the verification phase, KZG uses a constant number of pairings regardless of the degree of the polynomial. Thus, it allows the Verifier to verify Gemini's proofs almost instantly. Moreover, in KZG, the Verifier does not need to have access or to reconstruct the polynomials. This aligns with the Space-Efficient goal of Gemini, as neither the Prover nor the Verifier need to keep large quantities of information in memory to ensure the validity of a proof. We remark also that one of the main contributions of Gemini [**5**] was adapting KZG to the streaming model to allow for space-efficient commitments and opening proofs.

3. Generalized API for Polynomial Commitments

The main problem to solve was to implement a generalized API compatible with different types of PCS. It was necessary to identify the core functions that a PCS should have to be consistent with the Gemini framework. The main functions identified are:

- **commit()**: $\mathsf{commit}(\phi) \to \mathcal{C}$. Computes a commitment $\mathcal{C}$ to a polynomial $\phi(X)$.
- **open()** and **open_multi_points()**: $\mathsf{open}(\phi, \zeta) \to \pi$. Generates an evaluation proof (witness) π certifying that $\phi(\zeta) = \nu$. We also provide **open_multi_points** for batching evaluations at multiple points; i.e., proving multiple statements of the form $\phi(\zeta_i) = \nu_i$ for $i \in \{1, \ldots, \ell\}$ for some $\ell \geq 1$.

- **verify()**: $\mathsf{verify}(\mathcal{C}, \zeta, \nu, \pi) \to \{0, 1\}$. Validates the proof π against the commitment $\mathcal{C}$ and the evaluation result.
- **batch_commit()**: $\mathsf{batch_commit}(\{\phi_i\}) \to \mathcal{C}_{batch}$. Efficiently commits to a vector of polynomials.
- **commit_folding()**, **open_folding()**: commit_folding and open_folding. These functions are important for the Gemini framework as they handle the recursive reduction of polynomials (e.g., in Bulletproofs-style or IPA-style folding), enabling space-efficient arguments.

4. Implementation Issues with Gemini

The previously discussed `commit_folding` and `open_folding` are not sufficient for making Bulletproofs compatible with the Gemini SNARK. In fact, the space-efficient prover should give the exact same SNARK proof as the time-efficient prover. In this sense, space-efficiency is just a way to compute the time-efficient proof in a space-efficient manner. However, only using the standard implementation of `commit_folding` and `open_folding` (which assumes the coefficient stream is in order from smallest to largest degree coefficients) leads to different SNARK proofs. Upon investigation, we discovered the issue was due to the specific way Gemini follows to implement some of its streaming algorithms. These streaming algorithms assume reverse streaming access to the underlying polynomials and the `FoldedPolynomialTree`. Specifically:

- For some polynomial $h = (h_0, \ldots, h_d)$, the algorithm assumes a stream of the form $(h_d, h_{d-1}, \ldots, h_0)$.
- For a polynomial $f^{(0)}$ and object `FoldedPolynomialTree`, it assumes streaming access to each $f^{(1)}, \ldots, f^{(c)}$ in reverse.

The standard `commit_folding` and `open_folding` assume streaming access from lowest-order coefficient to highest-order Therefore, they are incompatible with the required access in reverse order. Moreover, it is not possible to simply "reverse" the iterator passed by Gemini to interface with these functions, because this operation would break other parts of the Gemini

framework. One last incompatibility arises: our `open_rev` function must additionally take as input another (reverse) polynomial stream P to batch with the `FoldedPolynomialTree` polynomials $f^{(1)}, \ldots, f^{(c)}$. With KZG, this step is unnecessary, as the evaluation proof for the stream P can simply be added homomorphically to the evaluation proof for a `FoldedPolynomialTree` to obtain an evaluation proof for both; however, this is not possible with Bulletproofs.

To hande these incompatibilities, we add two new functions: `commit_folding_rev` and `open_multi_folding_rev`. The biggest technical challenge behind these two algorithms is the need to calculate the actual index of each coefficient within a reverse stream. Differently from the standard forward-order (i.e., degree-0 coefficient to degree-d coefficient) approach, in our case the prover has to handle coefficients starting from the highest degree. For example, when the `FoldedPolynomialTree` begins streaming, the first tuple received is $(1, f^{(1)}_{d_1})$, followed by $(1, f^{(1)}_{d_1-1})$, and eventually $(2, f^{(2)}_{d_2})$, where d_1 and d_2 represent the degrees of $f^{(1)}$ and $f^{(2)}$ respectively. The logic within the `open_multi_folding_rev` function becomes significantly more complex. In the forward version of `open_folding`, the stream may end at an index j that is *not* a power of 2 (which is necessary to compute the correct folding). To handle this, we simply check if we are at the end of the stream then "advance" the stream to the next power of two larger than j (treating the "missing" coefficients as 0). This occurs only if the degree of $f^{(i)}$ is not a multiple of 2^k (`k_pow` in our algorithms we define later). Intuitively, $f^{(i)}$ is processed in chunks of size `k_pow`; the final chunk consists of any "leftover" coefficients from the stream. If the degree of $f^{(i)}$ is a power of two, we never need to perform this stream "advance" action. In the reverse-order implementation, the algorithm must correctly identify these boundaries; it has to compute the corresponding folded coefficients, and the (g_k, z_k) pairs, performing a backward-processing of the data. This synchronization remains the most important technical challenge to ensure the correctness of the `open_multi_folding_rev` protocol.

CHAPTER 4

Time-Optimized Implementation (bp_time)

The `bp_time` module has as main goal the implementation of the Bulletproofs-based polynomial commitment scheme compatible with the Gemini framework. This implementation does not consider memory constraints, but is instead optimized for the fastest possible prover.

1. Architecture

The primary objective of `bp_time` is to minimize the Prover's latency. In this context, latency refers to the time required to generate a complete evaluation proof for a given polynomial. In our implementation, we focus on **Univariate Polynomials** of degree $d \leq n-1$, where n is a power of two:

$$f(X) = \sum_{i=0}^{n-1} w_i X^i \tag{18}$$

The coefficients are represented as a dense vector $\mathbf{w} = (w_0, w_1, \ldots, w_{n-1}) \in \mathbb{F}^n$. To commit to this polynomial, the architecture utilizes a `CommitterKey` consisting of a vector of n independent group generators $\mathbf{g} = (g_0, \ldots, g_{n-1})$ and an additional generator h (the `ip_gen`) used for the inner product argument. The core of the `bp_time` architecture is the **Even-Odd (E/O) folding** strategy. Unlike the traditional Left-Right split, this approach partitions the vectors $\mathbf{w}, \mathbf{z}, \mathbf{g}$ into their even (e) and odd (o) indices. In each reduction round k (for $k = 0, \ldots, \log_2(n) - 1$), the prover computes cross-term commitments L_k and R_k:

$$L_k = \langle \mathbf{w}_e, \mathbf{g}_o \rangle + h \cdot \langle \mathbf{w}_e, \mathbf{z}_o \rangle \tag{19}$$

$$R_k = \langle \mathbf{w}_o, \mathbf{g}_e \rangle + h \cdot \langle \mathbf{w}_o, \mathbf{z}_e \rangle \tag{20}$$

Upon receiving a challenge α, the vectors are reduced linearly:

$$\mathbf{w}' = \alpha \mathbf{w}_e + \alpha^{-1} \mathbf{w}_o \tag{21}$$

$$\mathbf{z}' = \alpha^{-1} \mathbf{z}_e + \alpha \mathbf{z}_o \tag{22}$$

$$\mathbf{g}' = \alpha^{-1} \mathbf{g}_e + \alpha \mathbf{g}_o \tag{23}$$

To maintain a clear link between the theoretical protocol and the Rust implementation, it is important to map the mathematical symbols to the variables used in `bp_time`

2. Implementation

The implementation of `bp_time` is written in Rust, utilizing the `arkworks` ecosystem. The protocol flow is managed through three main functions:

(1) `batch_commit`: Computes the initial commitment using a Multi-Scalar Multiplication (MSM). This operation is parallelized via the `Rayon` library to utilize all available CPU cores.

(2) `open`: Executes the recursive folding process. At each round, the function allocates new vectors for the reduced instances. It ensures the fastest possible computation by maintaining data in contiguous memory blocks.

(3) `open_multi_points`: It focuses on proving multiple evaluations of the same polynomial $p(X)$ at different points $z_0, z_1, \ldots, z_{k-1}$ without repeating the opening protocol.

2.1. `batch_commit` **Function.** The commitment phase of this protocol is the first step. To facilitate the readability of the code, we provide a mapping between the mathematical notation and the variables used in the implementation in Table 1.

In the `bp_time`, we use MSM to compute the commitment $C = \langle \mathbf{w}, \mathbf{g} \rangle$, as shown in Listing 4.1

LISTING 4.1. Our Commitment Implementation.

```
1 Commitment(E::msm_unchecked(&self.vec_g, polynomial))
```

TABLE 1. Correspondence between theoretical notation and implementation variables for `batch_commit`.

Math Symbol	Rust Variable	Description
$\{\phi_i(X)\}_{i\in[m]}$	`polynomials`	The collection of m polynomials to be committed.
$\phi_i(X)$	`p`	The individual polynomial currently being processed by the iterator.
$\mathrm{Com}(\phi_i)$	`self.commit(p)`	The commitment function applied to each polynomial.
$\{C_i\}_{i\in[m]}$	`Vec<Commitment<E>>`	The resulting vector of commitments returned to the caller.
$\mathbf{g}$	`self.vec_g`	The vector of group generators used as the basis for the commitments.

In order to manage multiple polynomials, we create a `batch_commit` function as described in Listing 4.2.

LISTING 4.2. Parallel commitment implementation in bp_time

```
pub fn batch_commit<J>(&self, polynomials: J) -> Vec<Commitment<E>>
where
    J: IntoParallelIterator,
    J::Item: Borrow<Vec<E::ScalarField>>,
{
    polynomials
        .into_par_iter()
        .map(|p| self.commit(p.borrow()))
        .collect::<Vec<_>>()
}
```

We utilize `.into_par_iter()` from the `Rayon` library [**19**]. Thus, the prover can use all the available CPU processors to perform MSM for each polynomial.

2.2. `open` Function. At the beginning of every round, the witness $\mathbf{w}$, the generators$\mathbf{g}$ and the evaluation vector $\mathbf{z}$ are split into even and odd components. To facilitate the

TABLE 2. Correspondence between theoretical notation and implementation variables for `open`

Math Symbol	Rust Variable	Description
$\mathbf{w}$	`polynomial` / `a`	The vector of polynomial coefficients, padded with zeros to match the generators' length.
z	`eval_point`	The single point at which the polynomial is opened.
$\mathbf{z}$	`z`	The evaluation vector $(1, z, z^2, \ldots, z^{n-1})$.
h	`self.ip_gen`	The auxiliary generator used for the Inner Product binding.
$\mathbf{w}_{ev}, \mathbf{w}_{od}$	`a_ev`, `a_od`	Even and odd indexed coefficients of the current polynomial vector.
$\mathbf{g}_{ev}, \mathbf{g}_{od}$	`ag_ev`, `ag_od`	Even and odd indexed group generators at the current folding step.
y_L, y_R	`ev_inner_prod`, `od_inner_prod`	Inner product cross-terms used to bind the evaluation during folding.
L_k, R_k	`com_ev`, `com_od`	Left and Right commitments (even/odd cross-terms) for the k-th round.
x, x^{-1}	`x`, `x_inv`	The verifier's challenge and its modular inverse for the folding step.
w_0	`a[0]` (final)	The final scalar remaining after all folding rounds (the "folded" witness).
π	`EvaluationProof`	The final proof object containing all round commitments and the final scalar.

readability of the code, we provide a mapping between the mathematical notation and the variables used in the implementation in Table 2. As shown in Listing 4.3, the prover partitions the current coefficient vector `a` into its even and odd components using Rust's `step_by` and `skip` iterators. In each round, the prover computes two cross-term commitments, `com_ev` and `com_od`, which are then sent to the verifier through the transcript to generate the challenge `x`. This iterative process continues until the vectors reach a length of 1, reducing the polynomial evaluation to a single scalar proof.

LISTING 4.3. Implementation of the `open` function in `bp_time`

```
1    while a.len() > 1 {
2        assert!(a.len() % 2 == 0);
3        let a_ev: Vec<E::ScalarField> = a.iter().step_by(2).cloned()
             .collect();
4        let a_od: Vec<E::ScalarField> = a.iter().skip(1).step_by(2).
             cloned().collect();
5        let ag_ev: Vec<E::Affine> = current_gen.iter().step_by(2).
             cloned().collect();
6        let ag_od: Vec<E::Affine> = current_gen.iter().skip(1).
             step_by(2).cloned().collect();
7        let z_ev: Vec<E::ScalarField> = z.iter().step_by(2).cloned()
             .collect();
8        let z_od: Vec<E::ScalarField> = z.iter().skip(1).step_by(2).
             cloned().collect();
```

`a_ev` and `a_od` represent the even and odd components of the witness $\mathbf{w}$; we use `step_by(2)` to get elements at indeces $(0, 2, 4 \dots)$ and `skip(1).step_by(2)` for indices $(1, 3, 5 \dots)$. `ag_ev` and `ag_od` represent the even and odd components of the generators ($\mathbf{g_e}, \mathbf{g_o}$), respectively. Then the prover computes the cross terms L_k and R_k.

```
1        let ev_inner_prod = ip(&a_ev, &z_od);
2        let od_inner_prod = ip(&a_od, &z_ev);
3        let com_ev = E::msm(
4            &ag_od.iter().chain(once(&self.ip_gen)).cloned().collect
                 ::<Vec<_>>(),
5            &a_ev.iter().chain(once(&ev_inner_prod)).cloned().
                 collect::<Vec<_>>(),
6        ).unwrap();
7        let com_od = E::msm(
```

```
8               &ag_ev.iter().chain(core::iter::once(&self.ip_gen)).
                    cloned().collect::<Vec<_>>(),
9               &a_od.iter().chain(core::iter::once(&od_inner_prod)).
                    cloned().collect::<Vec<_>>(),
10          ).unwrap();
```

In this case `com_ev` implements $C_L = \langle \mathbf{w}_e, \mathbf{g}_o \rangle + h \cdot y_L$. `chain(once(&self.ip_gen))` is used to add the special generator h to the generator vector After adding the commitments to the transcript, the Prover obtains the challenge `x`. This challenge is used to linearly combine the even and odd parts, preparing the vectors for the next round as already described in the equation X.

```
1           even_coms.push(com_ev);
2           odd_coms.push(com_od);
3           transcript.append_serializable(b"even_com", &com_ev);
4           transcript.append_serializable(b"odd_com", &com_od);
5           let x: E::ScalarField = transcript.get_challenge(b"
                bp_eval_verif_chal");
6           ver_chals.push(x);
7           let x_inv = x.inverse().unwrap();
8           a = a_ev.iter().zip(&a_od)
9               .map(|(ev, od)| (x * ev) + (x_inv * od))
10              .collect();
11          current_gen = ag_ev.iter().zip(&ag_od)
12              .map(|(ev, od)| (ev.mul(x_inv) + od.mul(x)).into_affine
                    ())
13              .collect();
14          z = z_ev.iter().zip(&z_od)
15              .map(|(ev, od)| (x_inv * ev) + (x * od))
16              .collect();
17      }
```

TABLE 3. Correspondence between theoretical notation and implementation variables for `open_multi_points`

Math Symbol	**Rust Variable**	**Description**
$\mathbf{w}$	`polynomial` / `a`	The vector of polynomial coefficients, padded to match the generators' length.
$\{z_0, \ldots, z_{k-1}\}$	`eval_points`	The set of points at which the polynomial is evaluated.
β	`beta`	Random challenge used for the linear combination of multiple evaluation claims.
$\mathbf{z}$	`z`	The batched evaluation vector, computed as the linear combination of the powers of each z_i.
h	`self.ip_gen`	The dedicated generator used for binding the inner product value in the commitments.
z_i^j	`p_val`	The j-th power of the i-th evaluation point, used to construct the batched evaluation vector.

2.3. `open_multi_points` **Function.** It is important to note that in the Block et al. protocol, there was no defined method to open multiple points. Thus, we have developed this functionality from scratch, to enhance the original protocol's flexibility. Algorithm 2 (at the end of this section) describes the logic we implemented in the `open_multi_points`, while Listing 4.4 specifies our implementation.

The `bp_time` module includes an implementation for opening a single polynomial $p(X)$ at multiple evaluation points $\{z_1, z_2, \ldots, z_k\}$. As outlined in the following reasoning, this is achieved by reducing k separate evaluations into a single Inner Product Argument. Let $\mathbf{w}$ be the vector of coefficients of $p(X)$, and let $Z_i = (1, z_i, z_i^2, \ldots, z_i^{N-1})$ be the vector of powers for each evaluation point z_i. Instead of proving each $y_i = \langle \mathbf{w}, Z_i \rangle$ individually, we

incorporate a random challenge β to batch these proofs, which the Prover uses to crate a linear combination:

$$y = \sum_{i=0}^{k-1} \beta^i y_i \tag{24}$$

$$\mathbf{z} = \sum_{i=0}^{k-1} \beta^i \mathbf{z}_i \tag{25}$$

To facilitate the readability of the code, we provide a mapping between the mathematical notation and the variables used in the implementation in Table 3.

First, the Prover has to guarantee the non-interactivity of the proof. In order to do so, the Prover gets a random challenge `beta` from the transcript (in the code, `beta` refers to the mathematical β).

LISTING 4.4. `open_multi_points` logic

```
1     open_multi_points(&self, polynomial, eval_points, transcript) {
2     // 1. Obtain the batching challenge beta from the transcript
3     beta = transcript.get_challenge("batching");
```

The most-time consuming part of this function is the construction of the vector $\mathbf{z}$. For every evaluation point z_i, it needs to generate all possible powers and sum them up.

```
1     let beta_powers = powers(beta, eval_points.len());
2     let mut z = vec![E::ScalarField::ZERO; a.len()];
3     for (z_i, beta_i) in eval_points.iter().zip(beta_powers.iter())
         {
4         let z_i_powers = powers(*z_i, a.len());
5         for (j, p_val) in z_i_powers.iter().enumerate() {
6             z[j] += *beta_i * p_val;
7         }
8     }
9 }
```

$\mathbf{z}$ is iteratively modified to correspond to the expression $\sum \beta^i \mathbf{z}_i$. After $\mathbf{z}$ is constructed, the problem is reduced to a simple opening (as described in Section 2.2).

Algorithm 2: Open Multi-Points Algorithm

Data: Polynomial coefficients $\mathbf{a} \in \mathbb{F}^N$, generators $\mathbf{g} \in \mathbb{G}^N$, evaluation points $\{z_0, \ldots, z_{k-1}\}$, transcript $\mathcal{T}$
Result: Evaluation Proof $\pi = (\text{even_coms}, \text{odd_coms}, a_{final})$

begin
- $\beta \leftarrow \mathcal{T}$.get_challenge("batching_challenge")
- $\mathbf{z} \leftarrow \sum_{i=0}^{k-1} \beta^i \cdot (1, z_i, z_i^2, \ldots, z_i^{N-1})$
- Initialize even_coms $\leftarrow [\,]$, odd_coms $\leftarrow [\,]$
- **while** $|\mathbf{a}| > 1$ **do**
  - $\mathbf{a}_{ev}, \mathbf{a}_{od} \leftarrow$ split_even_odd($\mathbf{a}$)
  - $\mathbf{g}_{ev}, \mathbf{g}_{od} \leftarrow$ split_even_odd($\mathbf{g}$)
  - $\mathbf{z}_{ev}, \mathbf{z}_{od} \leftarrow$ split_even_odd($\mathbf{z}$)
  - `// Compute cross-term inner products`
  - $ip_{ev} \leftarrow \langle \mathbf{a}_{ev}, \mathbf{z}_{od} \rangle, \quad ip_{od} \leftarrow \langle \mathbf{a}_{od}, \mathbf{z}_{ev} \rangle$
  - `// Commit to cross-terms (MSM)`
  - $com_{ev} \leftarrow \text{MSM}(\mathbf{g}_{od} \cup \{g_{ip}\}, \mathbf{a}_{ev} \cup \{ip_{ev}\})$
  - $com_{od} \leftarrow \text{MSM}(\mathbf{g}_{ev} \cup \{g_{ip}\}, \mathbf{a}_{od} \cup \{ip_{od}\})$
  - even_coms.push(com_{ev}), odd_coms.push(com_{od})
  - $\mathcal{T}$.append(com_{ev}, com_{od})
  - `// Get challenge and update vectors`
  - $x \leftarrow \mathcal{T}$.get_challenge("bp_eval_verif_chal")
  - $\mathbf{a} \leftarrow x \cdot \mathbf{a}_{ev} + x^{-1} \cdot \mathbf{a}_{od}$
  - $\mathbf{g} \leftarrow x^{-1} \cdot \mathbf{g}_{ev} + x \cdot \mathbf{g}_{od}$
  - $\mathbf{z} \leftarrow x^{-1} \cdot \mathbf{z}_{ev} + x \cdot \mathbf{z}_{od}$
- **return** (even_coms, odd_coms, $\mathbf{a}[0]$)

CHAPTER 5

Space-Efficient Implementation (bp_space)

This chapter describes `bp_space`, a version of the protocol designed to save memory. While the `bp_time` version (Chapter 4) loads everything into RAM to be as fast as possible, this implementation works differently because it uses a streaming approach that allows the system to handle large polynomials, as proposed in the Gemini framework.

1. Architecture

The main idea is that RAM is a limited rsource. In the code there is the distinction between the data that is streamed and the data that is not:

- **Witness Streaming:** The polynomial coefficients are accessed as an `Iterable` stream. The prover reads one coefficient, processes it, and moves to the next.
- **Random Access to Keys:** The commitment keys (generators $\mathbf{g}$) require random access due to the folding pattern of Bulletproofs. As shown in Listing 5.1, to handle this, the implementation keeps the keys in memory to allow jumping between different indices during the protocol. Note that as discussed by Block et al. [**1**], in practice one can implement random access to group generators via a small and efficiently computable hash function $h\colon \{0,1\}^* \to \mathbb{G}$.

Listing 5.1. Committer Key structure and Streaming Commitment

```
// The structure keeps generators in RAM for Random Access
pub struct CommitterKeyStream<G> where G: CurveGroup {
    pub(crate) s_vec_g: Vec<G::Affine>,
    pub(crate) s_ip_gen: G::Affine,
}
```

2. Implementation

The main functions implemented are:

- `batch_commit`: Iterates over the polynomial coefficients to multiply them with their respective generator, while adding the results together.
- `open`: calculates the proof for an evaluation point.
- `open_multi_points`: generates a proof for many different evaluation points of the same polynomial.
- `compute_z_g`: a helper function which computes the folded evaluation points as well as the folded generators.
- `compute_multi_z_g`: function with similar purpose to `compute_z_g`, but designed for the `open_multi_points` function.
- `compute_final_z_g`: A function that creates the final folded evaluation and generator.
- `compute_final_multi_z_g`: A function that creates the final folded evaluation and generator for `open_multi_points` function.

2.1. The `batch_commit` Function. The `batch_commit` function in this module does not take a full vector as input, but an object that implements the `Iterable` trait. This allows the prover to compute the commitment by accumulating the results one element at a time. Table 1 gives the correspondence between the theoretical variables and the Rust implementation. We give discuss the implementation given in Listing 5.2.

Listing 5.2. Streaming commitment implementation

```
1    pub fn batch_commit<'b, F>(
2        &self,
3        polynomials: &[&'b dyn Iterable<Item = F, Iter = &mut dyn
           Iterator<Item = F>>],
4    ) -> Vec<Commitment<G>>
5    where
```

```
6           F: Borrow<G::ScalarField>,
7       {
8           polynomials.iter().map(|&p| self.commit(p)).collect()
9       }
```

TABLE 1. Correspondence between theoretical notation and implementation variables for `batch_commit` (Space-Efficient)

Math Symbol	**Rust Variable**	**Description**
$\{\phi_i(X)\}_{i\in[m]}$	`polynomials`	The collection of m polynomials to be committed, passed as a slice of `Iterable` traits.
$\mathbf{g}$	`self.s_vec_g`	The space-efficient vector of group generators.
$\phi_i(X)$	`p`	The specific polynomial currently being committed during the mapping process.
$\text{Com}(\phi_i)$	`self.commit(p)`	The core commitment function that iterates over the streamed coefficients to produce a group element.
$\{C_i\}_{i\in[m]}$	`Vec<Commitment<G>>`	The resulting vector of commitments, collected after processing each polynomial.

2.2. The `open` Function. The `Open` function processes original polynomial coefficients sequentially in $log(n)$ iterations. Unlike the time-based version that creates and asigns new and smaller vectors during each round, `bp_space` processes the original polynomial data stream with a constant memory usage of $O(1)$ per iteration. In each round `k`, the prover has to compute both the Cross-Term Commitments L_k and R_k (`ev_com` and `od_com`) from the previous folded witness. This can be done with Multiscalar-Multiplication (MSM). In our case, we realize MSM by computing the individual contribution of every single original

TABLE 2. Correspondence between theoretical notation and implementation variables for space-efficient `open`

Math Symbol	Rust Variable	Description
$\phi(X)$	`polynomial`	The polynomial to be opened, streamed via the `Iterable` trait.
z	`eval_point`	The evaluation point for the polynomial opening.
k	`k`	The current folding round index $(0 \dots \log N - 1)$.
x_j, x_j^{-1}	`alpha_j`, `aj_inv`	Verifier challenges from previous rounds, used to compute the "virtual" folded coefficients.
$a_i^{(k)}$	`y_k`	The i-th coefficient of the folded polynomial at round k, computed on-the-fly.
$g_i^{(k)}, z_i^{(k)}$	`gk`, `zk`	The i-th folded generator and evaluation scalar, returned by the `compute_z_g` sub-routine.
L_k, R_k	`ev_com`, `od_com`	Even and odd cross-term commitments for the current round.
h	`self.s_ip_gen`	The auxiliary space-efficient generator for inner product binding.
$a_0^{(\log N)}$	`y_final`	The final scalar resulting from the complete folding of the polynomial.

Coefficient w_i to the actual folded state at the end of round `k` "on the fly". We use a Transcript object called `merlin::Transcript` to generate challenges x_k (stored in `ver_chals`) that, after each round's commitments, are added to the transcript.

Specifically, for each coefficient w_i at index `stream_ind`, the function:

- Determines a temporary factor `tmp` based on analyzing the first $k-1$ bits.
- Decides how to route the contribution to either the "even" part (`ev_com`) or the "odd" part (`od_com`) of this level based on the k-th bit of `stream_ind` (`bit_n_k`).

By making these routing decisions, we are able to collect all contributions together into exactly two group elements per round and we are limited to storing at most O(1) amounts of memory per witness. This is shown below in the following code snippet:

LISTING 5.3. Core logic of the streaming `open` function

```
1 for (stream_ind, curr_coeff) in polynomial.iter().enumerate() {
2     let coeff = *curr_coeff.borrow();
```

```
3       let mut tmp = G::ScalarField::ONE;
4           for j in 0..k {
5           let bit_j = (stream_ind >> j) & 1;
6           let alpha_j = ver_chals[j];
7           let aj_inv = alpha_j.inverse().unwrap();
8
9           if bit_j == 0 { tmp *= alpha_j; }
10          else { tmp *= aj_inv; }
11      }
12      y_k += tmp * coeff;
```

In this block, `tmp` calculates the cumulative products of the challenge. If the j-th bit of the index is 0, the element belongs to the "even" partition of that round and is scaled by x_j; otherwise, it is scaled by x_j^{-1}

```
1   count += 1;
2       if count == k_pow {
3           let bit_n_k = (stream_ind >> k) & 1;
4           let (zk, gk) = self.compute_z_g(k, stream_ind >> k,
                eval_point, &ver_chals);
5           if bit_n_k == 0 {
6               ev_com += gk * y_k;
7               ev_inner_prod += y_k*zk;
8           } else {
9               od_com += gk * y_k;
10              od_inner_prod += y_k*zk;
11          }
12          y_k = G::ScalarField::ZERO;
13          count = 0;
14      }
15  }
```

TABLE 3. Correspondence between theoretical notation and implementation variables for space-efficient `open_multi_points`

Math Symbol	Rust Variable	Description
$\phi(X)$	`polynomial`	The polynomial to be opened, streamed via the `Iterable` trait.
$\{z_0, \ldots, z_{k-1}\}$	`eval_points`	The set of multiple evaluation points.
β	`beta`	The batching challenge used to combine multiple evaluation claims into one.
k	`k`	The current folding round index $(0 \ldots \log N - 1)$.
$a_i^{(k)}$	`y_k`	The i-th virtual coefficient of the folded polynomial at round k.
$g_i^{(k)}, \mathbf{z}_i^{(k)}$	`gk`, `zk`	The folded generator and the batched evaluation scalar returned by `compute_multi_z_g`.
L_k, R_k	`ev_com`, `od_com`	Even and odd commitments for the k-th round.
h	`self.s_ip_gen`	The auxiliary space-efficient generator for inner product binding.
$a_0^{(\log N)}$	`y_final`	The final scalar resulting from the complete streaming folding of the polynomial.

This logic implements the following mathematical operations specified in our background:

- The variable `bit_n_k` checks the (k)-th bit of the index. If it is 0, the element contributes to $\mathbf{w}_e$, otherwise to $\mathbf{w}_o$.
- The helper function `compute_z_g` calculates the folded generators and evaluation points, in order to be synchronized with the witness stream.
- Instead of storing the vectors $\mathbf{w}_e$ and $\mathbf{w}_o$, we accumulate their products into `ev_com` (L_k) and `od_com` (R_k).

After the $\log(n)$ rounds, the function computes `y_final` by using the following formula:

$$y_{final} = \sum_{i=0}^{N-1} w_i \cdot \prod_{j=0}^{\log n - 1} x_j^{b_j(i)} \tag{26}$$

2.3. The `open_multi_points` function. The `open_multi_points` extends the logic of `open` function to handle multiple evaluation points $\{z_0, z_1, \ldots, z_{k-1}\}$. A batching challenge β is first obtained from the Transcript `merlin::Transcript`.

```
let beta: G::ScalarField = transcript.get_challenge(b"
    batching_challenge");
```

Then, the function performs the $\log(n)$ rounds. In each round, it iterates over the stream of polynomials.

LISTING 5.4. Implementation of the space-efficient `open_multi_points` function.

```
for (stream_ind, curr_coeff) in polynomial.iter().enumerate() {
    let coeff = *curr_coeff.borrow();
    let mut tmp = G::ScalarField::ONE;
    for j in 0..k {
        let bit_j = (stream_ind >> j) & 1;
        let alpha_j = ver_chals[j];
        let aj_inv = alpha_j.inverse().unwrap();
        if bit_j == 0 {
            tmp *= alpha_j;
        } else {
            tmp *= aj_inv;
        }
    }
    y_k += tmp * coeff;
```

In this block, `y_k` represents the folded witness w_i at round k. The code processes each coefficient in a similar manner as the standard `open` function. In the following block, `zk`, is a single element in the combined evaluation vector $\mathbf{z} = \sum \beta^i \mathbf{z}_i$ at the k-th folding level.

(1) In the `compute_multi_z_g` function we calculate each zk on-the-fly using the formula $zk = \sum \beta^i \cdot z_i^{(k)}$.

(2) Using the k-th bit of the index (`bit_n_k`), we determine whether the current folded witness `y_k` will be considered a part of either $\mathbf{w}_e$ (even) or $\mathbf{w}_o$ (odd).

(3) After we have finished processing the entire stream for that round, we use `ev_inner_prod` and `od_inner_prod` to blind the the cross-term commitments L_k and R_k prior to pushing them into the transcript.

```
            count += 1;
            if count == k_pow {
                let bit_n_k = (stream_ind >> k) & 1;
                let (zk, gk) = self.compute_multi_z_g(k, stream_ind
                   >> k, eval_points, &ver_chals, &beta);
                if bit_n_k == 0 {
                    ev_com += gk * y_k;
                    ev_inner_prod += y_k * zk;
                } else {
                    od_com += gk * y_k;
                    od_inner_prod += y_k * zk;
                }
                y_k = G::ScalarField::ZERO;
                count = 0;
            }
        }
```

After all $\log N$ rounds are completed, the Prover performs one last pass over the polynomial to compute the final scalar y_{final}.

```
    let mut y_final = G::ScalarField::ZERO;
    for (stream_ind, curr_coeff) in polynomial.iter().enumerate() {
        let coeff = *curr_coeff.borrow();
        let mut tmp = G::ScalarField::ONE;
        for j in 0..n_rounds {
            let bit_j = (stream_ind >> j) & 1;
            let aj = ver_chals[j];
            let aj_inv = aj.inverse().unwrap();
```

TABLE 4. Correspondence between theoretical notation and implementation variables for `compute_z_g`

Math Symbol	Rust Variable	Description
k	`k`	The current folding round index.
z	`eval_point`	The scalar evaluation point.
$\{x_j\}_{j\in[k]}$	`ver_chals`	The sequence of verifier challenges from previous rounds.
2^k	`k_pow`	The size of the current folding block.
tmp	`tmp`	The combined weight of verifier challenges for the current coefficient within the block.
$z^{(k)}$	`z`	The accumulated folded evaluation scalar for the current round.
$g^{(k)}$	`g`	The accumulated folded generator (group element) for the current round.
$\mathbf{g}$	`self.s_vec_g`	The base vector of space-efficient group generators.

```
9               if bit_j == 0 {
10                  tmp *= aj;
11              } else {
12                  tmp *= aj_inv;
13              }
14          }
15          y_final += tmp * coeff;
```

2.4. The `compute_z_g` Function. The `compute_z_g` function generates the components required for each round k. It identifies the coefficient w_i's corresponding partner in the virtual folding. Algorithm 3 combines the `Computez` function and the `Computeg` function developed by Block et al. [**1**]. Moreover, in Block et al.'s work, these functions were implemented in the context of *multilinear polynomials*. In this work, we modify the description of `Computez` to be compatible with *univariate* polynomials (note that `Computeg` remains unchanged from [**1**]).

`compute_z_g` computes the following values:

Algorithm 3: Combined Computation of z and g

Input: Dimension k, stream index s, evaluation point ζ, verifier challenges $\boldsymbol{\alpha} = (\alpha_0, \ldots, \alpha_{k-1})$, bases $\mathbf{G}$
Output: Scalar $z \in \mathbb{F}$, group element $g \in \mathbb{G}$

$z \leftarrow 0_{\mathbb{F}}, \quad g \leftarrow 0_{\mathbb{G}}$;
$c \leftarrow (s \text{ is even})?(s+1) : (s-1)$; `// Determine the companion stream`
for $a = 0$ **to** $2^k - 1$ **do**
  $tmp \leftarrow 1_{\mathbb{F}}$;
  for $j = 0$ **to** $k - 1$ **do**
    if $(a \gg j)\&1 == 1$ **then** $tmp \leftarrow tmp \cdot \alpha_j$;
    else $tmp \leftarrow tmp \cdot \alpha_j^{-1}$;
  $index \leftarrow (c \cdot 2^k) + a$;
  $z \leftarrow z + tmp \cdot \zeta^{index}$;
  $g \leftarrow g + tmp \cdot \mathbf{G}[index]$;
return (z, g);

- Folded evaluation point
- Folded generator

The reconstruction is based on the calculation of `tmp`, which indicates how much weight the verifier has assigned to a particular index i, which can be expressed as:

$$tmp = \prod_{j=0}^{k-1} x_j^{(b_{i,j})} \tag{27}$$

where $b_{i,j}$ is the j-th bit of the index i.

Listing 5.5. Calculation of evaluations and generators

```
let c = if stream_ind % 2 == 0 {
    stream_ind + 1
} else {
    stream_ind - 1
};
for a in 0..k_pow {
    let mut tmp = G::ScalarField::ONE;
    for j in 0..k {
        let bit_j = (a >> j) & 1;
```

```
10          let aj = ver_chals[j];
11          let aj_inv = aj.inverse().unwrap();
12          if bit_j == 1 {
13              tmp *= aj;
14          } else {
15              tmp *= aj_inv;
16          }
17      }
```

Finally, the function performs an inner product to return the virtual folded generator and evaluation value.

```
1               let index = (c << k) + a;
2               z += tmp * (eval_point.pow([index as u64]);
3               g += self.s_vec_g[index] * tmp;
4           }
5           (z, g)
6       }
```

`compute_z_g` enables the prover to generate commitments for each round k without needing to store any of the intermediate folded polynomial states.

2.5. The `compute_multi_z_g` Function. A new contribution of the thesis was to create and implement the `compute_multi_z_g` function, since Block et al.'s work only focused on the single-point version of the protocol. This function extends the `Computez` function of Block et al.'s work to handle batching *multiple* evaluation points $z_0, \ldots, z_{k-1}$ together with challenge β. The `compute_multi_z_g` function is an extension of `compute_z_g`, optimized to process multiple evaluation points. The difference is in how the folded evaluation part of z is computed: raher than computing a power of a single point the function computes a linear combination of evaluations across all points in `eval_points`.

TABLE 5. Correspondence between theoretical notation and implementation variables for `compute_multi_z_g`

Math Symbol	Rust Variable	Description
$\{z_0, \ldots, z_{k-1}\}$	`eval_points`	The set of multiple points at which the polynomial is being evaluated.
β	`batch_chal`	The random challenge used to combine multiple evaluations into a single scalar claim.
$\{x_j\}_{j\in[k]}$	`ver_chals`	The sequence of verifier challenges accumulated from previous rounds.
tmp	`tmp`	The product of verifier challenges (or their inverses) determining the weight of the i-th folded coefficient.
$z_{multi}^{(k)}$	`z`	The accumulated batched evaluation scalar for the current round, computed as $\sum_i \beta^i \cdot z_i^{index}$.
$g^{(k)}$	`g`	The accumulated folded group generator for the current round.
$\mathbf{g}$	`self.s_vec_g`	The base vector of space-efficient group generators.

LISTING 5.6. Calculation of evaluations and generators

```
        let index = (c << k) + a;
        z += tmp * eval_points
                .iter().enumerate()
                .map(
                    |(i,ei)| (ei.pow([index as u64])) * (batch_chal.
                       pow([i as u64]))
                )
                .sum::<G::ScalarField>();

        g += self.s_vec_g[index] * tmp;
    }

    (z, g)
}
```

TABLE 6. Correspondence between theoretical notation and implementation variables for space-efficient `compute_final_z_g`

Math Symbol	Rust Variable	Description
$\log N$	`n_rounds`	The total number of folding rounds.
z	`eval_point`	The initial evaluation point for the polynomial.
$\{x_j\}_{j\in[\log N]}$	`ver_chals`	The sequence of verifier challenges accumulated during the protocol.
$z^{(j)}$	`current_eval`	The evaluation point "evolved" at the j-th round (squared at each step: z^{2^j}).
z_{final}	`z`	The final folded evaluation scalar, computed through the recursive relation $z^{(j+1)} = z^{(j)}(x_j^{-1} + x_j z^{2^j})$.
g_{final}	`g`	The final folded generator, computed as the weighted sum of all base generators.
$\mathbf{g}$	`self.s_vec_g`	The base vector of space-efficient group generators.
tmp	`tmp`	The product of verifier challenges (or their inverses) for a specific index ind.

This block implements the batched evaluation $\mathbf{Z} = \sum_{j=0}^{k-1} \beta^j \cdot \mathbf{Z_j}^{(k)}$. Each e_i^{index} is the contribution of the i-th point to the current folded index (z_i).

2.6. The `compute_final_z_g` Function. The purpose of the `compute_final_z_g` function is to find the final folded evaluation point (z_{final}) and the final combined generator (g_{final}) after all the rounds have been completed.

The first part of the function calculates the final evaluation of the polynomial (z_{final}). Mathematically, in each round j, the evaluation point z is updated based on the challenge x_j and its inverse x_j^{-1}.

LISTING 5.7. Final aggregation for multi-point verification

```
1     for j in 0..n_rounds {
2         let aj = ver_chals[j];
3         let aj_inv = aj.inverse().unwrap();
4         z *= aj_inv + (aj * current_eval);
5         current_eval = current_eval.square();
6     }
```

The values in this snippet are defined as follows:

- `z`: The final evaluation value z_{final}. It defines the recursive equation: $z_{j+1} = x_j^{-1}(z_j + x_j * z_j)$.
- `current_eval`: The current round's value of z^{2^j}. By squaring the value of z at every iteration, the code is able to represent an efficient simulation of the univariate evaluation vector $\mathbf{z} = (1, z, z^2, \cdots, z^{n-1})$.

The second part of the function performs the final generator. `compute_final_z_g` creates it performing a linear combination of the original generators g.

```
    for ind in 0..n_len {
        let mut tmp = G::ScalarField::from(1u64);
        for j in 0..n_rounds {
            let bit_j = (ind >> j) & 1;
            let aj = ver_chals[j];
            let aj_inv = aj.inverse().unwrap();

            if bit_j == 1 {
                tmp *= aj;
            } else {
                tmp *= aj_inv;
            }
        }
        g += self.s_vec_g[ind] * tmp;
    }

    (z, g)
}
```

The `tmp` variable is used for the reconstruction of the final weight of each generator and `index.s_vec_g` is the i-th original generator g_i.

2.7. The `compute_final_multi_z_g` Function. The `compute_final_multi_z_g` function is an extension of `compute_final_z_g`. It calculates the final folded evaluation point and the combined generator. This function handles two vectors: `current_eval` and `vec_z`. Each vector is composed of elements corresponding to each challenge point in `eval_points`.

```
1     for j in 0..n_rounds {
2         let aj = ver_chals[j];
3         let aj_inv = aj.inverse().unwrap();
4         vec_z = vec_z.iter().enumerate()
5                 .map(
6                     |(i,zi)| (*zi)*(aj_inv + (aj*current_evals[i]))
7                 )
8                 .collect();
9         current_evals = current_evals.iter()
10                        .map(
11                            |ei| (*ei).square()
12                        )
13                        .collect();
```

During each round j, it updates each point in parallel:

- Updates the value of z_i via the Bulletproofs folding equation.
- Squares each challenge e_i to prepare them for the next round.

Finally, the values obtained from each point are then aggregated in the linear combination. The calculation of the final generator g is performed identically to the single-point version.

2.8. Space-Efficient Polynomial Folding: The FoldedPolynomialTree. In the context of space-efficient in Gemini, the Prover has to avoid storing the intermediate polynomials. Given a polynomial $\phi^{(0)}(X)$ of degree d, Gemini creates a sequence of c polynomials $\phi^{(1)}, \ldots, \phi^{(c)}$, where $c = \lceil \log_2(d+1) \rceil$. Folding is performed using the following formula:

$$\phi_j^{(i)} = \phi_{2j}^{(i-1)} + \gamma_{i-1} \cdot \phi_{2j+1}^{(i-1)}. \tag{28}$$

TABLE 7. Correspondence between notation and implementation variables for `compute_final_multi_z_g`

Math Symbol	Rust Variable	Description
$\log N$	`n_rounds`	The total number of folding rounds.
$\{z_0, \ldots, z_{k-1}\}$	`eval_points`	The initial set of multiple evaluation points.
β	`batch_chal`	The batching challenge used for the final linear combination.
$\{x_j\}_{j \in [\log N]}$	`ver_chals`	The complete sequence of verifier challenges for all rounds.
$z_i^{(j)}$	`current_evals[i]`	The i-th evaluation point "evolved" at the j-th folding round (squared at each step).
z_{final}	`z`	The final batched evaluation scalar, computed as $\sum \beta^i \cdot v_i^{(\log N)}$.
g_{final}	`g`	The final folded generator, computed as the weighted sum of all base generators.
$\mathbf{g}$	`self.s_vec_g`	The base vector of space-efficient group generators.

In the time-efficient version of Gemini, the prover computes and stores every intermediate polynomial $\phi^{(1)}, \ldots, \phi^{(c)}$, commits to every one of them and performs a batch opening operation for the set of verifier's challenges $\{\gamma_0, \ldots, \gamma_{c-1}\}$. While, in the space-efficient implementation, we need to use a data structure called `FoldedPolynomialTree`. The `FoldedPolynomialTree` is an iterable object that takes the streaming coefficients of $\phi^{(0)}$ and the set of challenges $\mathcal{C}$ as input. Some important properties of the `FoldedPolynomialTree` iterator are:

- Every round of the iteration returns a pair (i, coeff) representing the current polynomial $\phi^{(i)}$ and its next coefficient respectively.
- The `FoldedPolynomialTree` allows the prover to compute all coefficients necessary for constructing the entire tree of folded polynomials $(\phi^{(1)}, \ldots, \phi^{(c)})$ by passing through the input stream of $\phi^{(0)}$ once.
- The `FoldedPolynomialTree` consumes only $O(\log_2 d)$ extra space when processing the recursion tree and therefore satisfies the space requirements specified in BP-Space.

In order to work with Gemini, a Polynomial Commitment Scheme will need to process coefficients that are received directly from the `FoldedPolynomialTree`, therefore we created special functions for our API (such as `commit_folding` and `open_folding`).

2.9. The `commit_folding` function. To integrate the streaming prover with the Gemini framework, we added the `commit_folding` function to our Bulletproofs implementation. This function uses an iterator of type `FoldedPolynomialTree`, to generate a vector of commitments for all the folded polynomials $\phi^{(1)}, \ldots, \phi^{(c)}$. Since the iterator produces the coefficients as (i, coeff) where i is the index of the polynomial, the function must track the progress of each polynomial independently. For this reason we initialize the following data structure:

- `coms`: a vector of size c that keeps the accumulation of the commitments for each polynomial.
- `curr_coeff`: the current coefficient index of polynomial $\phi^{(i)}$

```
1            for (i, coeff) in polynomials.iter() {
2            coms[i-1] += self.s_vec_g[curr_coeff[i-1]]*coeff;
3            curr_coeff[i-1] += 1;
4         }
```

When an iterator will produce a new coefficient for $\phi^{(i)}$, the function will retrieve the actual index j from the counter `curr_coeff[i]`. The commitment will be incremented as follows:

$$\text{coms}[i] \leftarrow \text{coms}[i] + (\text{coeff} \cdot \mathbf{g}_j) \tag{29}$$

By the end of the stream, the `coms` vector contains the final commitments for all c polynomials.

2.10. The `open_folding` function. The `open_folding` function is implemented to correctly batch open the polynomials $f^{(1)}, \ldots, f^{(c)}$ contained within the `FoldedPolynomialTree` iterator. It takes as input the Fiat-Shamir transcript, the `FoldedPolynomialTree` iterator *polynomials*, a vector of evaluation points *eval_points* (for multi-point opening), and a vector

η used to batch the polynomials $f^{(1)}, \ldots, f^{(c)}$ together. At a high-level, this function is a space-efficient implementation of the following open protocol:

Algorithm 4: High-Level Batch Opening Protocol

Input: Folded polynomials $\{f^{(1)}, \ldots, f^{(c)}\}$, evaluation points $\{\zeta_0, \ldots, \zeta_{k-1}\}$, batching challenges η, transcript challenges β

Output: Inner product y such that $\langle \mathbf{h}, \mathbf{z} \rangle = y$

`// 1. Define the combined batched polynomial`

$h(X) \leftarrow \sum_{i=1}^{c} \eta_{i-1} \cdot f^{(i)}(X)$;

`// 2. Define the multi-point evaluation tensor`

$d \leftarrow \deg(h)$;

for $j = 0$ **to** $k - 1$ **do**

  $\mathbf{z}_j \leftarrow (1, \zeta_j, \zeta_j^2, \ldots, \zeta_j^d)$;

$\mathbf{z} \leftarrow \sum_{j=0}^{k-1} \beta^j \cdot \mathbf{z}_j$;

`// 3. Compute the target evaluation value`

$y \leftarrow \langle \mathbf{h}, \mathbf{z} \rangle$;

return y;

As with `commit_folding`, the key idea is to take advantage of the homomorphic properties of the Bulletproofs commitments and inner products. The algorithm individually tracks the inner products $\langle f^{(i)}, \mathbf{z} \rangle$ and commitments $\langle f^{(i)}, \mathbf{g} \rangle$ in vectors, subsequently computing the combined inner product $\langle \mathbf{h}, \mathbf{z} \rangle$ and commitment $\langle \mathbf{h}, \mathbf{g} \rangle$ by summing the individual components. First, the algorithm calculates the recursion depth n as the base-two logarithm of the smallest power of 2 greater than *polynomials.len()* (the length of $f^{(0)}$). This value n represents the number of times the batched polynomial h must be folded. To manage the stream, a vector of length n called `max_strm_ind` is created, where `max_strm_ind`[i] stores the degree of $f^{(i+1)}$. Intuitively, this vector allows the prover to detect when the end of each stream $f^{(i+1)}$ has been reached, as the iterator does not explicitly signal the end of a polynomial's coefficients. The algorithm essentially executes the `open_multi_points` logic from

`bp_space` simultaneously for each polynomial $f^{(1)}, \ldots, f^{(c)}$. It iterates over $k = 0, \ldots, n-2$ and maintains several vectors of length n to track the state:

- `k_ev_coms` and `k_od_coms`: Track the "even" and "odd" commitments for each $f^{(i)}$.
- `ev_inner_prods` and `od_inner_prods`: Track the current even and odd inner products.
- `y_ks`: Tracks the current folded coefficient being computed at recursion depth k for each $f^{(i)}$.
- `processed_coeffs` and `stream_index`: Track how many coefficients have been processed and the current stream index for each $f^{(i)}$, respectively.

```
let mut k_ev_coms = vec![G::zero(); n];
let mut k_od_coms = vec![G::zero(); n];
let mut y_ks = vec![G::ScalarField::ZERO; n];
let mut processed_coeffs = vec![0; n];
let mut stream_index = vec![0; n];
let mut ev_inner_prods = vec![G::ScalarField::ZERO; n];
let mut od_inner_prods = vec![G::ScalarField::ZERO; n];
```

The prover iterates over the `FoldedPolynomialTree`. For each pair *(dep, curr_coeff)*, the algorithm proceeds as in `open_multi_points`, but adapted for interleaved streams:

- Parameters such as *stream_index*, y_k, and *processed_coeffs* are accessed using the depth index (*dep-1*).
- If *processed_coeffs[dep-1]* reaches 2^k, the algorithm proceeds with the standard `open_multi_points` logic, resetting the counters.

```
for (dep, curr_coeff) in polynomials.iter() {
let coeff = *curr_coeff.borrow();
let stream_ind = stream_index[dep-1];
let mut tmp = G::ScalarField::ONE;

for j in 0..k {
```

```
        let bit_j = (stream_ind >> j) & 1;
        let alpha_j = ver_chals[j];
        if bit_j == 0 { tmp *= alpha_j; }
        else { tmp *= alpha_j.inverse().unwrap(); }
    }
    y_ks[dep-1] += tmp * coeff * etas[dep-1];
```

- The "else if" statement checks if *stream_index[dep-1]* equals *max_strm_ind[dep-1] - 1*. Since folded polynomials are not guaranteed to be powers of two, the algorithm must detect the end of the stream and compute a *new_strm_ind* ($t \cdot 2^k - 1$) to correctly finalize the folding at that depth.

```
        else if stream_ind == max_strm_ind[dep-1]-1 {
    let mut new_strm_ind = 0;
    while new_strm_ind < stream_ind+1 {
        new_strm_ind += k_pow;
    }
    new_strm_ind -= 1;

    let bit_n_k = (new_strm_ind >> k) & 1;
    let (zk, gk) = self.compute_multi_z_g(k, new_strm_ind >>
       k, eval_points, &ver_chals, &beta);

    if bit_n_k == 0 {
        k_ev_coms[dep-1] += gk * y_ks[dep-1];
        ev_inner_prods[dep-1] += y_ks[dep-1]*zk;
    } else {
        k_od_coms[dep-1] += gk * y_ks[dep-1];
        od_inner_prods[dep-1] += y_ks[dep-1]*zk;
    }
    y_ks[dep-1] = G::ScalarField::ZERO;
```

```
19 }
```

- The *stream_index[dep-1]* is only incremented if the end of the stream has not been reached

Once the iteration over *polynomials* is complete, the prover compresses the results to compute the final folded inner products and commitments:

$$ev_{ip} = \sum_i \text{ev_inner_prods}[i] \tag{30}$$

$$od_{ip} = \sum_i \text{od_inner_prods}[i] \tag{31}$$

$$ev_{com} = (\sum_i \text{k_ev_coms}[i]) + \mathbf{g}_{ip} \cdot ev_{ip} \tag{32}$$

$$od_{com} = (\sum_i \text{k_od_coms}[i]) + \mathbf{g}_{ip} \cdot od_{ip} \tag{33}$$

```
1     let ev_ip: G::ScalarField = ev_inner_prods.iter().sum();
2     let od_ip: G::ScalarField = od_inner_prods.iter().sum();
3     let ev_com = k_ev_coms.iter().sum::<G>() + self.s_ip_gen*ev_ip;
4     let od_com = k_od_coms.iter().sum::<G>() + self.s_ip_gen*od_ip;
```

The process then continues identically to `open_multi_points` until the depth-$(n-1)$ recursion is completed. In the final step, the final folded value is computed as $y_{final} = \sum_i y_{finals}[i]$.

2.11. The `commit_folding_rev` function. The `commit_folding_rev` function has been designed to process a `FoldedPolynomialTree` iterator which streams coefficients from the highest degree to the lowest. The `commit_folding_rev` function takes as input an iterator for a `FoldedPolynomialTree`, which is assumed to be a reverse iterator that streams the coefficients in reverse order.

As with the `commit_folding` implementation, the number of commitments to be computed is n = polynomials.depth(). A vector `coms` is used to track these, where `coms[i]`

stores the commitment to the polynomial $f^{(i+1)}$. Similarly, a vector `current_coeff` is initialized to track the number of coefficients processed so far, effectively serving as a reverse stream index.

To map these coefficients, we compute another vector called `offsets` with the following property:

$$\text{offsets}[i] = d_{i+1} \tag{34}$$

where d_{i+1} is the degree of the polynomial $f^{(i+1)}$.

```
    let mut offsets = vec![0; n];
    offsets[0] = (polynomials.len() + 1) / 2;
    for i in 1..n {
        offsets[i] = (offsets[i-1] + 1) / 2;
}
```

Consequently, $\text{offsets}[i] - 1$ represents the final index of the coefficient stream, which is received first due to the reverse access.

When iterating over the *polynomials* as (i, coeff), the commitments are updated as follows:

$$\text{coms}[i-1] \leftarrow \text{coms}[i-1] + \mathbf{g}_{\text{index}} \cdot \text{coeff} \tag{35}$$

where the index is calculated as:

$$\text{index} = \text{offsets}[i-1] - \text{curr_coeff}[i-1] - 1. \tag{36}$$

It is important to notice that $\text{offsets}[i-1] - \text{curr_coeff}[i-1] - 1$ is the index of the coefficient *coeff* as if the stream were accessed in a standard forward order (from $f_0^{(i)}$ to $f_{d_i}^{(i)}$). This logic ensures that the function correctly computes the commitment to each $f^{(i)}$, obtaining the same result as a traditional low-to-high order streaming access.

```
            for (i, coeff) in polynomials.iter() {
            coms[i-1] += self.s_vec_g[offsets[i-1]-curr_coeff[i
               -1]-1]*coeff;
```

```
            curr_coeff[i-1] += 1;
        }
```

2.12. The `open_multi_folding_rev` function. The function accepts almost the same inputs as `open_folding`, with the addition of a separate polynomial stream, denoted as *base_polynomials*, which is unrelated to the `FoldedPolynomialTree`. The set of inputs includes:

- Fiat-Shamir Transcript: For maintaining the interactive state and obtaining challenges.
- *base_polynomials*: A polynomial stream P provided in reverse order (from high-order to low-order coefficients).
- *polynomials*: The `FoldedPolynomialTree` iterator in reverse order.
- *eval_points*: A vector of evaluation points for the multi-point opening proof.
- η (eta): A batching vector used to combine the polynomials $f^{(1)}, \ldots, f^{(c)}$ and P into a single opening argument.

In the first step, the function establishes the boundaries for each polynomial stream. The prover calculates the vector `max_strm_ind` of size $n+1$ where n is the base-two logarithm of the smallest power od two larger than *polynomials.len()*. The components in this vector are as follows:

- Index 0: We set `max_strm_ind[0]` equal to the length of *polynomials.len()*. This index is used to track *base_polynomials* throughout the algorithm.
- Indices 1 to n: Each `max_strm_ind[i]` stores the maximum stream index d_i of the corresponding folded polynomial $f^{(i)}$.

Then, the algorithm calculates the integer m, which represents the total number of recursion rounds. The calculation follows these conditions:

- If *base_polynomials.len*() $\leq$ *polynomials.len*(), the number of rounds m is set equal to the previously calculated n.

- Otherwise, m is defined as the base-two logarithm of the smallest power of two larger than $base_polynomials.len()$.

Formally, in the latter case, $m = \log_2(M)$, where $M = 2^k$ is the smallest power of two such that $base_polynomials.len() \leq M$ and $2^{k-1} < base_polynomials.len()$. The function follows the general structure of `open_folding` but incorporating the following changes:

- The prover executes a reverse version of the `open_multi_point` logic (from the `bp_space` framework), utilizing the previously vectors, such as `last_indices[0]`, `max_strm_ind[0]`, `k_ev_coms[0]`.
- The algorithm processes the `FoldedPolynomialTree` in reverse order. Each tuple $(\text{dep}, \text{coeff})$ in the tree is handled identically to the base polynomials, with the logic specialized for each depth (`dep`).
- At the end of the recursion, the Prover processes *base_polynomials* and the *Folded-PolynomialTree* to compute the final scalar y_{final}

Algorithm 5: Streaming Reverse Multi-Folding Opening (`open_multi_folding_rev`)

Input: Reverse streams P, $\{f^{(i)}\}$, eval points $\{\zeta_j\}$, challenges η

$n \leftarrow \lfloor \log_2(\text{smallest power of } 2 > \text{polynomials.len}()) \rfloor$
`max_strm_ind`$[0] \leftarrow P$.len() `max_strm_ind`$[i] \leftarrow \deg(f^{(i)})$ for $i = 1 \dots n$
if *P.len()* $\leq$ *polynomials.len()* **then**
    $m \leftarrow n$
else
    $m \leftarrow \lceil \log_2(P.\text{len}()) \rceil$
for $k \leftarrow 0$ **to** $m - 1$ **do**
    $k_{pow} \leftarrow 2^k$
    Initialize `k_ev_coms, k_od_coms, ev_inner_prods, od_inner_prods, y_ks, processed_coeffs, stream_index` of length $n + 1$
    Compute `last_indices`$[i] = t \cdot k_{pow}$ such that $t \cdot k_{pow} \geq$ `max_strm_ind`$[i]$
    foreach `coeff` *in stream* P **do**
        `offset_strm_ind` $\leftarrow$ `max_strm_ind`$[0]$ $-$ `stream_index`$[0] - 1$
        if `processed_coeffs`$[0] == k_{pow}$ **then**
            Update (g_k, z_k) as in forward `open_multi_point`
        else if `offset_strm_ind` == `last_indices`$[0]$ **then**
            `new_strm_ind` $\leftarrow$ `last_indices`$[0] + k_{pow} - 1$
            Compute (g_k, z_k) using `new_strm_ind` and reset $y_k[0]$, `processed_coeffs`$[0]$
    foreach (`dep`, `coeff`) *in tree stream* **do**
        `offset_strm_ind` $\leftarrow$ `max_strm_ind`$[dep]$ $-$ `stream_index`$[dep] - 1$
        if `processed_coeffs`$[dep] == k_{pow}$ **then**
            Update (g_k, z_k) using $offset_strm_ind$
        else if `offset_strm_ind` == `last_indices`$[dep]$ **then**
            `new_strm_ind` $\leftarrow$ `last_indices`$[dep] + k_{pow} - 1$
            Compute (g_k, z_k) using `new_strm_ind` and reset counters
    Compute $\mathtt{ev}_{ip}, \mathtt{od}_{ip}$ and $\mathtt{ev}_{com}, \mathtt{od}_{com}$ summarizing all $n + 1$ indices
    Update transcript and obtain challenge x_{k+1}
Compute y_{final}: `index` $\leftarrow$ `max_strm_ind`$[i]$ $-$ `stream_index`$[i] - 1$

CHAPTER 6

Experimental Results

In this chapter, both the time-optimized and space-optimized architecture of the Prover and Verifier are evaluated through an experimental approach. The main purpose of the benchmark is to analyze the trade-off inherent in different cryptographic assumptions and memory management methods. We evaluated the performance of the essential operations to the SNARK pipeline. Our architecture was tested over a range of commitment sizes, from 2^{12} to 2^{18} or 2^{20} constraints. The second goal is to perform a comparative analysis of Bulletproofs implementation and KZG-based systems. KZG-based systems are faster but rely on trusted setup assumpions, while the Bulletproofs implementation prioritizes a transparent setup. Therefore, our goal was to assess the possible performance advantages resulting from the trade-offs made when choosing a transparent setup over a trusted setup approach. A valuable feature of this research is the level of detail provided in the methodology used to collect data during our experiments. In particular, the original Gemini paper [**5**] *only* reports memory consumption during benchmarks, and does not provide a comprehensive time and space analysis and comparison. We improve upon Gemini by making these comparisons.

1. Experimental Setup

We established a benchmarking environment to measure execution time and memory dynamics. In order to provide an accurate and repeatable assessment of the execution time performance of our implementations, we utilized the Criterion library, a benchmarking tool written in Rust. This choice ensures that our measurements of execution time are consistent with those from the original Gemini methodology. To determine the amount of memory consumed during the execution of our algorithms, we implemented a custom memory tracking system, consisting of a script, detailed in Listing 6.1, that reads the contents of

`/proc/{pid}/statm` and tracks the stack and heap usage of the benchmark process. Then, we incorporated the memory tracking process into the Rust benchmarking suite through the following methods:

- We used the `std::process::Command` module within Rust to launch the monitoring script for each benchmark.
- We invoked the script with three parameters: the PID of the benchmark process, and two strings to specify the directory and filename.
- After the benchmark is complete, we ensure that the monitoring script is terminated.

Although memory tracking via `/proc/pid/statm` may have some inaccuracies, we employed this method to remain consistent with the original Gemini benchmarking methodology. All experiments were conducted on a system equipped with an AMD Ryzen 7 9800X3D and 32 GB of DDR5 RAM (6000 MT/s), running CachyOS with Linux kernel 6.19.13. The codebase was compiled using `rustc` and `cargo` version `1.85.0-nightly`.

LISTING 6.1. Bash script for measuring memory consumption.

```
1  #!/bin/bash
2  pname="/proc/$1"
3  fname="./$2_$3_memory.txt"
4  proc="$pname/statm"
5  while true; do
6      getcat="$(cat␣$proc)"
7      catarr=($getcat)
8      #echo ${catarr[5]}
9      printf "${catarr[5]}\n" >> $fname
10     sleep 3
11 done
```

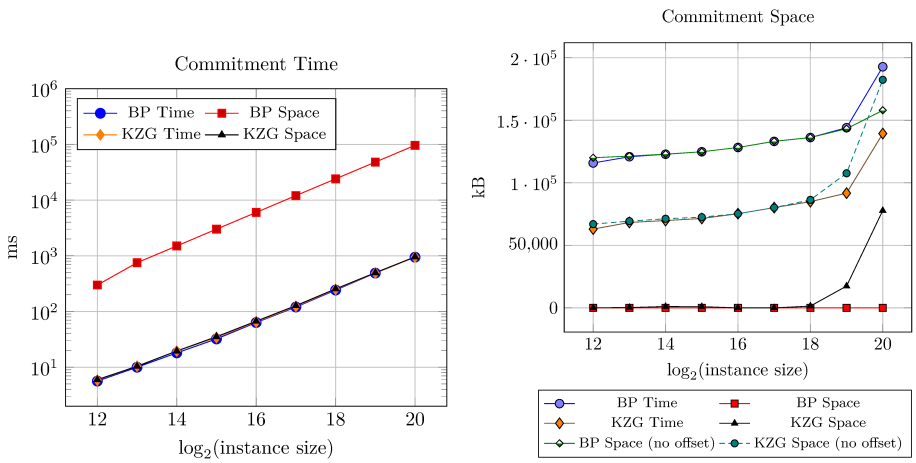
2. Testing Methodology

To provide a high granularity evaluation, the benchmarking suite was organized into discrete modules which measured the performance of specific operations:

- {pcs}_commit_bench: Measures the time and memory it itakes to perform the first commit.
- {pcs}_open_bench: Evaluates the performance of generating a proof for a single point opening.
- {pcs}_open_multi_bench: Evaluates the performance of generating a proof for multiple point openings.
- {pcs}_snark_bench: Evaluates the entire process of generating and verifying a SNARK proof.

Each of these modules were created for both Bulletproofs (bp) and KZG (kzg), allowing for a comprehensive comparison between the two polynomial commitment schemes. As previously described, the Criterion library is used to aggregate and calculate the execution time benchmarks, while memory tracking is performed using the custom script that reads from /proc/{pid}/statm. To accurately quantify the *additional* space used by the space-efficient implementations, we adopted the following procedure to calculate an offset value:

- Each benchmark requires an initial setup, which is completed before the actual benchmarking begins.
- While space-efficient implementations are theoretically designed for streaming or random access to these objects, in our current benchmark and implementation, these objects reside within the scope of the Rust stack and heap.
- To isolate the space-efficiency of the PCS implementations, the program performs the initial setup and then enters a 10-second sleep period. During this time, we measure the amount of memory allocated by Rust up to that point.
- We then subtract this offset from the actual memory consumption reported during the execution of the algorithm. This allows us to obtain an estimate of the additional memory used specifically by our space-efficient algorithms.



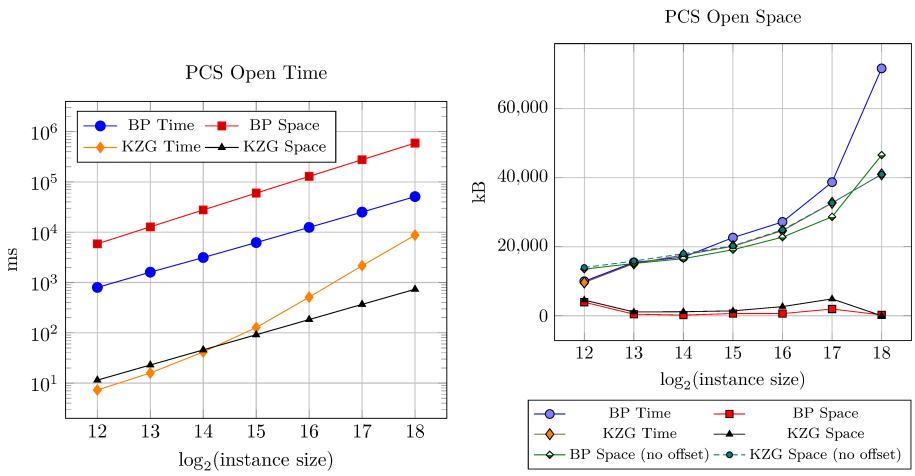
(A) Execution time (B) Memory usage

FIGURE 1. Execution time and memory usage for the commitment phase.

Remark: Due to the inaccuracies of reading from `/proc/{pid}/statm` and possible optimizations from the Rust compiler and run-time environment, the reported memory values obtained during the execution of the space-efficient PCS algorithm were *less* than the recorded memory offset. In such cases, we simply take the offset value to be the minimum of the due measurements to obtain our estimates.

3. Micro-benchmark Results

3.1. Evaluation of Commitment Phase. The results for the commitment phase, shown in Figure 1, illustrate the execution times and memory usage across various sizes from 2^{12} to 2^{20}. Figure 1a reports the execution time. All implementations demonstrate linear scalability $O(n)$. However, there is an important difference in performance among the architectures. Specifically, the KZG implementations (both Time and Space) and the BP Time version are nearly indistinguishable, taking around 10^3 ms to complete a 2^{20} commitment. In contrast, the BP Space prover represents an outlier, showing two orders of magnitude higher latency. This is because Gemini uses a space-efficient MSM algorithm for KZG, that is not yet implemented for our Space-efficient Bulletproofs. It is expected that the addition of this MSM algorithm to the BP Space Prover will result in a significant reduction in the commitment time, bringing it in line with the other implementations. After

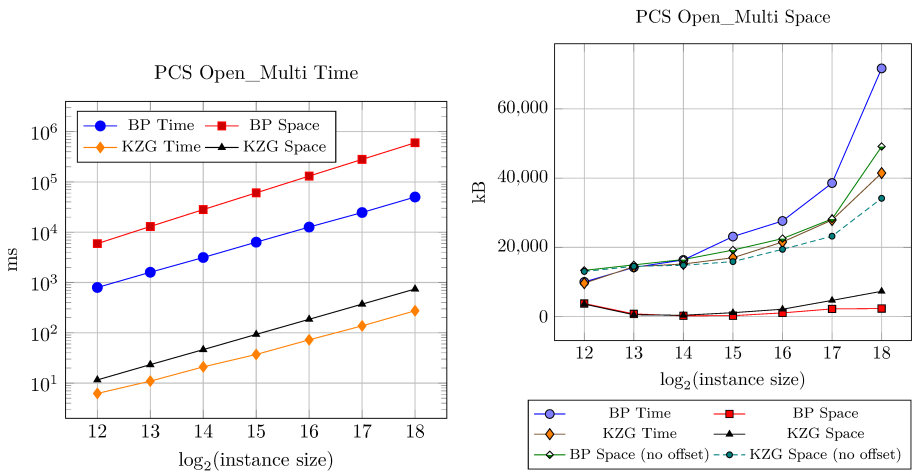


(A) Execution time (B) Memory usage

FIGURE 2. Execution time and memory usage for the opening phase.

evaluating the execution time, we now move to the analysis of memory consumption for the commitment phase. Figure 1b shows the memory usage during the commitment phase. BP Time and KZG Time show a large increase in the amount of memory consumed as the instance size increases. These data confirm that provers optimized for speed are unsuitable for operating within memory-limited constraints. On the other hand, BP Space and KZG Space show a much more stable memory usage. As values reach 2^{20}, the memory consumption for BP Space remains constant while KZG Space begins to increase.

3.2. Evaluation of PCS Open. Figure 2 reports the execution time and memory usage of the PCS opening algorithms. The graph in Figure 2a shows the execution time for the `open` function using different instance sizes ranging between 2^{12} and 2^{18}. These results show significant differences in execution times among architectures. As expected, the KZG-based configurations were the fastest. BP Time is approximately one order of magnitude slower than all the KZG implementations, while BP Space is clearly the worst performing configuration: its latency is one to two orders of magnitude higher than its time-efficient counterpart. To complement the time analysis, Figure 2b shows memory usage during the opening phase. The results are consistent with the trends we saw during the commitment phase, validating the efficiency of our space-optimized approach. Interesting result is that BP Space always demonstrates a lower memory usage than KZG Space.



(A) Execution time (B) Memory usage

FIGURE 3. Execution time and memory usage for the multi-opening phase.

3.3. Evaluation of PCS Open Multi-Points. Figure 3 shows the execution time and memory usage for the multi-opening phase. Figure 3a reports the execution time. The reported results confirm the performance trends established by the prior benchmarks. KZG Time is faster than all other implementations, while BP Time is ten times slower than its KZG counterparts. Figure 3b shows the memory usage during the multi-opening phase. Similar to the single opening phase, both BP Time and KZG Time show an increasing memory usage as the instance size increases. In contrast, both BP Space and KZG Space provide constant memory efficiency, as shown by the data collected. Furthermore, BP Space demonstrates stability and flatness even during the testing of the largest instance sizes.

3.4. Keygen Time and the "Trusted Setup" Problem. Figure 4 shows the time required to generate the key. Although KZG may appear to be faster, it is important to interpret this result taking into consideration the fact that KZG relies on a trusted setup. In our research, in order to implement KZG, the needed parameters were generated locally. However, in a real-world scenario, the Trusted Setup can take a significant amount of time (up to a year, see `https://notes.ethereum.org/@CarlBeek/kzg_ceremony_timelines`). On the other hand, Bulletproofs does not need a trusted setup, so the total time required to generate the key is negligible.

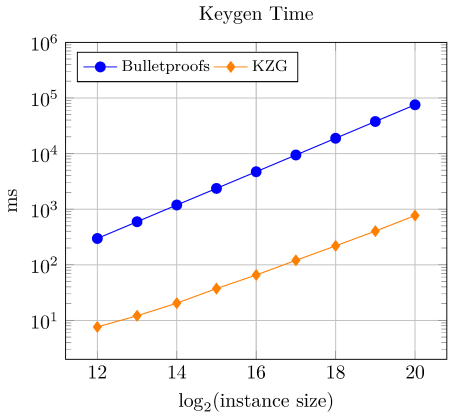
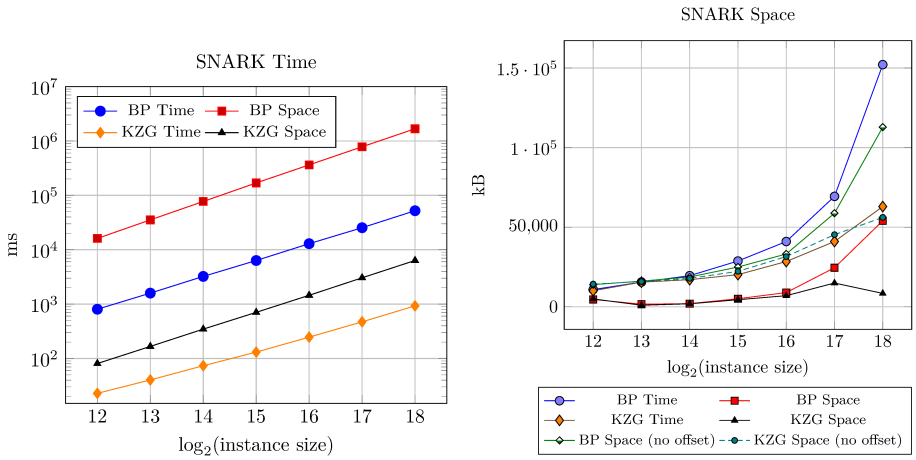


FIGURE 4. Execution time for the key generation phase.



(A) Execution time (B) Memory usage

FIGURE 5. Execution time and memory usage for the complete SNARK proof generation.

4. Macro-benchmark Results

4.1. Evaluation of SNARK Performance. Figure 5 shows the total end-to-end execution time and peak memory usage for the complete SNARK proof generation. Figure 5a reports the execution time. The benchmarks were performed on instance sizes ranging from 2^{12} to 2^{18} constraints. The data clearly indicate that the computational efficiency is ordered as follows: KZG Time is the fastest, followed by KZG Space; BP Time is significantly slower than all KZG implementations, and finally, BP Space is the slowest configuration.

Figure 5b shows the memory usage during the complete SNARK proof generation. BP Time consumes more memory than all the other implementations, while KZG Time follows

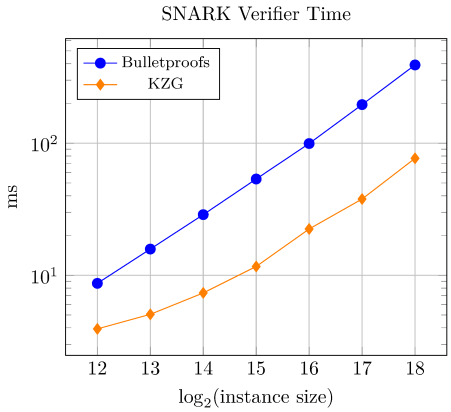


FIGURE 6. Verification time for the complete SNARK proof generation.

a similar but slower curve. This confirms that without specific memory optimizations, it will be impossible to generate a full SNARK for large circuits on hardware with limited amount of RAM. Both space-efficient implementations successfully decouple memory growth from the instance size much more effectively than their time-efficient equivalents.

4.2. SNARK Verifier Time. Figure 6 shows the execution time of the SNARK verifier. Consistent with their expected complexity ($O(1)$ for KZG and $O(N)$ for Bulletproofs), the plot demonstrates that KZG verification is very fast, while BP verification time is much slower than KZG. It is important to note that this thesis does not focus on the measure of the space-efficiency of the verifier, for three main reasons:

- The optimization of the verifier is out of scope of this project.
- In the case of KZG, the verifier is already trivially space-efficient due to its nature.
- The BP verifier can be made space-efficient using the same ideas and methodology as the BP prover.

5. Conclusions

The experimental results presented in this chapter confirm that it is possible to decouple the memory requirements of a Bulletproofs-based SNARK from the instance size, however at a significant increase in computation time. BP Space uses the same amount of memory

to process the data even if the instance sizes reach 2^{20}. While KZG remains the best-performing architecture overall, the Bulletproofs architecture has a significant benefit: no trusted setup is required. In summary, if execution speed and constant-time proof verification are important factors, then KZG should be the preferred choice. However, the space-efficient Bulletproofs prover developed in this thesis provides an alternative for applications that require a transparent setup and operate with limited memory resources.

CHAPTER 7

Conclusions and Future Work

1. Conclusions

This reasearch was focused on the problem of memory scalability in Zero-Knowledge Proof systems and how to improve the trade-off between time and memory usage in the Gemini framework. The contributions of this thesis are:

- **Architectural modularization**: We designed a generalized Gemini API to overcome the shortcomings of the initial implementation. This abstraction allows us to integrate various commitment schemes. In particular, we identified the *Folded-PolynomialTree* structure and implementation details that made a straightforward implementation of Bulletproofs with in Gemini impossible.
- **Implementation of two different architectures for the Prover, one time-optimized and one space-optimized**: We designed and tested two distinct implementations based on the even-odd approach: `bp_time`, which is optimized for time-performance by utilizing available RAM, and `bp_space`, which uses streaming. We also implemented new functions such as `compute_multi_z_g` and `open_multi_points`, that enable the prover to batch open *multiple* evaluation proofs for a single polynomial. Moreover, we extended the specification of Block et al. to support univariate polynomials, and directly implemented this extension.
- **Experimental validation**: Our results indicate that although the time-complexity of Bulletproofs-based approaches (requiring considerable execution time for large statements) is higher than KZG-based approaches, the memory usage is similar to the KZG-based approaches. Moreover, Bulletproofs implementation has the significant advantage of being transparent, not requiring a trusted setup.

In summary, this thesis presents an extensive analysis of the costs and benefits of trustless architectures. The generalization of the API is used to allow developers to decide what is the most optimal balance between computation time and memory usage based on their specific hardware requirements.

2. Future work

The research presented in this thesis provides a foundation for future work in the area of space-efficient verifiable computation:

- **Automated Elasticity**: One possible area for future study will be developing an "elastic" prover architecture. The elastic prover would operate with a hybrid unit allowing to dynamically transition between streaming efficiently using limited memory resources (time consuming but efficient) and the full memory based method (memory intensive but fast).
- **Broadening Commitment Schemes**: Due to the generalized API developed in this thesis, Gemini could be used with other PCS. Future work could explore the integration of other space-efficient PCS, such as the DARK protocol [**6**], which operates over integer groups of unknown order.
- **Designing New Space-Efficient Variants**: An important research could be creating space-efficient versions of popular PCSs, such as Brakedown [**7**], FRI [**8**] and Dory [**9**].

APPENDIX A

Software Repository

The source code developed for this thesis, including the implementations of the `bp_time` and `bp_space` provers, the generalized Gemini API, and the benchmarking suite, is publicly available on GitHub.

`https://github.com/fraPoli28/minimal-overhead-zk-arguments`

APPENDIX B

Benchmark Data Tables

This appendix provides the raw data used to generate the plots discussed in Chapter 6. The values represent the measurements for execution time (in milliseconds) and memory consumption (in kilobytes) across the different architectures: KZG Time, KZG Space, BP Time, and BP Space.

TABLE 1. BP Commit Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	56.31357605550887	115839	
	13	99.86424357023811	120858	
	14	180.6897096345238	122831	
	15	321.59706312248684	124812	
	16	633.5126956521825	128241	
	17	1202.6107343015873	133128	
	18	2423.793038333333	136182	
	19	4885.005110	144038	
	20	9449.117933	192778	
space	12	2983.7176885	120030	120030
	13	7510.63677000	121374	121374
	14	15026.0612690	122915	122915
	15	30048.1690090	124812	124812
	16	60096.2705050	128241	128241
	17	120187.770964	133173	133173
	18	240331.476033	136210	136210
	19	480703.805469	143222	143222
	20	961427.149685	157961	157961

TABLE 2. BP Open Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	7977.473920	10000	
	13	16044.585603	15469	
	14	31372.835940	16988	
	15	62250.123416	22637	
	16	124697.729982	27184	
	17	251148.350894	38692	
	18	510487.294201	71653	
space	12	58738.077628	13517	9580
	13	127970.960782	15155	14693
	14	277219.296328	16543	16348
	15	599208.416219	19133	18477
	16	1289514.134504	22770	23408
	17	2763899.573105	28672	30624
	18	5912733.881626	46539	46795

TABLE 3. BP Open_Multi Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	7965.543544	10000	
	13	15984.224163	14203	
	14	31420.167563	16353	
	15	63430.752320	23114	
	16	126591.234140	27615	
	17	246303.477635	38574	
	18	500831.847903	71700	
space	12	59388.722469	13347	9580
	13	130037.349279	14933	14179
	14	281866.535743	16477	16257
	15	606398.219913	19206	18954
	16	1306837.607471	22527	23583
	17	2803673.225685	28201	30409
	18	6000632.078051	49121	46817

TABLE 4. BP SNARK Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	8042.758118	10825	
	13	15914.405135	15522	
	14	32275.006753	19560	
	15	63313.970204	28768	
	16	129000.722749	41002	
	17	253735.655866	69369	
	18	520681.055299	152121	
space	12	161151.599680	13927	9308
	13	353468.577594	16130	14486
	14	770786.656771	18723	16736
	15	1689156.587054	24997	19917
	16	3629289.591320	33172	24243
	17	7810933.321068	58777	34242
	18	16744312.089282	112825	58777

TABLE 5. KZG Commit Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	58.5372	62978	
	13	104.28635	68293	
	14	195.5441	69811	
	15	341.59159	71484	
	16	651.68984	75279	
	17	1237.29866	80147	
	18	2500.90968	84928	
	19	4908.58215	91724	
	20	9477.54971	139363	
space	12	59.75965	66968	66923
	13	103.95242	69363	69091
	14	195.17869	71249	70193
	15	352.73388	72535	71575
	16	669.61418	75279	75279
	17	1274.93504	80147	80147
	18	2543.64348	86339	84928
	19	4961.65976	107617	90196
	20	9569.31642	182374	104546

TABLE 6. KZG Open Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	72.82871	9644	
	13	158.95165	15161	
	14	415.58461	17582	
	15	1272.05863	20143	
	16	5104.47565	24671	
	17	21565.04877	32688	
	18	87338.45973	40963	
space	12	114.58199	13974	9436
	13	228.90609	15852	14745
	14	457.92951	17987	16814
	15	914.68696	20255	18831
	16	1833.49520	24895	22239
	17	3674.93672	32712	27824
	18	7336.2608	40963	40963

TABLE 7. KZG Open_Multi Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	62.27345	9642	
	13	109.47776	14410	
	14	210.14815	15185	
	15	371.00142	16990	
	16	724.80577	21540	
	17	1373.08679	27893	
	18	2742.21006	41464	
space	12	115.61964	13028	9434
	13	232.95883	14506	13994
	14	463.13044	14814	15185
	15	929.63676	15876	16990
	16	1855.94675	19427	21540
	17	3715.7223	23224	27893
	18	7423.47941	34168	41464

TABLE 8. KZG SNARK Data

	$\log_2(N)$	time (ms)	space (kB)	space offset (kB)
time	12	228.54446	10097	
	13	403.09235	15472	
	14	735.20306	17047	
	15	1309.4017	20201	
	16	2468.71051	28433	
	17	4715.27024	41019	
	18	9285.92345	62959	
space	12	807.78077	14269	9308
	13	1666.36598	15864	14864
	14	3465.24117	17847	15958
	15	7049.79988	22246	17794
	16	14515.41887	31549	24561
	17	30432.1844	45319	30382
	18	63402.52836	56175	47748

TABLE 9. SNARK Verifier Data

	$\log_2(N)$	time (ms)
Bulletproofs	12	86.85425
	13	157.87951
	14	287.73180
	15	535.54741
	16	993.65018
	17	1959.08832
	18	3910.63956
KZG	12	39.27353
	13	50.60858
	14	73.48571
	15	116.5449
	16	224.0485
	17	377.79232
	18	768.11397

TABLE 10. Keygen Data

	$\log_2(N)$	time (ms)
Bulletproofs	12	297.1468
	13	594.26927
	14	1188.38466
	15	2362.62465
	16	4710.36267
	17	9422.91102
	18	18847.81140
	19	37696.78145
	20	75382.6868
KZG	12	7.63107
	13	12.09355
	14	20.38470
	15	37.34409
	16	65.50977
	17	120.06745
	18	218.72257
	19	402.48847
	20	768.31947

Bibliography

[1] Block, A.R., Holmgren, J., Rosen, A., Rothblum, R.D., and Soni, P. *Public-Coin Zero-Knowledge Arguments with (almost) Minimal Time and Space Overheads*. In: Pass, R., Pietrzak, K. (eds.) Theory of Cryptography - TCC 2020, Proceedings, Part II, Lecture Notes in Computer Science, vol. 12551, pp. 168–197. Springer, 2020.

[2] Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., and Ward, N.P. *Aurora: Transparent Succinct Arguments for R1CS*. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2019, Proceedings, Part I, Lecture Notes in Computer Science, vol. 11476, pp. 103–128. Springer, 2019.

[3] Fiat, A., and Shamir, A. *How to Prove Yourself: Practical Solutions to Identification and Signature Problems*. In: Odlyzko, A.M. (ed.) Advances in Cryptology - CRYPTO '86, Proceedings, Lecture Notes in Computer Science, vol. 263, pp. 186–194. Springer, 1986.

[4] Kate, A., Zaverucha, G.M., and Goldberg, I. *Constant-Size Commitments to Polynomials and Their Applications*. In: Abe, M. (ed.) Advances in Cryptology - ASIACRYPT 2010, Proceedings, Lecture Notes in Computer Science, vol. 6477, pp. 177–194. Springer, 2010.

[5] Bootle, J., Chiesa, A., Hu, Y., and Orrù, M. *Gemini: Elastic SNARKs for Diverse Environments*. In: Dunkelman, O., and Dziembowski, S. (eds.) Advances in Cryptology - EUROCRYPT 2022, Proceedings, Lecture Notes in Computer Science, vol. 13276, pp. 427–457. Springer, 2022.

[6] Block, A. R., Holmgren, J., Rosen, A., Rothblum, R. D., and Soni, P. *Time- and Space-Efficient Arguments from Groups of Unknown Order*. In: Malkin, T., and Peikert, C. (eds.) Advances in Cryptology - CRYPTO 2021, Proceedings, Lecture Notes in Computer Science, vol. 12828, pp. 123–152. Springer, 2021.

[7] Golovnev, A., Lee, J., Setty, S. T. V., Thaler, J., and Wahby, R. S. *Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS*. In: Handschuh, H., and Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023, Proceedings, Lecture Notes in Computer Science, vol. 14082, pp. 193–226. Springer, 2023.

[8] Ben-Sasson, E., Bentov, I., Horesh, Y., and Riabzev, M. *Fast Reed-Solomon Interactive Oracle Proofs of Proximity*. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., and Sannella, D. (eds.) 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018), LIPIcs, vol. 107, pp. 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[9] Lee, J. *Dory: Efficient, Transparent Arguments for Generalised Inner Products and Polynomial Commitments*. In: Nissim, K., and Waters, B. (eds.) Theory of Cryptography - TCC 2021, Proceedings, Lecture Notes in Computer Science, vol. 13043, pp. 1–34. Springer, 2021.

[10] arkworks contributors. *`arkworks` zkSNARK ecosystem*. `https://arkworks.rs`, 2022. [Online; accessed 04/20/2026].

[11] The Rust Team. *The Rust programming language*. `https://rust-lang.org`, 2020. [Online; accessed 04/20/2026].

[12] Bootle, J., Cerulli, A., Chaidos, P. Groth, J. and Petit, C. *Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting*. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Heidelberg, May 2016.

[13] Bünz, B., Bootle, J., Boneh, D., Polestra, A., Wuille, P., Maxwell, G. *Bulletproofs: Short proofs for confidential transactions and more*. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[14] Bootle, J., Chiesa, A., Hu, Y., and Orrù, M. *Gemini reference implementation*. `https://github.com/arkworks-rs/gemini`, 2022.[Online; accessed 04/20/2026].

[15] Goldwasser, S., Micali, S., and Rackoff, C. *The knowledge complexity of interactive proof systems. SIAM J. Comput.*, 18(1):186–208, 1989.

[16] Ben-Sasson, E., Chiesa, A., and Spooner, N. *Interactive Oracle Proofs*. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, October/November 2016.

[17] Reingold, O., Rothblum, G. N., and Rothblum, R. D. *Constant-round interactive proofs for delegating computation*. In Daniel Wichs and Yishay Mansour, editors, *48th ACM STOC*, pages 49–62. ACM Press, June 2016.

[18] de Valence, H. *Merlin: composable proof transcripts for public-coin arguments of knowledge*. `https://docs.rs/crate/merlin/latest`, 2021. [Online; accessed 04/20/2026].

[19] The Rayon contributors. *Rayon*. `https://docs.rs/rayon/latest/rayon/`, 2016. [Online; accessed 04/20/2026].

Vita

Francesca Stella

Graduate Student in Cybersecurity and Computer Science
Politecnico di Torino — University of Illinois Chicago (UIC)

Education

- **M.S. in Computer Science** 2024 –
  University of Illinois Chicago, USA.
- **Laurea Magistrale in Cybersecurity** 2024 –
  Politecnico di Torino, Italy.
- **B.S. in Computer Engineering (Ingegneria Informatica)** 2021 – 2024
  Politecnico di Torino, Italy.
  Final Grade: 105/110.
- **High School Diploma** 2016 – 2021
  Liceo Scientifico A. Pacinotti, La Spezia, Italy.
  Final Score: 100/100.

Academic Experience

- **Teaching Assistant (Collaborazione alla didattica)** [Spring 2025]
  Politecnico di Torino, Department of Control and Computer Engineering.
  Selected for a paid assistantship position to support students during lab sessions and exercise hours for undergraduate engineering courses.

Languages

- **Italian:** Native speaker.
- **English:** Proficient (IELTS Academic score: 6.5, obtained in 2024).