Advances in Zero-Knowledge Proofs: Bridging the Gap between Theory and Practice

Tiancheng Xie

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-35
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-35.html

May 1, 2024

Advances in Zero-Knowledge Proofs: Bridging the Gap between Theory and Practice

By

Tiancheng Xie

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Associate Professor Alessandro Chiesa
Associate Professor Nikhil Srivastava

Spring 2023

Advances in Zero-Knowledge Proofs: Bridging the Gap between Theory and Practice

Abstract

Advances in Zero-Knowledge Proofs: Bridging the Gap between Theory and Practice

By

Tiancheng Xie

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

This dissertation presents a series of novel zero-knowledge proof (ZKP) protocols—Libra, deVirgo, Orion, and Pianist—each achieving significant improvements in proof generation speed. Zero-knowledge proofs are critical cryptographic tools, enabling secure and privacy-preserving transactions without revealing sensitive information. However, their practical adoption is hindered by the inefficiency of existing proof generation methods.

Our research proposes four distinct protocols, each making substantial contributions to the efficiency of ZKP generation. We begin with the Libra protocol, which introduces a more efficient proof construction method compared to state-of-the-art ZKPs at that time. Next, we present the deVirgo protocol, which builds upon Libra's design and further optimizes proof generation by leveraging the power of parallelization.

The third protocol, Orion, takes a different approach, resulting in significant improvements in proof generation speed. We detail the innovative design and methodology of Orion, highlighting its unique features and performance gains.

Finally, we introduce the Pianist protocol, which employs parallel computation strategies to achieve remarkable improvements in proof generation speed and is compatible with existing popular protocol Plonk. Pianist builds upon the foundation established by Plonk, while incorporating novel techniques to enhance performance.

The dissertation includes a comprehensive comparative analysis of the four proposed protocols, evaluating their scalability, security, and practicality. In conclusion, our research contributes to the field of cryptography by providing a series of innovative ZKP protocols that significantly enhance proof generation speed, paving the way for more widespread adoption of privacy-preserving technologies.

To my family, friends, mentors, and colleagues.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Dawn Song, for her unwavering support, guidance, and mentorship throughout my Ph.D. journey. Her profound expertise, insightful feedback, and constant encouragement have been invaluable to my research and personal growth. Prof. Song's dedication to her students and commitment to fostering a nurturing academic environment have truly inspired me, and I am incredibly fortunate to have had the opportunity to learn from such an outstanding advisor.

In addition, I would like to extend my heartfelt appreciation to my Postdoc mentor, Dr. Yupeng Zhang, currently an Assistant Professor at Texas A&M University. Dr. Zhang has been instrumental in my academic development, providing invaluable guidance and expertise as he introduced me to the field. His enthusiasm for research, deep understanding of the subject matter, and unwavering commitment to my success have been truly inspiring. I am immensely grateful for his mentorship and the countless hours he has dedicated to helping me grow both academically and professionally.

Furthermore, I would like to extend my sincere gratitude to Prof. Elaine Shi from Carnegie Mellon University for her invaluable assistance during the early stages of my research. Prof. Shi's collaboration and insights have greatly contributed to the development of my work, and her guidance has been instrumental in shaping my research direction. We have had the pleasure of co-authoring two papers, and I am truly grateful for the knowledge and expertise she has shared with me throughout our collaboration.

I would also like to acknowledge and thank the distinguished group of professors who have made significant contributions as co-authors in various stages of my research. I am grateful for the opportunity to work with Prof. Peihan Miao, Dr. Saikrishna Badrinarayanan, Prof. Muthu Venkitasubramaniam, Prof. Carmit Hazay, Prof. Charalampos (Babis) Papamanthou, Prof. Fan Zhang, and Prof. Dan Boneh. Each of these esteemed scholars has provided invaluable insights, expertise, and encouragement, greatly enriching my work and enhancing my understanding of the field. Collaborating with such a talented and dedicated group of researchers has been an immense privilege, and I deeply appreciate their contributions to my academic growth.

My student co-authors: Jiaheng Zhang, Tianyi Liu, Zhiyong Fang, Rishabh Bhadauria, and Weikai Lin, are talented and dedicated individuals. Their passion for research, hard work, and innovative ideas have not only enriched our collaborative projects but also contributed to my personal growth as a researcher. I am grateful for the opportunity to learn from and work with such a remarkable group of students, and I look forward to witnessing their future achievements in the field.

Moreover, I am truly fortunate to have been surrounded by an incredible group of friends throughout my academic journey, who have provided both intellectual and emotional support. I would like to extend a special mention to my dear friend, co-author, and roommate, Jiaheng Zhang. His unwavering support, constant encouragement, and dedication to our shared passion for research have made a lasting impact on my life. Jiaheng's camaraderie, companionship, and brilliant mind have enriched my experiences during our time together.

I would like to express my heartfelt appreciation to the following friends who have shared countless memorable moments with me, whether it was playing games, enjoying dinner, or simply engaging in thoughtful conversations, Bicheng Gao, Wanxin Cai, Duxing Hao, Yiwei Bai, Enze Zhang, Yue Qiu, Zhipeng Cai, Guangxin Lyu, Fukang She, Yile Dai, Ruyang Sun, Linjie Zhou, Youning Chen, Xiaojun Xu, Huichen Li, Xuan Zhang, Haoming Lu, Yunqi Li, Lun Wang, Sijun Tan, Zhe Ye, Yanpei Liu, Lianmin Zheng, Ying Sheng, Weikeng Chen, Xiaoyuan Liu, Zecong Hu, Lunjia Hu, and Cheng Wan. A special acknowledgment goes to Lianmin Zheng and Ying Sheng, who have not only been my roommates but also my pillars of support during this journey.

The laughter, camaraderie, and encouragement from this remarkable group of friends have greatly contributed to my well-being and success, and I will cherish these friendships for a lifetime.

Finally, I would like to express my deepest love and gratitude to my family, who have been my bedrock of support throughout my life. To my cousin, Xuan Wan, I am grateful for the encouragement and camaraderie we have shared. Your presence in my life has been a source of comfort and inspiration.

Most importantly, I would like to extend my heartfelt appreciation to my parents, whose importance in my life cannot be overstated. My father, Wenping Xie, and my mother, Liu Yang, have always believed in me, encouraged me, and provided unwavering love and care. Their sacrifices, guidance, and understanding have been instrumental in shaping who I am today, and their constant presence has been a source of strength and inspiration throughout my academic journey. I am eternally grateful for their love and support, and I dedicate this work to them as a testament to their unwavering belief in my abilities.

Chapter 1

Introduction

The rapid advancement of technology has led to a growing need for secure and efficient cryptographic protocols, particularly in the realm of privacy-preserving computation. Zero-knowledge proofs (ZKPs) have emerged as a powerful cryptographic primitive, enabling one party to prove the validity of a statement without revealing any additional information about the underlying data. However, a major challenge in the practical deployment of ZKPs is the efficiency of proof generation, as it has a direct impact on the overall performance and scalability of privacy-preserving applications.

This thesis focuses on the development and optimization of novel techniques to significantly improve the proof generation speed of zero-knowledge proofs. We propose a series of innovative protocols, divided into two parts: the first part, consisting of Libra and Orion, aims to achieve optimal prover complexity, while the second part, comprising deVirgo and Pianist, focuses on enabling distributed provers. These protocols leverage cutting-edge cryptographic techniques, advanced optimizations, and novel constructions to enhance the performance and scalability of ZKPs in real-world applications.

In the first part of this thesis, we introduce Libra, a protocol that addresses the existing limitations of proof generation speed by employing an optimal GKR protocol and introducing several optimizations. We then present Orion, which provides an optimal polynomial commitment, offering substantial performance improvements over previous protocols and combined with Libra, we can achieve optimal prover complexity.

In the second part, we describe deVirgo, a protocol that builds upon the advancements made by Libra and Orion, and focuses on enabling distributed provers. deVirgo further improves proof generation speed by incorporating a novel recursive composition technique. Lastly, we introduce Pianist, a groundbreaking protocol that leverages parallelization techniques to facilitate distributed provers, setting a new benchmark in the field of zero-knowledge proofs.

Throughout this thesis, we provide rigorous security analysis and comprehensive performance evaluations for each proposed protocol, demonstrating their practicality and effectiveness in various privacy-preserving applications. By significantly improving the proof generation speed of zero-knowledge proofs and enabling distributed provers, our work not only advances the state-of-the-art in cryptographic research but also paves the way for more widespread adoption of privacy-preserving technologies across diverse domains.

1.1 Achieving Optimal Prover time

Previous protocols require a prover running quasi-linear time in the statement size, and each statement needs a separate trusted setup. They are both time-consuming. The first part of this thesis focuses on the development and optimization of Libra[XZZPS19a] and Orion[XZS22], two innovative zero-knowledge proof systems that aim to achieve optimal prover complexity. These protocols build upon cutting-edge cryptographic techniques and advanced optimizations to improve the efficiency of proof generation in various privacy-preserving applications. We also developed libraries of these protocols and released codes online [Libb; Vira; Virb] for developers.

1.1.1 Libra

Libra is built upon a new linear-time algorithm for the prover of the interactive proof protocol by Goldwasser, Kalai, and Rothblum[GKR15], also known as the GKR protocol. Additionally, Libra employs an efficient approach to convert the GKR protocol into a zero-knowledge proof using small masking polynomials. Notably, Libra features a one-time trusted setup that depends only on the size of the input to the circuit and not on the circuit logic.

Not only does Libra boast excellent asymptotic performance, but it is also highly efficient in practice. For instance, our implementation demonstrates that it takes only 200 seconds to generate a proof for constructing a SHA2-based Merkle tree root on 256 leaves, outperforming all existing zero-knowledge proof systems. Both the proof size and verification time of Libra are highly competitive, making it a significant advancement in the field of zero-knowledge proofs.

1.1.2 Orion

In this subsection, we introduce Orion, a groundbreaking zero-knowledge argument system that achieves $O(N)$ prover time of field operations and hash functions, and $O(\log^2 N)$ proof size. Orion stands out for its concretely efficient performance, addressing the high overhead on proof generation time that has limited the efficiency and scalability of existing schemes with succinct proof size.

Our implementation of Orion demonstrates remarkable performance, with a prover time of 3.09 seconds and a proof size of 1.5MB for a circuit with 2^{20} multiplication gates. Notably, the prover time is the fastest among all existing succinct proof systems, and the proof size is an order of magnitude smaller than a recent scheme proposed by Golovnev et al. in 2021 [GLSTW].

Orion's efficiency improvements can be attributed to two novel techniques. Firstly, we propose a new algorithm for testing whether a random bipartite graph is a lossless expander graph based on the densest subgraph algorithm. This approach allows us to sample lossless expander with overwhelming probability, improving the efficiency and/or security of all existing zero-knowledge argument schemes with linear prover time. The testing algorithm based on the densest subgraph may also be of independent interest for other applications of expander graphs.

Secondly, we develop an efficient proof composition scheme called code switching, which reduces the proof size from square root to polylogarithmic in the size of the computation. This scheme is built on the encoding circuit of a linear code and demonstrates that the witness of a second zero-knowledge argument is the same as the message in the linear code. The proof composition introduces only a small overhead on the prover time, further enhancing the performance of Orion.

By combining Orion with Libra, a fully optimal prover in terms of complexity can be achieved, significantly advancing the field of zero-knowledge proofs.

1.2 Distributed Proving

The second part of this thesis presents another line of work focused on parallelizing and distributing zero-knowledge proof systems. In this direction, we introduce deVirgo[Xie+22], a novel protocol that builds upon the Libra and Virgo protocols to achieve parallel and distributed proof generation. This approach improves the scalability and efficiency of zero-knowledge proofs, expanding their applicability in various privacy-preserving applications.

deVirgo takes advantage of the strengths of both Libra and Virgo. By parallelizing the prover and enabling distributed proof generation, deVirgo tackles the challenges posed by large-scale computations and high prover complexity, which have traditionally limited the practicality of zero-knowledge proof systems.

The development of deVirgo represents a significant advancement in the field of zero-knowledge proofs, enhancing their performance and scalability.

Another significant protocol is Pianist, a significant protocol built on the Plonk proof system, leverages key features and optimizations such as bi-variate KZG[KZG] commitments and Lagrange-based techniques. These innovations make the distributed proving system more scalable and efficient.

The bi-variate KZG commitments with Lagrange-based techniques aim at reducing the size of communication between machines. As a result, Pianist achieves a significantly reduced communication overhead of just several kilobytes.

1.2.1 deVirgo

In this subsection, we delve into the details of deVirgo, a distributed SNARK protocol designed for data-parallel circuits. Data-parallel circuits, as mentioned earlier, consist of multiple identical sub-circuits with no connections between them. This characteristic allows each sub-circuit to be processed independently, providing an opportunity to accelerate proof generation by handling them in parallel.

deVirgo is built upon the Virgo protocol for two main reasons: firstly, Virgo does not require a trusted setup and is plausibly post-quantum secure; secondly, Virgo is among the fastest protocols with succinct verification time and proof size for large-scale problems. deVirgo extends Virgo to handle data-parallel arithmetic circuits, achieving optimal scalability without any overhead on proof size. The protocol is specifically designed to process data-parallel circuits with N copies using N parallel machines, resulting in N times faster performance than the original Virgo while maintaining the same proof size.

The deVirgo protocol operates with a master node and several ordinary nodes, with the master node responsible for aggregating messages and proofs from distributed machines. This setup eliminates the need for a linear increase in proof size, which would occur if each sub-circuit generated its proof separately. The protocol is composed of two primary building blocks: the GKR protocol and the polynomial commitment (PC) scheme.

The GKR protocol consists of d sumcheck protocols for a circuit of depth d, and in the distributed sumcheck protocol of deVirgo, the master node aggregates messages from all machines in every round. This approach maintains the same proof size as the original sumcheck protocol, saving a factor N over the naive distributed protocol.

In the distributed PC protocol, the commitment phase is optimized, allowing the master node to aggregate N commitments into one, rather than sending N commitments directly to the verifier. During the opening phase, the proof can also be aggregated, improving the proof size by a logarithmic factor in the size of the polynomial.

By combining these techniques, deVirgo offers a powerful and efficient solution for processing data-parallel circuits, paving the way for more scalable and practical zero-knowledge proof systems.

1.2.2 Pianist

In this subsection, we introduce Pianist, a fully distributed zero-knowledge proof (ZKP) system designed to enhance the scalability of blockchain technologies such as zkRollups and zkEVM. One of the primary challenges in deploying blockchains is the limited throughput of transactions. Pianist addresses this issue by distributing the ZKP generation process across multiple machines, significantly reducing the burden on individual machines and improving overall efficiency.

Pianist is built upon Plonk, a highly efficient ZKP system with a universal trusted setup. The proposed distributed ZKP scheme enables proof generation to be distributed across multiple participants in a mining pool-like model. Pianist's first protocol is tailored for data-parallel circuits, offering a prover time complexity of $O(T \log T + M \log M)$ per machine when using M machines to process M sub-circuits of size T each. This is in contrast to the $O(MT \log MT)$ prover time complexity of the original Plonk on a single machine.

The protocol ensures minimal communication among machines, with $O(1)$ communication per machine. The proof size and verifier time complexity are also $O(1)$, which is the same as the original Plonk. Furthermore, Pianist's second protocol, with minor modifications, can support general circuits with arbitrary connections while maintaining the same proving, verifying, and communication complexities.

The Pianist system, when implemented, can generate a proof for 8,192 transactions in just 313 seconds using 64 machines. This represents a 64x improvement in scalability compared to the original Plonk scheme. The communication per machine is a mere 2.1 KB, independent of the number of machines and circuit size. The proof size is 2.2 KB, and the verifier time is 3.5 ms. Pianist also shows similar improvements for general circuits; for example, it takes only 5 seconds to generate a proof for a randomly generated circuit with 2^25 gates using 32 machines, which is 24.2 times faster than Plonk on a single machine.

By leveraging Pianist's distributed ZKP schemes, blockchain technologies can achieve higher scalability and efficiency, enabling broader adoption in various applications.

1.2.3 Cross-chain bridges for blockchains

In this subsection, we introduce zkBridge, a highly efficient and secure cross-chain bridge designed for the increasingly diverse blockchain ecosystem. As various blockchains coexist, cross-chain communication becomes a crucial building block, and zkBridge aims to address this need.

Existing cross-chain bridge solutions often face performance issues or rely on trust assumptions, which can result in compromised security. With over 1.5 billion USD lost in attacks against bridges, there is a pressing need for a more secure solution. zkBridge addresses this issue by providing strong security without external trust assumptions and using succinct proofs to guarantee correctness while reducing on-chain verification costs.

zkBridge takes advantage of the distributed proof systems deVirgo (or use the alternative Pianist), which offer significant performance improvements over existing solutions. By utilizing these proof systems, zkBridge can achieve dramatically faster proof generation speed, enabling secure and efficient cross-chain communication.

Moreover, the modular design of zkBridge allows it to support a wide range of use cases and capabilities, such as message passing, token transferring, and computational logic operating on state changes across different chains. To demonstrate the practicality of zkBridge, a prototype bridge from Cosmos to Ethereum was implemented, showcasing its ability to handle large proof circuits that other systems cannot efficiently manage. The evaluation of this prototype revealed that zkBridge achieves practical performance, with proof generation taking less than 20 seconds and on-chain verification costing less than 230K gas. Additionally, the implementation and evaluation of the direction from Ethereum to other EVM-compatible chains, like BSC, further proved zkBridge's versatility and efficiency.

Overall, zkBridge serves as an innovative and highly effective solution for cross-chain communication in the multi-chain ecosystem, leveraging the power of distributed proof systems like deVirgo to ensure secure, scalable, and efficient performance.

Chapter 2

Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation

We present Libra, the first zero-knowledge proof system that has both optimal prover time and succinct proof size/verification time. In particular, if C is the size of the circuit being proved (i) the prover time is $O(C)$ irrespective of the circuit type; (ii) the proof size and verification time are both $O(d \log C)$ for d-depth log-space uniform circuits (such as RAM programs). In addition Libra features an one-time trusted setup that depends only on the size of the input to the circuit and not on the circuit logic. Underlying Libra is a new linear-time algorithm for the prover of the interactive proof protocol by Goldwasser, Kalai and Rothblum (also known as GKR protocol), as well as an efficient approach to turn the GKR protocol to zero-knowledge using small masking polynomials. Not only does Libra have excellent asymptotics, but it is also efficient in practice. For example, our implementation shows that it takes 200 seconds to generate a proof for constructing a SHA2-based Merkle tree root on 256 leaves, outperforming all existing zero-knowledge proof systems. Proof size and verification time of Libra are also competitive.

This work was previously published in [XZZPS19b].

2.1 Introduction

Zero-knowledge proofs (ZKP) are cryptographic protocols between two parties, a *prover* and a *verifier*, in which the prover can convince the verifier about the validity of a statement without leaking any extra information beyond the fact that the statement is true. Since they were first introduced by Goldwasser et al. [GMR89], ZKP protocols have evolved from pure theoretical constructs to practical implementations, achieving proof sizes of just hundreds of bytes and verification times of several milliseconds, regardless of the size of the statement being proved. Due to this successful transition to practice, ZKP protocols have found numerous applications not only in the traditional computation delegation setting but most importantly in providing privacy of transactions in deployed cryptocurrencies (e.g., Zcash [Ben+14]) as well as in other blockchain research projects (e.g., Hawk [KMSWP]).

Despite such progress in practical implementations, ZKP protocols are still notoriously hard to scale for large statements, due to a particularly high overhead on generating the proof. For most systems, this is primarily because the prover has to perform a large number of cryptographic operations, such as exponentiation in an elliptic curve group. And to make things worse the asymptotic complexity of computing the proof is typically more than linear, e.g., $O(C \log C)$ or even $O(C \log^2 C)$, where C is the size of the statement.

Unfortunately, as of today we are yet to construct a ZKP system whose prover time is *optimal*, i.e., linear in the size of the statement C (this is irrespective of whether the ZKP system has per-statement trusted setup, one-time trusted setup or no trusted setup at all). The only notable exception is the recent work by Bünz et al. [BBBPWM18] that however suffers from linear verification time—for a detailed comparison see Table 2.1. Therefore designing ZKP systems that enjoy linear prover time as well as succinct[1] proof size and verification time is an open problem, whose resolution can have significant practical implications.

Our contributions. In this paper we propose Libra, the first ZKP protocol with *linear prover time* and *succinct proof size and verification time* in the size of the arithmetic circuit representing the statement C, when the circuit is *log-space uniform*. Libra is based on the doubly efficient interactive proof protocol proposed by Goldwasser et al. in [GKR15] (referred as GKR protocol in this paper), and the verifiable polynomial delegation scheme proposed by Zhang et al. in [ZGKPP17c]. As such it comes with *one-time trusted* setup (and not per-statement trusted setup) that depends only on the size of the input (witness) to the statement that is being proved. Not only does Libra have excellent asymptotic performance but also its prover outperforms in practice all other ZKP systems while verification time and proof size are also very competitive—see Table 2.1. Our concrete contributions are:

- **GKR with linear prover time.** Libra features a new linear-time algorithm to generate a GKR proof. Our new algorithm does not require any pattern in the circuit and our result subsumes all existing improvements on the GKR prover assuming special circuit structures, such as regular circuits in [Tha13a], data-parallel circuits in [Tha13a; Wah+17], circuits with different sub-copies in [ZGKPP18]. See related work for more details.

[1] In ZKP literature, "succinct" is poly-logarithmic in the size of the statement C.

- **Adding zero-knowledge.** We propose an approach to turn Libra into zero-knowledge efficiently. In particular, we show a way to mask the responses of our linear-time prover with small random polynomials such that the zero-knowledge variant of the protocol introduces minimal overhead on the verification time compared to the original (unmasked) construction.

- **Implementation and evaluation.** We implement Libra. Our implementation takes an arithmetic circuit with various types of gates (fan-in 2 and degree ≤ 2, such as $+, -, \times$, AND, XOR, etc.) and compiles it into a ZKP protocol. We conduct thorough comparisons to all existing ZKP systems (see Section 2.1.1). We plan to release our system as an open-source implementation.

2.1.1 Comparing to other ZKP Systems

Table 2.1 shows a detailed comparison between Libra and existing ZKP systems. First of all, Libra is the best among all existing systems in terms of practical prover time. In terms of asymptotics, Libra is the only system with linear prover time and succinct verification and proof size for log-space uniform circuits. The only other system with linear prover time is Bulletproofs [BBBPWM18] whose verification time is linear, *even for log-space uniform circuits*. In the practical front, Bulletproofs prover time and verification time are high, due to the large number of cryptographic operations required for every gate of the circuit.

The proof and verification of Libra are also competitive to other systems. In asymptotic terms, our proof size is only larger than libSNARK [BSCTV14c] and Bulletproofs [BBBPWM18], and our verification is slower than libSNARK [BSCTV14c] and libSTARK [BSBHR19]. Compared to Hyrax [WTSTW18], which is also based on similar techniques with our work, Libra improves the performance in all aspects (yet Hyrax does not have any trusted setup). One can refer to Section 2.5 for a detailed description of our experimental setting as well as a more detailed comparison.

Finally, among all systems, libSNARK [BSCTV14c] requires a trusted setup for every statement, and Libra requires an one-time trusted setup that depends on the input size. See Section 2.5.3 for a discussion on removing trusted setup in Libra.

Log-space uniform circuits. Though the prover time in Libra is optimal for all circuits, the verification time is succinct only when the circuit is structured (log-space uniform with logarithmic depth). This is the best that can be achieved for all ZKP protocols without per-circuit setup, as the verifier must read the entire circuit, which takes linear time in the worst case. We always refer to log-space uniform circuits when we say our scheme is succinct in this paper, to differentiate from schemes with linear verification time on all circuits (irrespective of whether the circuits are log-space uniform or not). Schemes such as libSTARK [BSBHR19], zkVSQL [ZGKPP17a] and Hyrax [WTSTW18] also have such property.

In practice, with the help of auxiliary input and circuit squashing, most computations can be expressed as log-space uniform circuits with low depth, such as matrix multiplication, image scaling and Merkle hash tree in Section 2.5. Asymptotically, as shown in [BSCTV14c; ZGKPP18; BSBHR19], all random memory access (RAM) programs can be validated by circuits that are log-space uniform with log-depth in the running time of the programs (but linear in the size of the programs) by RAM-to-circuit reduction, which justifies the expressiveness of such circuits.

Table 2.1: Comparison of Libra to existing ZKP systems, where $(\mathcal{G}, \mathcal{P}, \mathcal{V}, |\pi|)$ denote the trusted setup algorithm, the prover algorithm, the verification algorithm and the proof size respectively. Also, C is the size of the log-space uniform circuit with depth d, and n is the size of its input. The numbers are for a circuit computing the root of a Merkle tree with 256 leaves (511 instances of SHA256).[2]

	libSNARK [BSCTV14c]	Ligero [AHIV17]	Hyrax [WTSTW18]	libSTARK [BSBHR19]	Aurora [BSCRSVW19]	Libra
$\mathcal{G}$	$O(C)$ per-statement trusted setup	no trusted setup				$O(n)$ one-time trusted setup
$\mathcal{P}$	$O(C \log C)$	$O(C \log C)$	$O(C \log C)$	$O(C \log^2 C)$	$O(C \log C)$	$O(C)$
$\mathcal{V}$	$O(1)$	$O(C)$	$O(\sqrt{n} + d \log C)$	$O(\log^2 C)$	$O(C)$	$O(d \log C)$
$\|\pi\|$	$O(1)$	$O(\sqrt{C})$	$O(\sqrt{n} + d \log C)$	$O(\log^2 C)$	$O(\log^2 C)$	$O(d \log C)$
$\mathcal{G}$	1027s	NA				210s
$\mathcal{P}$	360s	400s	1,041s	2,022s	3199s	201s
$\mathcal{V}$	0.002s	4s	9.9s	0.044s	15.2s	0.71s
$\|\pi\|$	0.13KB	1,500KB	185KB	395KB	174.3KB	51KB

2.1.2 Our Techniques

Our main technical contributions are a GKR protocol with linear prover time and an efficient approach to turn the GKR protocol into zero-knowledge. We summarize the key ideas behind these two contributions. The detailed protocols are presented in Section 2.3 and 2.4 respectively.

GKR with linear prover. Goldwasser et al. [GKR15] showed an approach to model the evaluation of a layered circuit as a sequence of summations on polynomials defined by values in consecutive layers of the circuit. Using the famous sumcheck protocol (see Section 2.2.3.1), they developed a protocol (the GKR protocol) allowing the verifier to validate the circuit evaluation in logarithmic time with a logarithmic size proof. However, the polynomials in the protocol are multivariate with $2s$ variables, where S is the number of gates in one layer of the circuit and $s = \log S$. Naively running the sumcheck protocol on these polynomials incurs S^2 prover time, as there are at least $2^{2s} = S^2$ monomials in a $2s$-variate polynomial. Later, Cormode et al. [CMT12] observed that these polynomials are sparse, containing only S nonzero monomials and improved the prover time to $S \log S$.

In our new approach, we divide the protocol into two separate sumchecks. In each sumcheck, the polynomial only contains s variables, and can be expressed as the product of two multilinear polynomials. Utilizing the sparsity of the circuit, we develop new algorithms to scan through each gate of the circuit and compute the closed-form of all these multilinear polynomials explicitly, which takes $O(S)$ time. With this new way of representation, the prover can deploy a dynamic program-

[1] STARK is in the RAM model. To compare the performance, we convert a circuit of size C to a RAM program with $T = \Theta(C)$ steps.

ming technique to generate the proofs in each sumcheck in $O(S)$ time, resulting in a total prover time of $O(S)$.

Efficient zero-knowledge GKR. The original GKR protocol is not zero-knowledge, since the messages in the proof can be viewed as weighed sums of the values in the circuit and leak information. In [ZGKPP17a; WTSTW18], the authors proposed to turn the GKR protocol into zero-knowledge by hiding the messages in homomorphic commitments, which incurs a big overhead in the verification time. In [CFS17], Chiesa et al. proposed an alternative approach by masking the protocol with random polynomials. However, the masking polynomials are as big as the original ones and the prover time becomes exponential, making the approach mainly of theoretical interest.

In our scheme, we first show that in order to make the sumcheck protocol zero-knowledge, the prover can mask it with a "small" polynomial. In particular, the masking polynomial only contains logarithmically many random coefficients. The intuition is that though the original polynomial has $O(2^\ell)$ or more terms (ℓ is the number of variables in the polynomial), the prover only sends $O(\ell)$ messages in the sumcheck protocol. Therefore, it suffices to mask the original polynomial with a random one with $O(\ell)$ coefficients to achieve zero-knowledge. In particular, we set the masking polynomial as the sum of ℓ univariate random polynomials with the same variable-degree. In Section 2.4.1, we show that the entropy of this mask exactly counters the leakage of the sumcheck, proving that it is sufficient and optimal.

Besides the sumcheck, the GKR protocol additionally leaks two evaluations of the polynomial defined by values in each layer of the circuit. To make these evaluations zero-knowledge, we mask the polynomial by a special low-degree random polynomial. In particular, we show that after the mask, the verifier in total learns 4 messages related to the evaluations of the masking polynomial and we can prove zero-knowledge by making these messages linearly independent. Therefore, the masking polynomial is of constant size: it consists of 2 variables with variable degree 2.

2.1.3 Related Work

In recent years there has been significant progress in efficient ZKP protocols and systems. In this section, we discuss related work in this area, with the focus on those with sublinear proofs.

QAP-based. Following earlier work of Ishai [IKO], Groth [Gro10] and Lipmaa [Lip12], Gennaro et al. [GGPR13] introduced quadratic arithmetic programs (QAPs), which forms the basis of most recent implementations [PHGR13; BSCGTV13; BFRSBW13; BSCTV14a; Cos+15; WSRBW15; FFGKOP16] including libSNARK [BSCTV14c]. The proof size in these systems is constant, and the verification time depends only on the input size. Both these properties are particularly appealing and have led to real-world deployments, e.g., ZCash [Ben+14]. One of the main bottlenecks, however, of QAP-based systems is the high overhead in the prover running time and memory consumption, making it hard to scale to large statements. In addition, a separate trusted setup for every different statement is required.

IOPs. Based on "(MPC)-in-the-head" introduced in [IKOS07; GMO16; Cha+17], [AHIV17] proposed a ZKP scheme called Ligero. It only uses symmetric key operations and the prover time is fast in practice. However, it generates proofs of size $O(\sqrt{C})$, which is several megabytes in practice for moderate-size circuits. In addition, the verification time is quasi-linear to the size of the circuit. It is categorized as interactive PCP, which is a special case of interactive oracle proofs (IOPs). IOP generalizes the probabilistically checkable proofs (PCPs) where earlier works of Kilian [Kil92] and Micali [Mic00] are built on. In the IOP model, Ben-Sasson et al. built libstark [BSBHR19], a zero-knowledge transparent argument of knowledge (zkSTARK).libstark does not rely on trusted setup and executes in the RAM model of computation. Their verification time is only linear to the description of the RAM program, and succinct (logarithmic) in the time required for program execution. Recently, Ben-Sasson et al. [BSCRSVW19] proposed Aurora, a new ZKP system in the IOP model with the proof size of $O(\log^2 C)$.

Discrete log. Before Bulletproof [BBBPWM18], earlier discrete-log based ZKP schemes include the work of Groth [Gro09], Bayer and Groth [BG12] and Bootle et al. [BCCGP16]. The proof size of these schemes are larger than Bulletproof either asymptotically or concretely.

Hash-based. Bootle et al. [BCGGHJ17] proposed a ZKP scheme with linear prover time and verification time. The verification only requires $O(C)$ field additions. However, the proof size is $O(\sqrt{C})$ and the constants are large as mentioned in the paper [BCGGHJ17].

Interactive proofs. The line of work that relates to our paper the most is based on interactive proofs [GMR89]. In the seminal work of [GKR15], Goldwasser et al. proposed an efficient interactive proof for layered arithmetic circuits. Later, Cormode et al. [CMT12] improved the prover complexity of the interactive proof in [GKR15] to $O(C \log C)$ using multilinear extensions instead of low degree extensions. Several follow-up works further reduce the prover time assuming special structures of the circuit. For regular circuits where the wiring pattern can be described in constant space and time, Thaler [Tha13a] introduced a protocol with $O(C)$ prover time; for data parallel circuits with many copies of small circuits with size C', a $O(C \log C')$ protocol is presented in the same work, later improved to $O(C + C' \log C)$ by Wahby et al. in [Wah+17]; for circuits with many non-connected but different copies, Zhang et al. showed a protocol with $O(C \log C')$ prover time.

In [ZGKPP17c], Zhang et al. extended the GKR protocol to an argument system using a protocol for verifiable polynomial delegation. Zhang et al. [ZGKPP18] and Wahby et al. [WTSTW18] make the argument system zero-knowledge by putting all the messages in the proof into homomorphic commitments, as proposed by Cramer and Damgard in [CD98]. This approach introduces a high overhead on the verification time compared to the plain argument system without zero-knowledge, as each addition becomes a multiplication and each multiplication becomes an exponentiation in the homomorphic commitments. The multiplicative overhead is around two orders of magnitude in practice. Additionally, the scheme of [WTSTW18], Hyrax, removes the trusted setup of the argument system by introducing a new polynomial delegation, increasing the proof size and verification time to $O(\sqrt{n})$ where n is the input size of the circuit.

Lattice-based. Recently Baum et al. [BBCDPGL18] proposed the first lattice-based ZKP system with sub-linear proof size. The proof size is $O(\sqrt{C \log^3 C})$, and the practical performance is to be explored.

2.2 Preliminaries

2.2.1 Notation

In this paper, we use λ to denote the security parameter, and $\mathsf{negl}(\lambda)$ to denote the negligible function in λ. "PPT" stands for probabilistic polynomial time. We use $f(), h()$ for polynomials, x, y, z for vectors of variables and g, u, v for vectors of values. x_i denotes the i-th variable in x. We use bold letters such as **A** to represent arrays. For a multivariate polynomial f, its "variable-degree" is the maximum degree of f in any of its variables.

Bilinear pairings. Let $\mathbb{G}, \mathbb{G}_T$ be two groups of prime order p and let $g \in \mathbb{G}$ be a generator. $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ denotes a bilinear map and we use $\mathsf{bp} = (p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^\lambda)$ for the generation of parameters for the bilinear map. Our scheme relies on the q-Strong Bilinear Diffie-Hellman (q-SBDH) assumption and an extended version of the Power Knowledge of Exponent (PKE) assumption.

Assumption 1 (q-Strong Bilinear Diffie-Hellman)**.** *For any PPT adversary $\mathcal{A}$, the following holds:*

$$\Pr\left[\begin{array}{ll}\mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda) & \\ s \xleftarrow{R} \mathbb{Z}_p^* & : (x, e(g,g)^{\frac{1}{s+x}}) \leftarrow \mathcal{A}(1^\lambda, \sigma) \\ \sigma = (\mathsf{bp}, g^s, ..., g^{s^q}) & \end{array}\right] \leq \mathsf{negl}(\lambda)$$

The second assumption is a generalization of the q-PKE assumption [Gro10] to multivariate polynomials, proposed in [ZGKPP17c; ZGKPP17a]. Let $\mathcal{W}_{\ell,d}$ be the set of all multisets of $\{1, ..., \ell\}$ with the cardinality of each element being at most d.

Assumption 2 ((d, ℓ)-Extended Power Knowledge of Exponent)**.** *For any PPT adversary $\mathcal{A}$, there is a polynomial time algorithm $\mathcal{E}$ (takes the same randomness of $\mathcal{A}$ as input) such that for all benign auxiliary inputs $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$ the following probability is negligible:*

$$\Pr\left[\begin{array}{rr}\mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda) & \\ s_1, ..., s_\ell, s_{\ell+1}, \alpha \xleftarrow{R} Z_p^*, s_0 = 1 & \\ \sigma_1 = (\{g^{\prod_{i \in W} s_i}\}_{W \in \mathcal{W}_{\ell,d}, g^{s_{\ell+1}}}) & e(h, g^\alpha) = e(\tilde{h}, g) \\ \sigma_2 = (\{g^{\alpha \prod_{i \in W s_i}}\}_{W \in \mathcal{W}_{\ell,d}}, g^{\alpha s_{\ell+1}}) & : \prod_{W \in \mathcal{W}_{\ell,d}} g^{a_W \prod_{i \in W} s_i} g^{b s_{\ell+1}} \neq h \\ \sigma = (\mathsf{bp}, \sigma_1, \sigma_2, g^\alpha) & \\ \mathbb{G} \times \mathbb{G} \ni (h, \tilde{h}) \leftarrow \mathcal{A}\left(1^\lambda, \sigma, z\right) & \\ \left(a_0, \ldots, a_{|\mathcal{W}_{\ell,d}|}, b\right) \leftarrow \mathcal{E}\left(1^\lambda, \sigma, z\right) & \end{array}\right] \leq \mathsf{negl}(\lambda)$$

2.2.2 Interactive Proofs and Zero-knowledge Arguments

Interactive proofs. An interactive proof allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ the validity of some statement. The interactive proof runs in several rounds, allowing $\mathcal{V}$ to ask questions in each round based on $\mathcal{P}$'s answers of previous rounds. We phrase this in terms of $\mathcal{P}$ trying to convince $\mathcal{V}$ that $f(x) = 1$. The proof system is interesting only when the running time of $\mathcal{V}$ is less than the time of directly computing the function f. We formalize interactive proofs in the following:

Definition 2.2.1. *Let f be a Boolean function. A pair of interactive machines $\langle \mathcal{P}, \mathcal{V} \rangle$ is an interactive proof for f with soundness ϵ if the following holds:*

- **Completeness.** *For every x such that $f(x) = 1$ it holds that* $\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x) = \mathit{accept}] = 1$.
- **ϵ-Soundness.** *For any x with $f(x) \neq 1$ and any $\mathcal{P}^*$ it holds that* $\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = \mathit{accept}] \leq \epsilon$

Zero-knowledge arguments. An argument system for an NP relationship R is a protocol between a computationally-bounded prover $\mathcal{P}$ and a verifier $\mathcal{V}$. At the end of the protocol, $\mathcal{V}$ is convinced by $\mathcal{P}$ that there exists a witness w such that $(x; w) \in R$ for some input x. We focus on arguments of knowledge which have the stronger property that if the prover convinces the verifier of the statement validity, then the prover must know w. We use $\mathcal{G}$ to represent the generation phase of the public key pk and the verification key vk. Formally, consider the definition below, where we assume R is known to $\mathcal{P}$ and $\mathcal{V}$.

Definition 2.2.2. *Let R be an NP relation. A tuple of algorithm $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a zero-knowledge argument of knowledge for R if the following holds.*

- ***Correctness**. For every $(\mathsf{pk}, \mathsf{vk})$ output by $\mathcal{G}(1^\lambda)$ and $(x, w) \in R$,*

$$\langle \mathcal{P}(\mathsf{pk}, w), \mathcal{V}(\mathsf{vk}) \rangle(x) = \mathsf{accept}$$

- ***Soundness**. For any PPT prover $\mathcal{P}$, there exists a PPT extractor ε such that for every $(\mathsf{pk}, \mathsf{vk})$ output by $\mathcal{G}(1^\lambda)$ and any x, it holds that*

$$\Pr[\langle \mathcal{P}(\mathsf{pk}), \mathcal{V}(\mathsf{vk}) \rangle(x) = \mathsf{accept} \wedge (x, w) \notin R | w \leftarrow \varepsilon(\mathsf{pk}, x)] \leq \mathit{negl}(\lambda)$$

- ***Zero knowledge**. There exists a PPT simulator $\mathcal{S}$ such that for any PPT adversary $\mathcal{A}$, auxiliary input $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$, $(x; w) \in R$, it holds that*
$\Pr\left[\langle \mathcal{P}(\mathsf{pk}, w), \mathcal{A} \rangle = \mathsf{accept} : (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{G}(1^\lambda); (x, w) \leftarrow \mathcal{A}(z, \mathsf{pk}, \mathsf{vk})\right] =$
$\Pr\left[\langle \mathcal{S}(\mathsf{trap}, z, \mathsf{pk}), \mathcal{A} \rangle = \mathsf{accept} : (\mathsf{pk}, \mathsf{vk}, \mathsf{trap}) \leftarrow \mathcal{S}(1^\lambda); (x, w) \leftarrow \mathcal{A}(z, \mathsf{pk}, \mathsf{vk})\right]$

*We say that $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a **succinct** argument system if the running time of $\mathcal{V}$ and the total communication between $\mathcal{P}$ and $\mathcal{V}$ (proof size) are* $\mathsf{poly}(\lambda, |x|, \log |w|)$.

Protocol 1 (**Sumcheck**). The protocol proceeds in ℓ rounds.

- In the first round, $\mathcal{P}$ sends a univariate polynomial

  $$f_1(x_1) \overset{def}{=} \sum_{b_2,\ldots,b_\ell\in\{0,1\}} f(x_1, b_2, \ldots, b_\ell)\,,$$

  $\mathcal{V}$ checks $H = f_1(0) + f_1(1)$. Then $\mathcal{V}$ sends a random challenge $r_1 \in \mathbb{F}$ to $\mathcal{P}$.

- In the i-th round, where $2 \leq i \leq l-1$, $\mathcal{P}$ sends a univariate polynomial

  $$f_i(x_i) \overset{def}{=} \sum_{b_{i+1},\ldots,b_\ell\in\{0,1\}} f(r_1, \ldots, r_{i-1}, x_i, b_{i+1}, \ldots, b_\ell)\,,$$

  $\mathcal{V}$ checks $f_{i-1}(r_{i-1}) = f_i(0) + f_i(1)$, and sends a random challenge $r_i \in \mathbb{F}$ to $\mathcal{P}$.

- In the ℓ-th round, $\mathcal{P}$ sends a univariate polynomial

  $$f_\ell(x_\ell) \overset{def}{=} f(r_1, r_2, \ldots, r_{l-1}, x_\ell)\,,$$

  $\mathcal{V}$ checks $f_{\ell-1}(r_{\ell-1}) = f_\ell(0) + f_\ell(1)$. The verifier generates a random challenge $r_\ell \in \mathbb{F}$. Given oracle access to an evaluation $f(r_1, r_2, \ldots, r_\ell)$ of f, $\mathcal{V}$ will accept if and only if $f_\ell(r_\ell) = f(r_1, r_2, \ldots, r_\ell)$. The instantiation of the oracle access depends on the application of the sumcheck protocol.

2.2.3 GKR Protocol

In [GKR15], Goldwasser et al. proposed an efficient interactive proof protocol for layered arithmetic circuits, which we use as a building block for our new zero-knowledge argument and is referred as the *GKR* protocol. We present the detailed protocol here.

2.2.3.1 Sumcheck Protocol.

The sumcheck problem is a fundamental problem that has various applications. The problem is to sum a polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ on the binary hypercube

$$\textstyle\sum_{b_1,b_2,\ldots,b_\ell\in\{0,1\}} f(b_1, b_2, ..., b_\ell).$$

Directly computing the sum requires exponential time in ℓ, as there are 2^ℓ combinations of $b_1, \ldots, b_\ell$. Lund et al. [LFKN92] proposed a *sumcheck* protocol that allows a verifier $\mathcal{V}$ to delegate the computation to a computationally unbounded prover $\mathcal{P}$, who can convince $\mathcal{V}$ that H is the correct sum. We provide a description of the sumcheck protocol in Protocol 1. The proof size of the sumcheck protocol is $O(d\ell)$, where d is the variable-degree of f, as in each round, $\mathcal{P}$ sends a univariate polynomial of one variable in f, which can be uniquely defined by $d+1$ points. The verifier time of the protocol is $O(d\ell)$. The prover time depends on the degree and the sparsity of f, and we will give the complexity later in our scheme. The sumcheck protocol is complete and sound with $\epsilon = \frac{d\ell}{|\mathbb{F}|}$.

2.2.3.2 GKR protocol

Using the sumcheck protocol as a building block, Goldwasser et al. [GKR15] showed an interactive proof protocol for layered arithmetic circuits.

Definition 2.2.3 (Multi-linear Extension). *Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. The multilinear extension of V is the unique polynomial $\tilde{V} : \mathbb{F}^l \to \mathbb{F}$ such that $\tilde{V}(x_1, x_2, ..., x_l) = V(x_1, x_2, ..., x_l)$ for all $x_1, x_2, \dots, x_l \in \{0,1\}^l$.*

$\tilde{V}$ can be expressed as:

$$\tilde{V}(x_1, x_2, ..., x_l) = \sum_{b\in\{0,1\}^\ell} \prod_{i=1}^{l} [((1-x_i)(1-b_i) + x_i b_i) \cdot V(b)]$$

where b_i is i-th bit of b.

Multilinear extensions of arrays. Inspired by the close form equation of the multilinear extension given above, we can view an array $\mathbf{A} = (a_0, a_1, \dots, a_{n-1})$ as a function $A : \{0,1\}^{\log n} \to \mathbb{F}$ such that $\forall i \in [0, n-1], A(i) = a_i$. Therefore, in this paper, we abuse the use of multilinear extension on an array as the multilinear extension $\tilde{A}$ of A.

High Level Ideas. Let C be a layered arithmetic circuit with depth d over a finite field $\mathbb{F}$. Each gate in the i-th layer takes inputs from two gates in the $(i+1)$-th layer; layer 0 is the output layer and layer d is the input layer. The protocol proceeds layer by layer. Upon receiving the claimed output from $\mathcal{P}$, in the first round, $\mathcal{V}$ and $\mathcal{P}$ run the sumcheck protocol to reduce the claim about the output to a claim about the values in the layer above. In the i-th round, both parties reduce a claim about layer $i-1$ to a claim about layer i through the sumcheck protocol. Finally, the protocol terminates with a claim about the input layer d, which can be checked directly by $\mathcal{V}$, or is given as an oracle access. If the check passes, $\mathcal{V}$ accepts the claimed output.

Notation. Before describing the GKR protocol, we introduce some additional notations. We denote the number of gates in the i-th layer as S_i and let $s_i = \lceil \log S_i \rceil$. (For simplicity, we assume S_i is a power of 2, and we can pad the layer with dummy gates otherwise.) We then define a function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ that takes a binary string $b \in \{0,1\}^{s_i}$ and returns the output of gate b in layer i, where b is called the gate label. With this definition, V_0 corresponds to the output of the circuit, and V_d corresponds to the input layer. Finally, we define two additional functions $add_i, mult_i : \{0,1\}^{s_{i-1}+2s_i} \to \{0,1\}$, referred as *wiring predicates* in the literature. add_i ($mult_i$) takes one gate label $z \in \{0,1\}^{s_{i-1}}$ in layer $i-1$ and two gate labels $x, y \in \{0,1\}^{s_i}$ in layer i, and outputs 1 if and only if gate z is an addition (multiplication) gate that takes the output of gate x, y as input. With these definitions, V_i can be written as follows:

$$\begin{aligned} V_i(z) = \sum_{x,y\in\{0,1\}^{s_{i+1}}} & (add_{i+1}(z,x,y)(V_{i+1}(x) + V_{i+1}(y)) \\ & + mult_{i+1}(z,x,y)(V_{i+1}(x)V_{i+1}(y))) \end{aligned} \tag{2.1}$$

for any $z \in \{0,1\}^{s_i}$.

In the equation above, V_i is expressed as a summation, so $\mathcal{V}$ can use the sumcheck protocol to check that it is computed correctly. As the sumcheck protocol operates on polynomials defined on $\mathbb{F}$, we rewrite the equation with their multilinear extensions:

$$\begin{aligned}\tilde{V}_i(g) =& \sum\nolimits_{x,y\in\{0,1\}^{s_{i+1}}} f_i(x,y)\\ =& \sum\nolimits_{x,y\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(g,x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))\\ &+\tilde{mult}_{i+1}(g,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)))\,, \end{aligned} \tag{2.2}$$

where $g \in \mathbb{F}^{s_i}$ is a random vector.

Protocol. With Equation 2.2, the GKR protocol proceeds as follows. The prover $\mathcal{P}$ first sends the claimed output of the circuit to $\mathcal{V}$. From the claimed output, $\mathcal{V}$ defines polynomial $\tilde{V}_0$ and computes $\tilde{V}_0(g)$ for a random $g \in \mathbb{F}^{s_0}$. $\mathcal{V}$ and $\mathcal{P}$ then invoke a sumcheck protocol on Equation 2.2 with $i = 0$. As described in Section 2.2.3.1, at the end of the sumcheck, $\mathcal{V}$ needs an oracle access to $f_i(u,v)$, where u, v are randomly selected in $\mathbb{F}^{s_{i+1}}$. To compute $f_i(u,v)$, $\mathcal{V}$ computes $\tilde{add}_{i+1}(u,v)$ and $\tilde{mult}_{i+1}(u,v)$ locally (they only depend on the wiring pattern of the circuit, but not on the values), asks $\mathcal{P}$ to send $\tilde{V}_1(u)$ and $\tilde{V}_1(v)$ and computes $f_i(u,v)$ to complete the sumcheck protocol. In this way, $\mathcal{V}$ and $\mathcal{P}$ reduces a claim about the output to two claims about values in layer 1. $\mathcal{V}$ and $\mathcal{P}$ could invoke two sumcheck protocols on $\tilde{V}_1(u)$ and $\tilde{V}_1(v)$ recursively to layers above, but the number of claims and the sumcheck protocols would increase exponentially in d.

Combining two claims: condensing to one claim. In [GKR15], Goldwasser et al. presented a protocol to reduce two claims $\tilde{V}_i(u)$ and $\tilde{V}_i(v)$ to one as following. $\mathcal{V}$ defines a line $\gamma : \mathbb{F} \to \mathbb{F}^{s_i}$ such that $\gamma(0) = u, \gamma(1) = v$. $\mathcal{V}$ sends $\gamma(x)$ to $\mathcal{P}$. Then $\mathcal{P}$ sends $\mathcal{V}$ a degree s_i univariate polynomial $h(x) = \tilde{V}_i(\gamma(x))$. $\mathcal{V}$ checks that $h(0) = \tilde{V}_i(u), h(1) = \tilde{V}_i(v)$. Then $\mathcal{V}$ randomly chooses $r \in \mathbb{F}$ and computes a new claim $h(r) = \tilde{V}_i(\gamma(r)) = \tilde{V}_i(w)$ on $w = \gamma(r) \in \mathbb{F}^{s_i}$. $\mathcal{V}$ sends r, w to $\mathcal{P}$. In this way, the two claims are reduced to one claim $\tilde{V}_i(w)$. Combining this protocol with the sumcheck protocol on Equation 2.2, $\mathcal{V}$ and $\mathcal{P}$ can reduce a claim on layer i to one claim on layer $i + 1$, and eventually to a claim on the input, which completes the GKR protocol.

Combining two claims: random linear combination. In [CFS17], Chiesa et al. proposed an alternative approach using random linear combinations. Upon receiving the two claims $\tilde{V}_i(u)$ and $\tilde{V}_i(v)$, $\mathcal{V}$ selects $\alpha_i, \beta_i \in \mathbb{F}$ randomly and computes $\alpha_i\tilde{V}_i(u) + \beta_i\tilde{V}_i(v)$. Based on Equation 2.2, this random linear combination can be written as

$$\begin{aligned}
&\alpha_i\tilde{V}_i(u) + \beta_i\tilde{V}_i(v)\\
=&\alpha_i \sum_{x,y\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(u,x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y)) + \tilde{mult}_{i+1}(u,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)))\\
+&\beta_i \sum_{x,y\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(v,x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y)) + \tilde{mult}_{i+1}(v,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)))\\
=&\sum_{x,y\in\{0,1\}^{s_{i+1}}} ((\alpha_i\tilde{add}_{i+1}(u,x,y) + \beta_i\tilde{add}_{i+1}(v,x,y))(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))\\
&+ (\alpha_i\tilde{mult}_{i+1}(u,x,y) + \beta_i\tilde{mult}_{i+1}(v,x,y))(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y))) \qquad (2.3)
\end{aligned}$$

$\mathcal{V}$ and $\mathcal{P}$ then execute the sumcheck protocol on Equation 2.3 instead of Equation 2.2. At the end of the sumcheck protocol, $\mathcal{V}$ still receives two claims about $\tilde{V}_{i+1}$, computes their random linear combination and proceeds to an layer above recursively until the input layer.

In our new ZKP scheme, we will mainly use the second approach. The full GKR protocol using random linear combinations is given in Protocol 2.

Theorem 2.2.4. *[VSBW13][Tha13a][CMT12][GKR15]. Let $C : \mathbb{F}^n \rightarrow \mathbb{F}^k$ be a depth-d layered arithmetic circuit. Protocol 2 is an interactive proof for the function computed by C with soundness $O(d\log|C|/|\mathbb{F}|)$. It uses $O(d\log|C|)$ rounds of interaction and running time of the prover $\mathcal{P}$ is $O(|C|\log|C|)$. Let the optimal computation time for all $\tilde{add}_i$ and $\tilde{mult}_i$ be T, the running time of $\mathcal{V}$ is $O(n+k+d\log|C|+T)$. For log-space uniform circuits it is $T = \mathsf{poly}\log|C|$.*

Protocol 2. Let $\mathbb{F}$ be a prime field. Let $C\colon \mathbb{F}^n \to \mathbb{F}^k$ be a d-depth layered arithmetic circuit. $\mathcal{P}$ wants to convince that $\mathsf{out} = C(\mathsf{in})$ where in is the input from $\mathcal{V}$, and out is the output. Without loss of generality, assume n and k are both powers of 2 and we can pad them if not.

- Define the multilinear extension of array out as $\tilde{V}_0$. $\mathcal{V}$ chooses a random $g \in \mathbb{F}^{s_0}$ and sends it to $\mathcal{P}$. Both parties compute $\tilde{V}_0(g)$.
- $\mathcal{P}$ and $\mathcal{V}$ run a sumcheck protocol on

  $$\tilde{V}_0(g^{(0)}) = \sum_{x,y\in\{0,1\}^{s_1}} \tilde{mult}_1(g^{(0)},x,y)(\tilde{V}_1(x)\tilde{V}_1(y)) + \tilde{add}_1(g^{(0)},x,y)(\tilde{V}_1(x)+\tilde{V}_1(y))$$

  At the end of the protocol, $\mathcal{V}$ receives $\tilde{V}_1(u^{(1)})$ and $\tilde{V}_1(v^{(1)})$. $\mathcal{V}$ computes $\tilde{mult}_1(g^{(0)},u^{(1)},v^{(1)})$, $\tilde{add}_1(g^{(0)},u^{(1)},v^{(1)})$ and checks that $\tilde{mult}_1(g^{(0)},u^{(1)},v^{(1)})\tilde{V}_1(u^{(1)})\tilde{V}_1(v^{(1)}) + \tilde{add}_1(g^{(0)},u^{(1)},v^{(1)})(\tilde{V}_1(u^{(1)})+\tilde{V}_1(v^{(1)}))$ equals to the last message of the sumcheck.
- For $i = 1, ..., d-1$:
  - $\mathcal{V}$ randomly selects $\alpha^{(i)}, \beta^{(i)} \in \mathbb{F}$ and sends them to $\mathcal{P}$.
  - $\mathcal{P}$ and $\mathcal{V}$ run the sumcheck on the equation

    $$\begin{aligned}&\alpha^{(i)}\dot{V}_i(u^{(i)}) + \beta^{(i)}\dot{V}_i(v^{(i)}) = \\ &\quad\sum_{x,y\in\{0,1\}^{s_{i+1}}} ((\alpha^{(i)}\tilde{mult}_{i+1}(u^{(i)},x,y) + \beta^{(i)}\tilde{mult}_{i+1}(v^{(i)},x,y))(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)) \\ &\qquad + (\alpha^{(i)}\tilde{add}_{i+1}(u^{(i)},x,y) + \beta^{(i)}\tilde{add}_{i+1}(v^{(i)},x,y))(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))\end{aligned}$$

  - At the end of the sumcheck protocol, $\mathcal{P}$ sends $\mathcal{V}$ $\tilde{V}_{i+1}(u^{(i+1)})$ and $\tilde{V}_{i+1}(v^{(i+1)})$.
  - $\mathcal{V}$ computes the right hand side of the above equation by replacing x and y by $u^{(i+1)}$ and $v^{(i+1)}$ respectively, and checks if it equals to the last message of the sumcheck. If all checks in the sumcheck pass, V uses $\tilde{V}_{i+1}(u^{(i+1)})$ and $\tilde{V}_{i+1}(v^{(i+1)})$ to proceed to the $(i+1)$-th layer. Otherwise, $\mathcal{V}$ outputs reject and aborts.
- At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(u^{(d)})$ and $\tilde{V}_d(v^{(d)})$. $\mathcal{V}$ queries the oracle of evaluations of $\tilde{V}_d$ at $u^{(d)}$ and $v^{(d)}$ and checks that they are the same as the two claims. If yes, output accept; otherwise, output reject.

2.2.4 Zero-Knowledge Verifiable Polynomial Delegation Scheme

Let $\mathbb{F}$ be a finite field, $\mathcal{F}$ be a family of ℓ-variate polynomial over $\mathbb{F}$, and d be a variable-degree parameter. A zero-knowledge verifiable polynomial delegation scheme (zkVPD) for $f \in \mathcal{F}$ and $t \in \mathbb{F}^{\ell}$ consists of the following algorithms:

- $(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d)$,
- $\mathsf{com} \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$,
- $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{CheckComm}(\mathsf{com}, \mathsf{vp})$,
- $(y, \pi) \leftarrow \mathsf{Open}(f, t, r_f, \mathsf{pp})$,
- $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{Verify}(\mathsf{com}, t, y, \pi, \mathsf{vp})$.

A zkVPD scheme satisfies correctness, soundness and zero knowledge, which we formally define below.

Definition 2.2.5. *Let $\mathbb{F}$ be a finite field, $\mathcal{F}$ be a family of ℓ-variate polynomial over $\mathbb{F}$, and d be a variable-degree parameter. A zero-knowledge verifiable polynomial delegation scheme (zkVPD) consists of the following algorithms:* $(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d)$, $\mathsf{com} \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$,

$\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{CheckComm}(\mathsf{com}, \mathsf{vp})$, $(y, \pi) \leftarrow \mathsf{Open}(f, t, r_f, \mathsf{pp})$, $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{Verify}$ $(\mathsf{com}, t, y, \pi, \mathsf{vp})$, *such that*

- ***Perfect Completeness*** *For any polynomial $f \in \mathcal{F}$ and value t, the following probability is 1.*

$$\Pr_{r_f}\left[\begin{array}{ll} (\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d) & \\ \quad \mathsf{com} \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp}) : & \mathsf{CheckComm}(\mathsf{com}, \mathsf{vp}) = \mathsf{accept}\ \wedge \\ (y, \pi) \leftarrow \mathsf{Open}(f, t, r_f, \mathsf{pp}) & \quad \mathsf{Verify}(\mathsf{com}, t, y, \pi, \mathsf{vp}) = \mathsf{accept} \end{array}\right]$$

- ***Binding*** *For any PPT adversary $\mathcal{A}$ and benign auxiliary input z_1, z_2 the following probability is negligible of λ:*

$$\Pr\left[\begin{array}{rr} (\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d) & \mathsf{CheckComm}(\mathsf{com}^*, \mathsf{vp}) = \mathsf{accept}\ \wedge \\ (\pi^*, \mathsf{com}^*, y^*, state) \leftarrow \mathcal{A}(1^\lambda, z_1, \mathsf{pp}) : & \mathsf{Verify}(\mathsf{com}^*, t^*, y^*, \pi^*, \mathsf{vp}) = \mathsf{accept}\ \wedge \\ (f^*, t^*, r_f^*) \leftarrow \mathcal{A}(1^\lambda, z_2, state, \mathsf{pp}) & \mathsf{com}^* = \mathsf{Commit}(f^*, r_f^*, \mathsf{pp})\ \wedge \\ & (y^*, \pi^*) = \mathsf{Open}(f^*, t^*, r_f^*, \mathsf{pp})\ \wedge \\ & f^*(t^*) \neq y^* \end{array}\right]$$

- ***Zero Knowledge*** *For security parameter λ, polynomial f, adversary $\mathcal{A}$, and simulator $\mathcal{S}$, consider the* *following two experiments:*

$\mathsf{Real}_{\mathcal{A},f}(1^\lambda)$*:*

1. $(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d)$
2. $\mathsf{com} \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$
3. $k \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}, \mathsf{vp})$
4. *For* $i = 1, ..., k$ *repeat*
    - *a)* $t_i \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}, y_1, ..., y_{i-1}, \pi_1, ..., \pi_{i-1}, \mathsf{vp})$
    - *b)* $(y_i, \pi_i) \leftarrow \mathsf{Open}(f, t_i, r_f, \mathsf{pp})$
5. $b \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}, (y_1, ..., y_k, \pi_1, ..., \pi_k), \mathsf{vp})$
6. *Output b*

$\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$*:*

1. $(\mathsf{com}, \mathsf{pp}, \mathsf{vp}, \sigma) \leftarrow \mathsf{Sim}(1^\lambda, \ell, d)$
2. $k \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}, \mathsf{vp})$
3. *For* $i = 1, ..., k$ *repeat:*
    - *a)* $t_i \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}, y_1, ..., y_{i-1}, \pi_1, ..., \pi_{i-1}, \mathsf{vp})$
    - *b)* $(y_i, \pi_i, \sigma) \leftarrow \mathsf{Sim}(t_i, \sigma, \mathsf{pp})$
4. $b \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}, (y_1, ..., y_k, \pi_1, ..., \pi_k), \mathsf{vp})$
5. *Output b*

For any PPT adversary $\mathcal{A}$ *and all polynomial* $f \in \mathbb{F}$*, there exists simulator* $\mathcal{S}$ *such that*

$$|\Pr[\mathsf{Real}_{\mathcal{A},f}(1^\lambda) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

2.3 GKR Protocol with Linear Prover Time

In this section we present a new algorithm (see Algorithm 6) for the prover of the GKR protocol [GKR15] that runs in linear time for *arbitrary layered circuits*. Before that, we present some necessary building blocks.

2.3.1 Linear-time sumcheck for a multilinear function [Tha13a]

In [Tha13a], Thaler proposed a linear-time algorithm for the prover of the sumcheck protocol on a multilinear function f on ℓ variables (the algorithm runs in $O(2^\ell)$ time). We review this algorithm here. Recall that in the i-th round of the sumcheck protocol the prover sends the verifier the univariate polynomial on x_i

$$\sum_{b_{i+1},\ldots,b_{\ell,}\in\{0,1\}} f(r_1, \ldots, r_{i-1}, x_i, b_{i+1}, \ldots, b_\ell)\,,$$

where $r_1, \ldots, r_{i-1}$ are random values chosen by the verifier in previous rounds. Since f is multilinear, it suffices for the prover to send two evaluations of the polynomial at points $t = 0$ and $t = 1$, namely the evaluations

$$\sum_{b_{i+1},\ldots,b_{\ell,}\in\{0,1\}} f(r_1, \ldots, r_{i-1}, 0, b_{i+1}, \ldots, b_\ell) \tag{2.4}$$

and

$$\sum_{b_{i+1},\ldots,b_{\ell,}\in\{0,1\}} f(r_1, \ldots, r_{i-1}, 1, b_{i+1}, \ldots, b_\ell)\,. \tag{2.5}$$

Algorithm 1 $\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}, r_1, \ldots, r_\ell)$

Input: Multilinear f on ℓ variables, initial bookkeeping table $\mathbf{A}$, random $r_1, \ldots, r_\ell$;
Output: All function evaluations $f(r_1, \ldots, r_{i-1}, t, b_{i+1}, \ldots, b_\ell)$;

```
 1: for $i = 1, \ldots, \ell$ do
 2:     for $b \in \{0,1\}^{\ell-i}$ do                ▷ $b$ is both a number and its binary representation.
 3:         for $t = 0, 1, 2$ do
 4:             Let $f(r_1, \ldots, r_{i-1}, t, b) = \mathbf{A}[b] \cdot (1-t) + \mathbf{A}[b + 2^{\ell-i}] \cdot t$
 5:         end for
 6:         $\mathbf{A}[b] = \mathbf{A}[b] \cdot (1-r_i) + \mathbf{A}[b + 2^{\ell-i}] \cdot r_i$
 7:     end for
 8: end for
 9: Let $\mathcal{F}$ contain all function evaluations $f(.)$ computed at Step 4
10: return $\mathcal{F}$
```

To compute the above sums the prover maintains a *bookkeeping table* $\mathbf{A}$ for f. This table, at round i, has $2^{\ell-i+1}$ entries storing the values

$$f(r_1, \ldots, r_{i-1}, b_i, b_{i+1}, \ldots, b_\ell)$$

for all $b_i, \ldots, b_\ell \in \{0,1\}$ and is initialized with evaluations of f on the hypercube. For every entry of $\mathbf{A}$, the prover subsequently computes, as in Step 4 of Algorithm 1 FunctionEvaluations[2] two values

$$f(r_1, \ldots, r_{i-1}, 0, b_{i+1}, \ldots, b_\ell) \text{ and } f(r_1, \ldots, r_{i-1}, 1, b_{i+1}, \ldots, b_\ell)\,.$$

Once these function evaluations are in place, the prover can easily sum over them and compute the required sumcheck messages as reguired by Relations 2.4 and 2.5. This is done in Algorithm 2 SumCheck[3].

Complexity analysis. Both Algorithms 1 and 2 run in $O(2^\ell)$ time: The first iteration takes $O(2^\ell)$, the second $O(2^{\ell-1})$ and so on, and therefore the bound holds.

2.3.2 Linear-time sumcheck for products of multilinear functions [Tha13a]

The linear-time sumcheck in the previous section can be generalized to a product of two multilinear functions. Let now f and g be two multilinear functions on ℓ variables each, we describe a linear-time algorithm to compute the messages of the prover for the sumcheck on the product $f \cdot g$, as proposed in [Tha13a]. Note that we cannot use Algorithm 2 here since $f \cdot g$ is not multilinear. However, similarly with the single-function case, the prover must now send, at round i, the following evaluations at points $t = 0$, $t = 1$ and $t = 2$

$$\sum_{b_{i+1},\ldots,b_{\ell,} \in \{0,1\}} f(r_1, \ldots, r_{i-1}, t, b_{i+1}, \ldots, b_\ell) \cdot g(r_1, \ldots, r_{i-1}, t, b_{i+1}, \ldots, b_\ell)$$

[2] To be compatible with other protocols later, we use three values $t = 0, 1, 2$ in our evaluations instead of just two.

[3] We note here that although these two steps can be performed together in a single algorithm and without the need to store function evaluations, we explicitly decouple them with two different algorithms (FunctionEvaluations and SumCheck) for facilitating the presentation of more advanced protocols later.

Algorithm 2 $\{a_1, \dots, a_\ell\} \leftarrow \mathsf{SumCheck}(f, \mathbf{A}, r_1, \dots, r_\ell)$

```
Input: Multilinear $f$ on $\ell$ variables, initial bookkeeping table $\mathbf{A}$, random $r_1, \dots, r_\ell$;
Output: $\ell$ sumcheck messages for $\sum_{x \in \{0,1\}^\ell} f(x)$. Each message $a_i$ consists of 3 elements $(a_{i0}, a_{i1}, a_{i2})$;
$\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}, r_1, \dots, r_\ell)$
for $i = 1, \dots, \ell$ do
    for $t \in \{0, 1, 2\}$ do
        $a_{it} = \sum_{b \in \{0,1\}^{\ell-i}} f(r_1, \dots, r_{i-1}, t, b)$    ▷ All evaluations needed are in $\mathcal{F}$.
    end for
end for
return $\{a_1, \dots, a_\ell\}$;
```

Algorithm 3 $\{a_1, \dots, a_\ell\} \leftarrow \mathsf{SumCheckProduct}(f, \mathbf{A}_f, g, \mathbf{A}_g, r_1, \dots, r_\ell)$

```
Input: Multilinear $f$ and $g$, initial bookkeeping tables $\mathbf{A}_f$ and $\mathbf{A}_g$, random $r_1, \dots, r_\ell$;
Output: $\ell$ sumcheck messages for $\sum_{x \in \{0,1\}^\ell} f(x)g(x)$. Each message $a_i$ consists of 3 elements $(a_{i0}, a_{i1}, a_{i2})$;
$\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}_f, r_1, \dots, r_\ell)$
$\mathcal{G} \leftarrow \mathsf{FunctionEvaluations}(g, \mathbf{A}_g, r_1, \dots, r_\ell)$
for $i = 1, \dots, \ell$ do
    for $t \in \{0, 1, 2\}$ do
        $a_{it} = \sum_{b \in \{0,1\}^{\ell-i}} f(r_1, \dots, r_{i-1}, t, b) \cdot g(r_1, \dots, r_{i-1}, t, b)$ ▷ All evaluations needed are in $\mathcal{F}$ and $\mathcal{G}$.
    end for
end for
return $\{a_1, \dots, a_\ell\}$;
```

The above can be easily computed by computing evaluations for functions f and g *separately* using Algorithm 1 and the combining the results using our new Algorithm 3 SumCheckProduct. We now have the following lemma:

Lemma 2.3.1. *Algorithm* SumCheckProduct *runs in time* $O(2^\ell)$

Proof. All loops in SumCheckProduct require time $2^\ell + 2^{\ell-1} + \ldots = O(2^\ell)$. Also SumCheckProduct calls FunctionEvaluations twice (one for f and one for g) and each such call takes $O(2^\ell)$ time. □

2.3.3 Linear-time sumcheck for GKR functions

Let us now consider the sumcheck problem on a particular class of functions that are relevant for the GKR protocol (that is why we call them GKR functions). In particular we want to compute the sumcheck

$$\sum\nolimits_{x,y\in\{0,1\}^\ell} f_1(g,x,y)f_2(x)f_3(y)\,, \tag{2.6}$$

for a fixed point $g \in \mathbb{F}^\ell$, where $f_2(x), f_3(x) : \mathbb{F}^\ell \to \mathbb{F}$ are multilinear extensions of arrays $\mathbf{A}_{f_2}, \mathbf{A}_{f_3}$ of size 2^ℓ, and function $f_1 : \mathbb{F}^{3\ell} \to \mathbb{F}$ is the multilinear extension of a sparse array with $O(2^\ell)$ (out of $2^{3\ell}$ possible) nonzero elements. It is not hard to see that the sumcheck polynomials in GKR given by Equations 2.2 and 2.3 satisfy these properties.

We note here that applying Algorithm 1 FunctionEvaluations for this particular class of polynomials would lead to quadratic prover time. This is because f_1 has $2^{2\ell}$ variables to sum on yielding $O(2^{2\ell})$ complexity. However, one could take advantage of the sparsity of f_1: the prover can store only the $O(2^\ell)$ non-zero values of the bookkeeping table $\mathbf{A}$. This is exactly the approach used in many prior work [CMT12; Wah+17; ZGKPP18]. However, with this approach, the number of nonzero values that must be considered in Step 2 is always at most 2^ℓ, since it is not guaranteed that this number will reduce to half (i.e., to $2^{\ell-i}$) after every update in Step 6 of Algorithm 1 because it is sparse. Therefore, the overall complexity becomes $O(\ell \cdot 2^\ell)$.

In this section we effectively reduce this bound to $O(2^\ell)$. Our protocol divides the sumcheck into two phases: the first ℓ rounds bounding the variables of x to a random point u, and the last ℓ rounds bounding the variables of y to a random point v. The central idea lies in rewriting Equation 2.6 as follows

$$\begin{aligned}\sum\nolimits_{x,y\in\{0,1\}^\ell} f_1(g,x,y)f_2(x)f_3(y) &= \sum\nolimits_{x\in\{0,1\}^\ell} f_2(x)\sum\nolimits_{y\in\{0,1\}^\ell} f_1(g,x,y)f_3(y)\\ &= \sum\nolimits_{x\in\{0,1\}^\ell} f_2(x)h_g(x)\,,\end{aligned}$$

where $h_g(x) = \sum_{y\in\{0,1\}^\ell} f_1(g,x,y)f_3(y)$.

2.3.3.1 Phase one.

With the formula above, in the first ℓ rounds, the prover and the verifier are running exactly a sumcheck on a product of two multilinear functions $f_2 \cdot h_g$, since functions f_2 and h_g can be viewed as functions only in x—y can be considered constant (it is always summed on the hypercube). To compute the sumcheck messages for the first ℓ rounds, given their bookkeeping tables, we can call

$$\mathsf{SumCheckProduct}(h_g(x), \mathbf{A}_{h_g}, f_2(x), \mathbf{A}_{f_2}, u_1, \ldots, u_\ell)$$

in Algorithm 3. By Lemma 2.3.1 this will take $O(2^\ell)$ time. We now show how to initialize the bookkeeping tables in linear time.

Initializing the bookkeeping tables:

Initializing the bookkeeping table for f_2 in $O(2^\ell)$ time is trivial, since f_2 is a multilinear extension of an array and therefore the evaluations on the hypercube are known. Initializing the bookkeeping table for h_g in $O(2^\ell)$ time is more challenging but we can leverage the sparsity of f_1. Consider the following lemma.

Lemma 2.3.2. *Let $\mathcal{N}_x$ be the set of $(z, y) \in \{0,1\}^{2\ell}$ such that $f_1(z, x, y)$ is non-zero. Then for all $x \in \{0,1\}^\ell$, it is $h_g(x) = \sum_{(z,y)\in\mathcal{N}_x} I(g, z) \cdot f_1(z, x, y) \cdot f_3(y)$, where $I(g, z) = \prod_{i=1}^{\ell}((1 - g_i)(1 - z_i) + g_i z_i))$.*

Proof. As f_1 is a multilinear extension, as shown in [Tha13a], we have $f_1(g, x, y) = \sum_{z\in\{0,1\}^\ell} I(g, z)$ $f_1(z, x, y)$, where I is the multilinear extension of the identity polynomial, i.e., $I(w, z) = 1$ iff $w = z$ for all $w, z \in \{0,1\}^\ell$. Therefore, we have

$$h_g(x) = \sum_{y\in\{0,1\}^\ell} f_1(g, x, y) f_3(y) = \sum_{z,y\in\{0,1\}^\ell} I(g, z) f_1(z, x, y) f_3(y) = \sum_{(z,y)\in\mathcal{N}_x} I(g, z) \cdot f_1(z, x, y) \cdot f_3(y)$$

Moreover, $I(w, z) = \prod_{i=1}^{\ell}((1 - w_i)(1 - z_i) + w_i z_i))$ is the unique polynomial that evaluates to 1 iff $w = z$ for all $w, z \in \{0,1\}^\ell$. As the multilinear extension is unique, we have $I(g, z) = \prod_{i=1}^{\ell}((1 - g_i)(1 - z_i) + g_i z_i))$. □

Lemma 2.3.3. *The bookkeeping table $\mathbf{A}_{h_g}$ can be initialized in time $O(2^\ell)$.*

Proof. As f_1 is sparse, $\sum_{x\in\{0,1\}^\ell} |\mathcal{N}_x| = O(2^\ell)$. From Lemma 2.3.2, given the evaluations of $I(g, z)$ for all $z \in \{0,1\}^\ell$, the prover can iterate all $(z, y) \in \mathcal{N}_x$ for all x to compute $\mathbf{A}_{h_g}$. The full algorithm is presented in Algorithm 4.

Procedure $\mathsf{Precompute}(g)$ is to evaluate $\mathbf{G}[z] = I(g, z) = \prod_{i=1}^{\ell}((1-g_i)(1-z_i)+g_i z_i))$ for $z \in \{0,1\}^\ell$. By the closed-form of $I(g, z)$, the procedure iterates each bit of z, and multiples $1-g_i$ for $z_i = 0$ and multiples g_i for $z_i = 1$. In this way, the size of $\mathbf{G}$ doubles in each iteration, and the total complexity is $O(2^\ell)$.

Step 8-9 computes $h_g(x)$ using Lemma 2.3.2. When f_1 is represented as a map of (z, x, y), $f_1(z, x, y)$ for non-zero values, the complexity of these steps is $O(2^\ell)$. In the GKR protocol, this is exactly the representation of a gate in the circuit, where z, x, y are labels of the gate, its left input and its right input, and $f_1(z, x, y) = 1$. □

With the bookkeeping tables, the prover runs $\mathsf{SumCheckProduct}(h_g(x), \mathbf{A}_{h_g}, f_2(x), \mathbf{A}_{f_2}, u_1, \ldots, u_\ell)$ in Algorithm 3 and the total complexity for phase one is $O(2^\ell)$.

2.3.3.2 Phase two.

At this point, all variables in x have been bounded to random numbers u. In the second phase, the equation to sum on becomes

$$\sum_{y\in\{0,1\}^\ell} f_1(g, u, y) f_2(u) f_3(y)$$

Note here that $f_2(u)$ is merely a single value which we already computed in phase one. Both $f_1(g, u, y)$ and $f_3(y)$ are polynomials on y with ℓ variables. Similar to phase one, to compute the messages for the last ℓ rounds we can call

$$\mathsf{SumCheckProduct}(f_1(g, u, y), \mathbf{A}_{f_1}, f_3(y) \cdot f_2(u), \mathbf{A}_{f_3} \cdot f_2(u), , v_1, \ldots, v_\ell)\,.$$

Note here that $\mathbf{A}_{f_1}$ is the bookkeeping table for $f_1(g, u, y)$, not the original sparse function $f_1(g, x, y)$.

Initializing the bookkeeping table for f_1:

It now remains to initialize the bookkeeping table for $f_1(g, u, y)$ efficiently. Similar to phase one, we have the following lemma:

Algorithm 4 $\mathbf{A}_{h_g} \leftarrow \mathsf{Initialize\_PhaseOne}(f_1, f_3, \mathbf{A}_{f_3}, g)$

Input: Multilinear f_1 and f_3, initial bookkeeping tables $\mathbf{A}_{f_3}$, random $g = g_1, \dots, g_\ell$;
Output: Bookkeeping table $\mathbf{A}_{h_g}$;

```
procedure $\mathbf{G} \leftarrow \mathsf{Precompute}(g)$            ▷ G is an array of size $2^\ell$.
    Set $\mathbf{G}[0] = 1$
    for $i = 0, \dots, \ell - 1$ do
        for $b \in \{0,1\}^i$ do
            $\mathbf{G}[b, 0] = \mathbf{G}[b] \cdot (1 - g_{i+1})$
            $\mathbf{G}[b, 1] = \mathbf{G}[b] \cdot g_{i+1}$
        end for
    end for
end procedure
$\forall x \in \{0,1\}^\ell$, set $\mathbf{A}_{h_g}[x] = 0$
for every $(z, x, y)$ such that $f_1(z, x, y)$ is non-zero do
    $\mathbf{A}_{h_g}[x] = \mathbf{A}_{h_g}[x] + \mathbf{G}[z] \cdot f_1(z, x, y) \cdot \mathbf{A}_{f_3}[y]$
end for
return $\mathbf{A}_{h_g}$;
```

Lemma 2.3.4. *Let $\mathcal{N}_y$ be the set of $(z, x) \in \{0,1\}^{2\ell}$ such that $f_1(z, x, y)$ is non-zero. Then for all $y \in \{0,1\}^\ell$, it is $f_1(g, u, y) = \sum_{(z,x) \in \mathcal{N}_y} I(g, z) \cdot I(u, x) \cdot f_1(z, x, y)$.*

Proof. This immediately follows from the fact that f_1 is a multilinear extension. We have $f_1(g, u, y) = \sum_{z,y \in \{0,1\}^\ell} I(g, z) \cdot I(u, x) \cdot f_1(z, x, y)$, where the closed from of I is given in Lemma 2.3.2. □

Lemma 2.3.5. *The bookkeeping table $\mathbf{A}_{f_1}$ can be initialized in time $O(2^\ell)$.*

Proof. Similar to Algorithm 4, he prover again iterates all non-zero indices of f_1 to compute it using Lemma 2.3.4. The full algorithm is presented in Algorithm 5.

□

We now summarize the final linear-time algorithm for computing the prover messages for the sumcheck protocol on GKR functions. See Algorithm 6 SumCheckGKR.

Theorem 2.3.6. *Algorithm* SumCheckGKR *runs in $O(2^\ell)$ time.*

Proof. Follows from Lemma 2.3.1, 2.3.3 and 2.3.5. □

Algorithm 5 $\mathbf{A}_{f_1} \leftarrow \mathsf{Initialize\_PhaseTwo}(f_1, g, u)$

```
Input: Multilinear $f_1$, random $g = g_1, \ldots, g_\ell$ and $u = u_1, \ldots, u_\ell$;
Output: Bookkeeping table $\mathbf{A}_{f_1}$;
$\mathbf{G} \leftarrow \mathsf{Precompute}(g)$
$\mathbf{U} \leftarrow \mathsf{Precompute}(u)$
$\forall y \in \{0,1\}^\ell$, set $\mathbf{A}_{f_1}[y] = 0$
for every $(z, x, y)$ such that $f_1(z, x, y)$ is non-zero do
    $\mathbf{A}_{f_1}[y] = \mathbf{A}_{f_1}[y] + \mathbf{G}[z] \cdot \mathbf{U}[x] \cdot f_1(z, x, y)$
end for
return $\mathbf{A}_{f_1}$;
```

Algorithm 6 $\{a_1, \ldots, a_{2\ell}\} \leftarrow \mathsf{SumCheckGKR}(f_1, f_2, f_3, u_1 \ldots, u_\ell, v_1, \ldots, v_\ell, g)$

```
Input: Multilinear extensions $f_1(z, x, y)$ (with $O(2^\ell)$ non-zero entries), $f_2(x)$, $f_3(y)$ and their bookkeeping tables $\mathbf{A}_{f_2}, \mathbf{A}_{f_3}$, randomness $u = u_1, \ldots, u_\ell$ and $v = v_1, \ldots, v_\ell$ and point $g$;
Output: $2\ell$ sumcheck messages for $\sum_{x,y \in \{0,1\}^\ell} f_1(g, x, y) f_2(x) f_3(y)$;
$\mathbf{A}_{h_g} \leftarrow \mathsf{Initialize\_PhaseOne}(f_1, f_3, \mathbf{A}_{f_3}, g)$
$\{a_1, \ldots, a_\ell\} \leftarrow \mathsf{SumCheckProduct}(\sum_{y \in \{0,1\}^\ell} f_1(g, x, y) f_3(y), \mathbf{A}_{h_g}, f_2, \mathbf{A}_{f_2}, u_1, \ldots, u_\ell)$
$\mathbf{A}_{f_1} \leftarrow \mathsf{Initialize\_PhaseTwo}(f_1, g, u)$
$\{a_{\ell+1}, \ldots, a_{2\ell}\} \leftarrow \mathsf{SumCheckProduct}(f_1(g, u, y), \mathbf{A}_{f_1}, f_3(y) \cdot f_2(u), \mathbf{A}_{f_3} \cdot f_2(u), v_1, \ldots, v_\ell)$
return $\{a_1, \ldots, a_{2\ell}\}$
```

2.3.3.3 Generalizations of our technique.

Our technique can be extended to sumchecks of the general type

$$\sum\nolimits_{x_1, x_2, \ldots, x_c \in \{0,1\}^c} f_0(g, x_1, x_2, \ldots, x_c) f_1(x_1) f_2(x_2) \ldots f_c(x_c)\,,$$

where c is a constant, functions f_i are multilinear and $f_0()$ is sparse and consists of linearly-many nonzero monomials. We divide the protocol into c phases similarly as above. This generalization captures the sumcheck in the original GKR paper with identity polynomials (see [GKR15]), and our new algorithms also improve the prover time of this to linear.

2.3.4 Putting everything together

The sumcheck protocol in GKR given by Equation 2.2 can be decomposed into several instances that have the form of Equation 2.6 presented in the previous section. The term

$$\sum_{x,y\in\{0,1\}^{s_{i+1}}} \tilde{mult}_{i+1}(g,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y))$$

is exactly the same as Equation 2.6. The term $\sum_{x,y\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(g,x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))$ can be viewed as:

$$\sum_{x,y\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(g,x,y)\tilde{V}_{i+1}(x) + \sum_{x,y\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(g,x,y)\tilde{V}_{i+1}(y)$$

The first sum can be computed using the same protocol in Algorithm 6 without $f_3(y)$, and the second sum can be computed without $f_2(x)$. The complexity for both cases remains linear. Due to linearity of the sumcheck protocol, the prover can execute these 3 instances simultaneously in every round, and sum up the individual messages and send them to the veriifer.

Combining two claims. After the sumcheck in the GKR protocol is completed, as described in Section 2.2.3, the prover and the verifier need to combine the two claims about $\tilde{V}_{i+1}$ received at the end of the sumcheck protocol to one to avoid the exponential blow-up. There are two ways to combine the two claims and we show how to do each of them in linear time.

The second approach using random linear combinations is rather straight forward. After the output layers, $\mathcal{P}$ and $\mathcal{V}$ execute sumcheck protocol on Equation 2.3 instead of Equations 2.2, which still satisfies the properties of Equation 2.6. One could view it as 6 instances of Equation 2.6 and the prover time is still linear. Moreover, there is a better way to further improve the efficiency. Taking $\sum_{x,y\in\{0,1\}^{s_{i+1}}}(\alpha_i \tilde{mult}_{i+1}(u,x,y)+\beta_i \tilde{mult}_{i+1}(v,x,y))\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)$ as an example, in Algorithm 4, the prover runs Precompute twice on u and v to generate two arrays ($\mathbf{G}_1$ and $\mathbf{G}_2$), and sets $\mathbf{G}[b] = \alpha_i\mathbf{G}_1[b] + \beta_i\mathbf{G}_2[b]$ for all b. The rest of the algorithms remains the same. This only incurs a small overhead in practice in our implementation, compared to the original algorithm on Equation 2.6.

Though with the approach above we already have a linear prover GKR protocol, the technique to condense two points to one proposed in the original GKR protocol [GKR15] may still be interesting in some scenarios (e.g., in our implementation, we use this approach in the last layer and only make one query to the multi-linear extension of the input, which is more efficient practice). We present an algorithm to reduce two claims about $\tilde{V}_{i+1}$ to one in linear time. Recall that as described in Section 2.2.3, in the i-th layer, after the sumcheck, the verifier receives two claims $\tilde{V}(u), \tilde{V}(v)$. (Again we omit the superscript and subscript of i for the ease of interpretation.) She then defines a line $\gamma(x) : \mathbb{F} \to \mathbb{F}^s$ such that $\gamma(0) = u, \gamma(1) = v$ and the prover needs to provide $\tilde{V}(\gamma(x))$, a degree s univariate polynomial, to $\mathcal{V}$. If the prover computes it naively, which was done in all prior papers, it incurs $O(s2^s)$ time, as it is equivalent to evaluating $\tilde{V}()$ at $s+1$ points.

Algorithm 7 Compute $\tilde{V}(\gamma(x)) = \sum_{y\in\{0,1\}^s} I(\gamma(x), y)\tilde{V}(y)$

```
Initialize a binary tree $T$ with $s$ levels. We use $T_j[b]$ to denote the $b$-th node at level $j$.
for $b \in \{0,1\}^s$ do
    $T_s[b] = \tilde{V}(b)$.
    Multiply $T_s[b]$ with $b_s(c_s x + d_s) + (1 - b_s)(1 - c_s x - d_s)$.
end for
for $j = s - 1, \ldots, 1$ do
    for $b \in \{0,1\}^j$ do
        $T_j[b] = T_{j+1}[b, 0] + T_{j+1}[b, 1]$.
        $T_j[b] = T_j[b] \cdot (b_j(c_j x + d_j) + (1 - b_j)(1 - c_j x - d_j))$.
    end for
end for
Output $T_1[0]$.
```

In our new algorithm, we write $\tilde{V}(\gamma(x)) = \sum_{y\in\{0,1\}^s} I(\gamma(x), y)\tilde{V}(y)$, where $I(a, b)$ is an identity polynomial $I(a, b) = 0$ iff $a = b$. This holds by inspection of both sides on the Boolean hypercube. We then evaluate the right side in linear time with a binary tree structure. The key observation is that the identity polynomial can be written as $I(a, b) = \prod_{j=1}^{s}(a_j b_j + (1 - a_j)(1 - b_j))$, and we can process one variable (a_j, b_j) at a time and multiply them together to get the final result.

We construct a binary tree with 2^s leaves and initialize each leaf $b \in \{0,1\}^s$ with $\tilde{V}(b)$. As $\gamma(x)$ is a linear polynomial, we write it as $\gamma(x) = [c_1, \ldots, c_s]^T x + [d_1, \ldots, d_s]^T$. At the leaf level, we only consider the last variable of $I(\gamma(x), y)$. For each leaf $b \in \{0,1\}^s$, we multiply the value with $b_s(c_s x + d_s) + (1 - b_s)(1 - c_s x - d_s)$, the result of which is a linear polynomial. For a node $b \in \{0,1\}^j$ in the intermediate level j, we add the polynomials from its two children, and multiply it with $b_j(c_j x + d_j) + (1 - b_j)(1 - c_j x - d_j)$, the part in I that corresponds to the j-th variable. In this way, each node in the j-th level stores a degree j polynomial. Eventually, the root is the polynomial on the right side of degree s, which equals to $\tilde{V}(\gamma(x))$. The algorithm is given in Algorithm 7.

To see the complexity of Algorithm 7, both the storage and the polynomial multiplication at level j is $O(s - j + 1)$ in each node. So the total time is $O(\sum_{j=1}^{s} 2^j(s - j + 1)) = O(2^s)$, which is linear to the number of gates in the layer.

An alternative way to interpret this result is to add an additional layer for each layer of the circuit in GKR relaying the values. That is,

$$\tilde{V}_i(g) = \sum_{x\in\{0,1\}^{s_i}} I(g, x)\tilde{V}_{i+1}(x),$$

where $\tilde{V}_i = \tilde{V}_{i+1}$. Then when using the random linear combination approach, the sumcheck is executed on

$$\alpha\tilde{V}_i(u) + \beta\tilde{V}_i(v) = \sum_{x\in\{0,1\}^{s_i}} (\alpha I(u, x) + \beta I(v, x))\tilde{V}_{i+1}(x).$$

At the end of the sumcheck, the verifier receives a single claim on $\tilde{V}_{i+1} = \tilde{V}_i$. The sumcheck can obviously run in linear time, and the relay layers do not change the result of the circuit. This approach is actually the same as the condensing to one point in linear time above conceptually.

2.4 Zero-Knowledge Argument Protocols

In this section, we present the construction of our new zero-knowledge argument system. In [ZGKPP17c], Zhang et al. proposed to combine the GKR protocol with a verifiable polynomial delegation protocol, resulting in an argument system. Later, in [ZGKPP17a; WTSTW18], the construction was extended to zero-knowledge, by sending all the messages in the GKR protocol in homomorphic commitments and performing all the checks by zero-knowledge equality and product testing. This incurs a high overhead for the verifier compared to the plain version without zero-knowledge, as each multiplication becomes an exponentiation and each equality check becomes a Σ-protocol, which is around $100\times$ slower in practice.

In this paper, we follow the same blueprint of combining GKR and VPD to obtain an argument system, but instead show how to extend it to be zero-knowledge efficiently. In particular, the prover masks the GKR protocol with special random polynomials so that the verifier runs a “randomized” GKR that leaks no extra information and her overhead is small. A similar approach was used by Chiesa et al. in [CFS17]. In the following, we present the zero-knowledge version of each building block, followed by the whole zero-knowledge argument.

2.4.1 Zero Knowledge Sumcheck

As a core step of the GKR protocol, $\mathcal{P}$ and $\mathcal{V}$ execute a sumcheck protocol on Equation 2.2, during which $\mathcal{P}$ sends $\mathcal{V}$ evaluations of the polynomial at several random points chosen by $\mathcal{V}$. These evaluations leak information about the values in the circuit, as they can be viewed as weighted sums of these values.

To make the sumcheck protocol zero-knowledge, we take the approach proposed by Chiesa et al. in [CFS17], which is masking the polynomial in the sumcheck protocol by a random polynomial. In this approach, to prove

$$H = \sum\nolimits_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell),$$

the prover generates a random polynomial g with the same variables and individual degrees of f. She commits to the polynomial g, and sends the verifier a claim $G = \sum\limits_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} g(x_1, x_2, \ldots, x_\ell)$. The verifier picks a random number ρ, and execute a sumcheck protocol with the prover on

$$H + \rho G = \sum\nolimits_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} (f(x_1, x_2, \ldots, x_\ell) + \rho g(x_1, x_2, \ldots, x_\ell)).$$

At the last round of this sumcheck, the prover opens the commitment of g at $g(r_1, \ldots, r_\ell)$, and the verifier computes $f(r_1, \ldots, r_l)$ by subtracting $\rho g(r_1, \ldots, r_\ell)$ from the last message, and compares it with the oracle access of f. It is shown that as long as the commitment and opening of g are zero-knowledge, the protocol is zero-knowledge. Intuitively, this is because all the coefficients of f are masked by those of g. The soundness still holds because of the random linear combination of f and g.

Unfortunately, the masking polynomial g is as big as f, and opening it to a random point later is expensive. In [CFS17], the prover sends a PCP oracle of g, and executes a zero-knowledge sumcheck to open it to a random point, which incurs an exponential complexity for the prover. Even replacing it with the zkVPD protocol in [ZGKPP17a], the prover time is slow in practice.

In this paper, we show that it suffices to mask f with a small polynomial to achieve zero-knowledge. In particular, we set $g(x_1, \ldots, x_\ell) = a_0 + g_1(x_1) + g_2(x_2) + \ldots + g_\ell(x_\ell)$, where $g_i(x_i) = a_{i,1}x_i + a_{i,2}x_i^2 + \ldots + a_{i,d}x_i^d$ is a random univariate polynomial of degree d (d is the variable degree of f). Note here that the size of g is only $O(d\ell)$, while the size of f is exponential in ℓ.

The intuition of our improvement is that the prover sends $O(d\ell)$ messages in total to the verifier during the sumcheck protocol, thus a polynomial g with $O(d\ell)$ random coefficients is sufficient to mask all the messages and achieve zero-knowledge. We present the full protocol in Construction 1.

The completeness of the protocol holds obviously. The soundness follows the soundness of the sumcheck protocol and the random linear combination in step 2 and 3, as proven in [CFS17]. We give a proof of zero knowledge here.

Theorem 2.4.1 (Zero knowledge)**.** *For every verifier $\mathcal{V}^*$ and every ℓ-variate polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ with variable degree d, there exists a simulator $\mathcal{S}$ such that given access to $H = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$, $\mathcal{S}$ is able to simulate the partial view of $\mathcal{V}^*$ in step 1-4 of Construction 1.*

Proof. We build the simulator $\mathcal{S}$ as following.

Construction 1. *We assume the existence of a zkVPD protocol defined in Section 2.2.4. For simplicity, we omit the randomness r_f and public parameters pp, vp without any ambiguity. To prove the claim $H = \sum\limits_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$:*

1. *$\mathcal{P}$ selects a polynomial $g(x_1, \ldots, x_\ell) = a_0 + g_1(x_1) + g_2(x_2) + \ldots + g_l(x_\ell)$, where $g_i(x_i) = a_{i,1}x_i + a_{i,2}x_i^2 + \ldots + a_{i,d}x_i^d$ and all $a_{i,j}$s are uniformly random. $\mathcal{P}$ sends $H = \sum\limits_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$, $G = \sum\limits_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} g(x_1, x_2, \ldots, x_\ell)$ and $\mathsf{com}_g = \mathsf{Commit}(g)$ to $\mathcal{V}$.*

2. *$\mathcal{V}$ uniformly selects $\rho \in \mathbb{F}^*$, computes $H + \rho G$ and sends ρ to $\mathcal{P}$.*

3. *$\mathcal{P}$ and $\mathcal{V}$ run the sumcheck protocol on*

$$H + \rho G = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} (f(x_1, x_2, \ldots, x_\ell) + \rho g(x_1, x_2, \ldots, x_\ell))$$

4. *At the last round of the sumcheck protocol, $\mathcal{V}$ obtains a claim $h_\ell(r_\ell) = f(r_1, r_2, \ldots, r_\ell) + \rho g(r_1, r_2, \ldots, r_\ell)$. $\mathcal{P}$ and $\mathcal{V}$ opens the commitment of g at $r = (r_1, \ldots, r_\ell)$ by $(g(r), \pi) \leftarrow \mathsf{Open}(g, r), \mathsf{Verify}(\mathsf{com}_g, g(r), r, \pi)$. If Verify outputs reject, $\mathcal{V}$ aborts.*

5. *$\mathcal{V}$ computes $h_\ell(r_\ell) - \rho g(r_1, \ldots, r_\ell)$ and compares it with the oracle access of $f(r_1, \ldots, r_\ell)$.*

1. $\mathcal{S}$ selects a random polynomial $g^*(x_1,\ldots,x_\ell) = a_0^* + g_1^*(x_1) + g_2^*(x_2) + \cdots + g_\ell^*(x_\ell)$, where $g_i^*(x_i) = a_{i,1}^* x_i + a_{i,2}^* x_i^2 + \cdots + a_{i,d}^* x_i^d$. $\mathcal{S}$ sends H, $G^* = \sum_{x_1,x_2,\cdots,x_\ell \in \{0,1\}} g^*(x_1,x_2,\cdots,x_\ell)$ and $\mathsf{com}_{g^*} = \mathsf{Commit}(g^*)$ to $\mathcal{V}$.

2. $\mathcal{S}$ receives $\rho \neq 0$ from $\mathcal{V}^*$.

3. $\mathcal{S}$ selects a polynomial $f^* : \mathbb{F}^\ell \to \mathbb{F}$ with variable degree d uniformly at random conditioning on $\sum_{x_1,x_2,\cdots,x_\ell \in \{0,1\}} f^*(x_1,x_2,\cdots,x_\ell) = H$. $\mathcal{S}$ then engages in a sumcheck protocol with $\mathcal{V}$ on $H + \rho G^* = \sum_{x_1,x_2,\cdots,x_l \in \{0,1\}} (f^*(x_1,x_2,\cdots,x_\ell) + \rho g^*(x_1,x_2,\cdots,x_\ell))$

4. Let $r \in \mathbb{F}^\ell$ be the point chosen by $\mathcal{V}^*$ in the sumcheck protocol. $\mathcal{S}$ runs $(g^*(r),\pi) \leftarrow \mathsf{Open}(g^*, r)$ and sends them to $\mathcal{V}$.

As both g and g^* are randomly selected, and the zkVPD protocol is zero-knowledge, it is obvious that step 1 and 4 in $\mathcal{S}$ are indistinguishable from those in the real world of Construction 1. It remains to show that the sumchecks in step 3 of both worlds are indistinguishable.

To see that, recall that in round i of the sumcheck protocol, $\mathcal{V}$ receives a univariate polynomial $h_i(x_i) = \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} h(r_1,\ldots,r_{i-1},x_i,b_{i+1},\ldots,b_\ell)$ where $h = f + \rho g$. (The view of $\mathcal{V}^*$ is defined in the same way with h^*, f^*, g^* and we omit the repetition in the following.) As the variable degree of f and g is d, $\mathcal{P}$ sends $\mathcal{V}$ $h_i(0), h_i(1), \ldots, h_i(d)$ which uniquely defines $h_i(x_i)$. These evaluations reveal $d+1$ independent linear constraints on the coefficients of h. In addition, note that when these evaluations are computed honestly by $\mathcal{P}$, $h_i(0) + h_i(1) = h_{i-1}(r_{i-1})$, as required in the sumcheck protocol. Therefore, in all ℓ rounds of the sumcheck, $\mathcal{V}$ and $\mathcal{V}^*$ receives $\ell(d+1) - (\ell - 1) = \ell d + 1$ independent linear constraints on the coefficients of h and h^*.

As h and h^* are masked by g and g^*, each with exactly $\ell d + 1$ coefficients selected randomly, the two linear systems are identically distributed. Therefore, step 3 of the ideal world is indistinguishable from that of the real world.

□

2.4.2 Zero knowledge GKR

To achieve zero-knowledge, we replace the sumcheck protocol in GKR with the zero-knowledge version described in the previous section. However, the protocol still leaks additional information. In particular, at the end of the zero-knowledge sumcheck, $\mathcal{V}$ queries the oracle to evaluate the polynomial on a random point. When executed on Equation 2.2, this reveals two evaluations of the polynomial $\tilde{V}_i$ defined by the values in the i-th layer of the circuit: $\tilde{V}_i(u)$ and $\tilde{V}_i(v)$.

To prevent this leakage, Chiesa et al.[CFS17] proposed to replace the multi-linear extension $\tilde{V}_i$ with a low degree extension, such that learning $\tilde{V}_i(u)$ and $\tilde{V}_i(v)$ does not leak any information about V_i. Define a low degree extension of V_i as

$$\dot{V}_i(z) \overset{def}{=} \tilde{V}_i(z) + Z_i(z) \textstyle\sum_{w \in \{0,1\}^\lambda} R_i(z,w), \tag{2.7}$$

where $Z(z) = \prod_{i=1}^{s_i} z_i(1 - z_i)$, i.e., $Z(z) = 0$ for all $z \in \{0,1\}^{s_i}$. $R_i(z, w)$ is a random low-degree polynomial and λ is the security parameter. With this low degree extension, Equation 2.2 becomes

$$\begin{aligned}
\dot{V}_i(g) &= \sum\nolimits_{x,y\in\{0,1\}^{s_{i+1}}} \tilde{mult}_{i+1}(g,x,y)(\dot{V}_{i+1}(x)\dot{V}_{i+1}(y)) && (2.8)\\
&+ \tilde{add}_{i+1}(g,x,y)(\dot{V}_{i+1}(x) + \dot{V}_{i+1}(y)) + Z_i(g)\sum\nolimits_{w\in\{0,1\}^\lambda} R_i(g,w) \\
&= \sum\nolimits_{x,y\in\{0,1\}^{s_{i+1}},w\in\{0,1\}^\lambda} (I(\vec{0},w)\cdot \tilde{mult}_{i+1}(g,x,y)(\dot{V}_{i+1}(x)\dot{V}_{i+1}(y)) && (2.9)\\
&+ \tilde{add}_{i+1}(g,x,y)(\dot{V}_{i+1}(x) + \dot{V}_{i+1}(y)) + I((x,y),\vec{0})Z_i(g)R_i(g,w))
\end{aligned}$$

where $I(\vec{a}, \vec{b})$ is an identity polynomial $I(\vec{a}, \vec{b}) = 0$ iff $\vec{a} = \vec{b}$. The first equation holds because $\dot{V}_i$ agrees with $\tilde{V}_i$ on the Boolean hyper-cube $\{0,1\}^{s_i}$, as $Z_i(z) = 0$ for binary inputs. The second equation holds because the mask in $\dot{V}_i$ is in the form of a "sum" and can be moved into the sumcheck equation.

When executing the zero-knowledge sumcheck protocol on Equation 2.8, at the end of the protocol, $\mathcal{V}$ receives $\dot{V}_{i+1}(u)$ and $\dot{V}_{i+1}(v)$ for random points $u, v \in \mathbb{F}^{s_{i+1}}$ chosen by $\mathcal{V}$. They no longer leak information about V_{i+1}, as they are masked by $Z_{i+1}(z)\sum_{w\in\{0,1\}^\lambda} R_{i+1}(z,w)$ for $z = u$ and $z = v$. $\mathcal{V}$ computes $\tilde{mult}_{i+1}(g,u,v)$ and $\tilde{add}_{i+1}(g,u,v)$ as before, computes $Z_i(g), I(\vec{0},c), I((u,v),\vec{0})$ where $c \in \mathbb{F}^\lambda$ is a random point chosen by $\mathcal{V}$ for variable w, opens $R_i(g,w)$ at c with $\mathcal{P}$ through a polynomial commitment, and checks that together with $\dot{V}_{i+1}(u), \dot{V}_{i+1}(v)$ received from $\mathcal{P}$ they are consistent with the last message of the sumcheck.$\mathcal{V}$ then uses $\dot{V}_{i+1}(u), \dot{V}_{i+1}(v)$ to proceed to the next round.

Unfortunately, similar to the zk sumcheck, the masking polynomial R_i is very large in [CFS17]. Opening R_i at a random point takes exponential time for $\mathcal{P}$ either using a PCP oracle as in [CFS17] or potentially using a zkVPD, as R has $s_i + 2s_{i+1} + \lambda$ variables.

In this section, we show that we can set R_i to be a small polynomial to achieve zero-knowledge. In particular, R_i has only two variables with variable degree 2. This is because in the $(i-1)$-th round, $\mathcal{V}$ receives two evaluations of V_i, $\dot{V}_i(u)$ and $\dot{V}_i(v)$, which are masked by $\sum_w R_i(u,w)$ and $\sum_w R_i(v,w)$; in the i-th sumcheck, $\mathcal{V}$ opens R_i at $R_i(u,c)$ and $R_i(v,c)$. It suffices to make these four evaluations linearly independent, assuming the commitment and opening of R_i are using a zkVPD. Therefore, we set the low-degree term in Equation 2.7 as $Z_i(z)\sum_{w\in\{0,1\}} R_i(z_1,w)$, i.e. R_i only takes two variables, the first variable z_1 of z and an extra variable $w \in \{0,1\}$ instead of $\{0,1\}^\lambda$, with variable degree 2.

The full protocol is presented in Construction 2. Here we use superscriptions (e.g., $u^{(i)}$) to denote random numbers or vectors for the i-th layer of the circuit.

Construction 2. 1. *On a layered arithmetic circuit C with d layers and input* in*, the prover $\mathcal{P}$ sends the output of the circuit* out *to the verifier $\mathcal{V}$.*

2. *$\mathcal{P}$ randomly selects polynomials $R_1(z_1,w),\dots,R_d(z_1,w) : \mathbb{F}^2 \to \mathbb{F}$ with variable degree 2. $\mathcal{P}$ commits to these polynomials by sending* $\mathsf{com}_i \leftarrow \mathsf{Commit}(R_i)$ *to $\mathcal{V}$ for $i \in [1,d]$.*

3. *$\mathcal{V}$ defines $\dot{V}_0(z) = \tilde{V}_0(z)$, where $\tilde{V}_0(z)$ is the multilinear extension of* out. *$\dot{V}_0(z)$ can be viewed as a special case with $R_0(z_1,w)$ being the 0 polynomial. $\mathcal{V}$ evaluates it at a random point $\dot{V}_0(g^{(0)})$ and sends $g^{(0)}$ to $\mathcal{P}$.*

4. $\mathcal{P}$ and $\mathcal{V}$ execute the zero knowledge sumcheck protocol presented in Construction 1 on

$$\dot{V}_0(g^{(0)}) = \sum_{x,y\in\{0,1\}^{s_1}} \tilde{mult}_1(g^{(0)},x,y)(\dot{V}_1(x)\dot{V}_1(y)) \\ + \tilde{add}_1(g^{(0)},x,y)(\dot{V}_1(x)+\dot{V}_1(y))$$

If $u_1^{(1)} = v_1^{(1)}$, $\mathcal{P}$ aborts. At the end of the protocol, $\mathcal{V}$ receives $\dot{V}_1(u^{(1)})$ and $\dot{V}_1(v^{(1)})$. $\mathcal{V}$ computes $\tilde{mult}_1(g^{(0)},u^{(1)},v^{(1)})$, $\tilde{add}_1(g^{(0)},u^{(1)},v^{(1)})$ and checks that

$$\tilde{mult}_1(g^{(0)},u^{(1)},v^{(1)})\dot{V}_1(u^{(1)})\dot{V}_1(v^{(1)}) + \tilde{add}_1(g^{(0)},u^{(1)},v^{(1)})(\dot{V}_1(u^{(1)})+\dot{V}_1(v^{(1)}))$$

equals to the last message of the sumcheck (evaluation oracle).

5. For layer $i = 1,\dots,d-1$:

   a) $\mathcal{V}$ randomly selects $\alpha^{(i)},\beta^{(i)} \in \mathbb{F}$ and sends them to $\mathcal{P}$.

   b) Let $Mult_{i+1}(x,y) = \alpha^{(i)}\tilde{mult}_{i+1}(u^{(i)},x,y) + \beta^{(i)}\tilde{mult}_{i+1}(v^{(i)},x,y)$ and $Add_{i+1}(x,y) = \alpha^{(i)}\tilde{add}_{i+1}(u^{(i)},x,y) + \beta^{(i)}\tilde{add}_{i+1}(v^{(i)},x,y)$. $\mathcal{P}$ and $\mathcal{V}$ run the zero knowledge sumcheck on the equation

   $$\alpha^{(i)}\dot{V}_i(u^{(i)}) + \beta^{(i)}\dot{V}_i(v^{(i)}) = \\ \sum_{\substack{x,y\in\{0,1\}^{s_{i+1}} \\ w\in\{0,1\}}} (I(\vec{0},w)\cdot Mult_{i+1}(x,y)(\dot{V}_{i+1}(x)\dot{V}_{i+1}(y)) \\ + Add_{i+1}(x,y)(\dot{V}_{i+1}(x)+\dot{V}_{i+1}(y)) \\ + I((x,y),\vec{0})(\alpha^{(i)}Z_i(u^{(i)})R_i(u_1^{(i)},w) + \beta^{(i)}Z_i(v^{(i)})R_i(v_1^{(i)},w)))$$

   If $u_1^{(i+1)} = v_1^{(i+1)}$, $\mathcal{P}$ aborts.

   c) At the end of the zero-knowledge sumcheck protocol, $\mathcal{P}$ sends $\mathcal{V}$ $\dot{V}_{i+1}(u^{(i+1)})$ and $\dot{V}_{i+1}(v^{(i+1)})$.

   d) $\mathcal{V}$ computes

   $$a_{i+1} = \alpha^{(i)}\tilde{mult}_{i+1}(u^{(i)},u^{(i+1)},v^{(i+1)}) + \beta^{(i)}\tilde{mult}_{i+1}(v^{(i)},u^{(i+1)},v^{(i+1)})$$

   and

   $$b_{i+1} = \alpha^{(i)}\tilde{add}_{i+1}(u^{(i)},u^{(i+1)},v^{(i+1)}) + \beta^{(i)}\tilde{add}_{i+1}(v^{(i)},u^{(i+1)},v^{(i+1)})$$

   locally. $\mathcal{V}$ computes $Z_i(u^{(i)}), Z_i(v^{(i)}), I(\vec{0},c^{(i)}), I((u^{(i+1)},v^{(i+1)}),\vec{0})$ locally.

   e) $\mathcal{P}$ and $\mathcal{V}$ open R_i at two points $R_i(u_1^{(i)},c^{(i)})$ and $R_i(v_1^{(i)},c^{(i)})$ using Open and Verify.

f) $\mathcal{V}$ computes the following as the evaluation oracle and uses it to complete the last step of the zero-knowledge sumcheck.

$$\begin{aligned}&I(\vec{0}, c^{(i)})(a_{i+1}(\dot{V}_{i+1}(u^{(i+1)})\dot{V}_{i+1}(v^{(i+1)}))+\\&b_{i+1}(\dot{V}_{i+1}(u^{(i+1)}) + \dot{V}_{i+1}(v^{(i+1)})))+\\&I((u^{(i+1)}, v^{(i+1)}), \vec{0})(\alpha^{(i)} Z_i(u^{(i)})R_i(u_1^{(i)}, c^{(i)}) + \beta^{(i)} Z_i(v^{(i)})R_i(v_1^{(i)}, c^{(i)}))\end{aligned}$$

If all checks in the zero knowledge sumcheck and Verify *passes, $\mathcal{V}$ uses $\dot{V}_{i+1}(u^{(i+1)})$ and $\dot{V}_{i+1}(v^{(i+1)})$ to proceed to the $(i+1)$-th layer. Otherwise, $\mathcal{V}$ outputs* reject *and aborts.*

6. *At the input layer d, $\mathcal{V}$ has two claims $\dot{V}_d(u^{(d)})$ and $\dot{V}_d(v^{(d)})$. $\mathcal{V}$ opens R_d at 4 points $R_d(u_1^{(d)}, 0)$, $R_d(u_1^{(d)}, 1)$, $R_d(v_1^{(d)}, 0)$, $R_d(v_1^{(d)}, 1)$ and checks that $\dot{V}_d(u^{(d)}) = \tilde{V}_d(u^{(d)}) + Z_d(u^{(d)}) \sum_{w \in \{0,1\}} R_d$ $(u_1^{(d)}, w)$ and $\dot{V}_d(v^{(d)}) = \tilde{V}_d(v^{(d)}) + Z_d(v^{(d)}) \sum_{w \in \{0,1\}} R_d(v_1^{(d)}, w)$, given oracle access to two evaluates of $\tilde{V}_d$ at $u^{(d)}$ and $v^{(d)}$. If the check passes, output* accept; *otherwise, output* reject.

Theorem 2.4.2. *Construction 2 is an interactive proof protocol per Definition 2.2.1, for a function f defined by a layered arithmetic circuit C such that $f(\mathsf{in}, \mathsf{out}) = 1$ iff $C(\mathsf{in}) = \mathsf{out}$. In addition, for every verifier $\mathcal{V}^*$ and every layered circuit C, there exists a simulator $\mathcal{S}$ such that given oracle access to* out*, $\mathcal{S}$ is able to simulate the partial view of $\mathcal{V}^*$ in step 1-5 of Construction 2.*

The completeness follows from the construction explained above and the completeness of the zero knowledge sumcheck. The soundness follows the soundness of the GKR protocol with low degree extensions, as proven in [GKR15] and [CFS17]. We give the proof of zero knowledge here.

Proof. With oracle access to out, and the simulator $\mathcal{S}_{sc}$ of the zero-knowledge sumcheck protocol in Section 2.4.1 as a subroutine, we construct the simulator $\mathcal{S}$ as following:

1. $\mathcal{S}$ sends the out to $\mathcal{V}^*$.

2. $\mathcal{S}$ randomly selects polynomials $R_1^*(z_1, w), \ldots, R_d^*(z_1, w) : \mathbb{F}^2 \rightarrow \mathbb{F}$ with variable degree 2. $\mathcal{S}$ commits to these polynomials by sending $\mathsf{com}_i \leftarrow \mathsf{Commit}(R_i^*)$ to $\mathcal{V}^*$ for $i \in [1, d]$.

3. $\mathcal{S}$ receives $g^{(0)}$ from $\mathcal{V}^*$.

4. $\mathcal{S}$ calls $\mathcal{S}_{sc}$ to simulate the partial view of the zero knowledge sumcheck protocol on

$$\dot{V}_0(g^{(0)}) = \sum_{x,y \in \{0,1\}^{s_1}} \tilde{mult}_1(g^{(0)}, x, y)(\dot{V}_1(x)\dot{V}_1(y)) + \tilde{add}_1(g^{(0)}, x, y)(\dot{V}_1(x) + \dot{V}_1(y))$$

   If $u_1^{(1)} = v_1^{(1)}$, $\mathcal{S}$ aborts. At the end of the sumcheck, $\mathcal{S}$ samples $\dot{V}_1^*(u^{(1)})$ and $\dot{V}_1^*(v^{(1)})$ such that $\tilde{mult}_1(g^{(0)}, u^{(1)}, v^{(1)})\dot{V}_1^*(u^{(1)})\dot{V}_1^*(v^{(1)}) + \tilde{add}_1(g^{(0)}, u^{(1)}, v^{(1)})$ $(\dot{V}_1^*(u^{(1)}) + \dot{V}_1^*(v^{(1)}))$ equals to the last message of the sumcheck.

5. For layer $i = 1, \ldots, d-1$:

a) $\mathcal{S}$ receives $\alpha^{(i)}, \beta^{(i)}$ from $\mathcal{V}^*$.

b) Let $Mult_{i+1}(x, y) = \alpha^{(i)}\tilde{mult}_{i+1}(u^{(i)}, x, y) + \beta^{(i)}\tilde{mult}_{i+1}(v^{(i)}, x, y)$ and $Add_{i+1}(x, y) = \alpha^{(i)}\tilde{add}_{i+1}(u^{(i)}, x, y) + \beta^{(i)}\tilde{add}_{i+1}(v^{(i)}, x, y)$. $\mathcal{S}$ calls $\mathcal{S}_{sc}$ to simulate the partial view of the zero knowledge sumcheck protocol on

$$\begin{aligned}
&\alpha^{(i)}\dot{V}_i(u^{(i)}) + \beta^{(i)}\dot{V}_i(v^{(i)}) = \\
&\qquad \sum_{\substack{x,y \in \{0,1\}^{s_{i+1}} \\ w \in \{0,1\}}} (I(\vec{0}, w) \cdot Mult_{i+1}(x, y)(\dot{V}_{i+1}(x)\dot{V}_{i+1}(y)) \\
&\qquad + Add_{i+1}(x, y)(\dot{V}_{i+1}(x) + \dot{V}_{i+1}(y)) \\
&\qquad + I((x, y), \vec{0})(\alpha^{(i)}Z_i(u^{(i)})R_i(u_1^{(i)}, w) + \beta^{(i)}Z_i(v^{(i)})R_i(v_1^{(i)}, w)))
\end{aligned}$$

If $u_1^{(i+1)} = v_1^{(i+1)}$, $\mathcal{S}$ aborts.

c) At the end of the zero-knowledge sumcheck protocol, if $u_1^{(i+1)} = v_1^{(i+1)}$, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ samples $\dot{V}^*_{i+1}(u^{(i+1)})$ and $\dot{V}^*_{i+1}(v^{(i+1)})$ randomly such that the following equals to the last message of the sumcheck protocol.

$$\begin{aligned}
&I(\vec{0}, c^{(i)})(a_{i+1}(\dot{V}^*_{i+1}(u^{(i+1)})\dot{V}^*_{i+1}(v^{(i+1)})) + b_{i+1}(\dot{V}^*_{i+1}(u^{(i+1)}) + \dot{V}^*_{i+1}(v^{(i+1)}))) \\
&\quad + I((u^{(i+1)}, v^{(i+1)}), \vec{0})(\alpha^{(i)}Z_i(u^{(i)})R^*_i(u_1^{(i)}, c^{(i)}) + \beta^{(i)}Z_i(v^{(i)})R^*_i(v_1^{(i)}, c^{(i)}))
\end{aligned}$$

$a_{i+1} = \alpha^{(i)}\tilde{mult}_{i+1}(u^{(i)}, u^{(i+1)}, v^{(i+1)}) + \beta^{(i)}\tilde{mult}_{i+1}(v^{(i)}, u^{(i+1)}, v^{(i+1)})$ and $b_{i+1} = \alpha^{(i)}\tilde{add}_{i+1}(u^{(i)}, u^{(i+1)}, v^{(i+1)}) + \beta^{(i)}\tilde{add}_{i+1}(v^{(i)}, u^{(i+1)}, v^{(i+1)})$. $\mathcal{S}$ sends $\dot{V}_{i+1}(u^{(i+1)})$ and $\dot{V}_{i+1}(v^{(i+1)})$ to $\mathcal{V}^*$.

d) $\mathcal{V}^*$ computes the corresponding values locally as in step 5(d) of Construction 2.

e) $\mathcal{S}$ opens R^*_i at two points $R^*_i(u_1^{(i)}, c^{(i)})$ and $R^*_i(v_1^{(i)}, c^{(i)})$ using Open.

f) $\mathcal{V}^*$ performs the checks as in step 5(f) of Construction 2.

Note here that $\mathcal{V}^*$ can actually behave arbitrarily in step 5(d) and 5(f) above. We include these steps to be consistent with the real world in Construction 2 for the ease of interpretation.

To prove zero-knowledge, step 1,3, 5(a), 5(d) and 5(f) are obviously indistinguishable as $\mathcal{S}$ only receives messages from $\mathcal{V}^*$. Step 2 and 5(e) of both worlds are indistinguishable because of the zero knowledge property of the zkVPD, and the fact that R^* and R are sampled randomly in both worlds. Step 4 and 5(b) are indistinguishable as proven in Theorem 2.4.1 for $\mathcal{S}_{sc}$.

It remains to consider the messages received at the end of step 4 and in step 5(c), namely $\dot{V}_i(u^{(i)}), \dot{V}_i(v^{(i)})$ and $\dot{V}^*_i(u^{(i)}), \dot{V}^*_i(v^{(i)})$ for $i = 1, \ldots, d$. In the real world, $\dot{V}_i(z)$ is masked by $\sum_{w \in \{0,1\}} R_i(z_1, w)$ ($Z(z)$ is publicly known), thus $\dot{V}_i(u^{(i)})$ and $\dot{V}_i(v^{(i)})$ are masked by $\sum_{w \in \{0,1\}} R_i(u_1^{(i)}, w)$ and $\sum_{w \in \{0,1\}} R_i(v_1^{(i)}, w)$ correspondingly. In addition, in step 5(e), $\mathcal{V}^*$ opens R_i at $R_i(u_1^{(i)}, c^{(i)})$ and $R_i(v_1^{(i)}, c^{(i)})$. To simplify the notation here, we consider only a particular layer and omit the subscription and superscription of i. Let

$R(z_1, w) = a_0 + a_1 z_1 + a_2 w + a_3 z_1 w + a_4 z_1^2 + a_5 w^2 + a_6 z_1^2 w^2$, where $a_0, \dots, a_6$ are randomly chosen. We can write the four evaluations above as

$$\begin{bmatrix} 2 & 2u_1 & 1 & u_1 & 2u_1^2 & 1 & u_1^2 \\ 2 & 2v_1 & 1 & v_1 & 2v_1^2 & 1 & v_1^2 \\ 1 & u_1 & c & cu_1 & u_1^2 & c^2 & c^2 u_1^2 \\ 1 & v_1 & c & cv_1 & v_1^2 & c^2 & c^2 v_1^2 \end{bmatrix} \times \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \end{bmatrix}^T$$

After row reduction, the left matrix is

$$\begin{bmatrix} 2 & 2u_1 & 1 & u_1 & 2u_1^2 & 1 & u_1^2 \\ 0 & 2(v_1 - u_1) & 0 & v_1 - u_1 & 2(u_1^2 - v_1^2) & 0 & u_1^2 - v_1^2 \\ 0 & 0 & 2c - 1 & (2c-1)u_1 & 0 & 2c^2 - 1 & (2c^2 - 1)u_1^2 \\ 0 & 0 & 0 & (2c-1)(v_1 - u_1) & 0 & 0 & (2c^2 - 1)(v_1^2 - u_1^2) \end{bmatrix}$$

As $u_1 \neq v_1$, the matrix has full rank if $2c^2 - 1 \neq 0 \mod p$, where p is the prime that defines $\mathbb{F}$. This holds if 2^{-1} is not in the quadratic residue of p, or equivalently $p \not\equiv 1, 7 \mod 8$.[4] In case $p \equiv 1, 7 \mod 8$, we can add a check to both the protocol and the simulator to abort if $2c^2 - 1 = 0$. This does not affect the proof of zero knowledge, and only reduces the soundness error by a small amount. [5]

Because of the full rank of the matrix, the four evaluations are linearly independent and uniformly distributed, as $a_0, \dots a_6$ are chosen randomly. In the ideal world, $R^*(u_1, c)$ and $R^*(v_1, c)$ are independent and uniformly distributed, and $\dot{V}^*(u), \dot{V}^*(v)$ are randomly selected subject to a linear constraint (step 5(c)), which is the same as the real world. Therefore, they are indistinguishable in the two worlds, which completes the proof.

□

2.4.3 Zero knowledge VPD

In this section, we present the instantiations of the zkVPD protocol, as defined in Definition 2.2.5. For every intermediate layer i, we use the same zkVPD protocol as proposed by Zhang et al. in [ZGKPP17a] to commit and open the masking polynomials $g_i(x), R_i(z_1, w)$. In fact, as we show in the previous sections, these polynomials are very small (g_i is the sum of univariate polynomials and R_i has 2 variables with variable degree 2), the zkVPD protocols become very simple. The complexity of KeyGen, Commit, Open, Verify and proof size are all $O(s_i)$ for g_i and are all $O(1)$ for R_i. We omit the full protocols due to space limit.

For the zkVPD used for the input layer, we design a customized protocol based on the zkVPD protocol in [ZGKPP17a]. Recall that at the end of the GKR protocol, $\mathcal{V}$ receives two evaluations of the polynomial $\dot{V}_d(z) = \tilde{V}_d(z) + Z_d(z) \sum_{w \in \{0,1\}} R_d(z_1, w)$ at $z = u^{(d)}$ and $z = v^{(d)}$. In our zero knowledge proof protocol, which will be presented in Section 2.4.4, $\mathcal{P}$ commits to $\dot{V}_d(z)$ using the zkVPD at the beginning, and opens it to the two points selected by $\mathcal{V}$.

The protocol in [ZGKPP17a] works for any polynomial with ℓ variables and any variable degree, and is particularly efficient for multilinear polynomials. We modify the protocol for our zero-knowledge proof

[4] From the reduced matrix, we can see that setting $a_2 = a_3 = a_4 = 0$ does not affect the rank of the matrix, which simplifies the masking polynomial R in practice.

[5] If one is willing to perform a check like this, we can simplify the masking polynomial R to be multilinear. The reduced matrix will be the first 4 columns of the matrix showed above, and it has full rank if $c \neq 2^{-1}$.

Construction 3. *Let $\mathbb{F}$ be a prime-order finite field. Let $\dot{V}(x) : \mathbb{F}^{\ell} \to \mathbb{F}$ be an ℓ-variate polynomial such that $\dot{V}(x) = \tilde{V}(x) + Z(x)R(x_1)$, where $\tilde{V}(x)$ is a multilinear polynomial, $Z(x) = \prod_{i=1}^{\ell} x_i(1 - x_i)$ and $R(x_1) = a_0 + a_1 x_1$.*

- $(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^{\lambda}, \ell)$: *Select $\alpha, t_1, t_2, \cdots, t_l, t_{\ell+1} \in \mathbb{F}$ uniformaly at random, run* $\mathsf{bp} \leftarrow \mathsf{BilGen}(1^{\lambda})$ *and compute* $\mathsf{pp} = (\mathsf{bp}, g^{\alpha}, g^{t_{\ell+1}}, g^{\alpha t_{\ell+1}}, \{g^{\prod_{i \in W} t_i}, g^{\alpha \prod_{i \in W} t_i}\}_{W \in \mathcal{W}_{\ell}})$*, where $\mathcal{W}_{\ell}$ is the set of all subsets of $\{1, \ldots, \ell\}$. Set* $\mathsf{vp} = (\mathsf{bp}, g^{t_1}, \ldots, g^{t_{\ell+1}}, g^{\alpha})$.
- $\mathsf{com} \leftarrow \mathsf{Commit}(\dot{V}, r_V, r_R, \mathsf{pp})$: *Compute $c_1 = g^{\tilde{V}(t_1, t_2, \cdots, t_{\ell}) + r_V t_{\ell+1}}$, $c_2 = g^{\alpha(\tilde{V}(t_1, t_2, \cdots, t_{\ell}) + r_V t_{\ell+1})}$, $c_3 = g^{R(t_1) + r_R t_{\ell+1}}$ and $c_4 = g^{\alpha(R(t_1) + r_R t_{\ell+1})}$ output the commitment* $\mathsf{com} = (c_1, c_2, c_3, c_4)$.
- $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{CheckComm}(\mathsf{com}, \mathsf{vp})$: *Output* accept *if $e(c_1, g^{\alpha}) = e(c_2, g)$ and $e(c_3, g^{\alpha}) = e(c_4, g)$. Otherwise, output* reject.
- $(y, \pi) \leftarrow \mathsf{Open}(\dot{V}, r_V, r_R, u, \mathsf{pp})$: *Choose $r_1, \ldots, r_{\ell} \in \mathbb{F}$ at random, and compute polynomials q_i such that*
$$\tilde{V}(x) + r_V x_{\ell+1} + Z(u)(R(x_1) + r_R x_{\ell+1}) - (\tilde{V}(u) + Z(u)R(u_1)) =$$
$$\sum_{i=1}^{\ell}(x_i - u_i)(q_i(x_i, \ldots, x_{\ell}) + r_i x_{\ell+1}) + x_{\ell+1}(r_V + r_R Z(u) - \sum_{i=1}^{\ell} r_i(x_i - u_i)).$$
Set $\pi = (\{g^{q_i(t_i \ldots, t_{\ell}) + r_i t_{\ell+1}}, g^{\alpha(q_i(t_i \ldots, t_{\ell}) + r_i t_{\ell+1})}\}_{i \in [1, \ell]}, \quad g^{r_V + r_R Z(u) - \sum_{i=1}^{\ell} r_i(t_i - u_i)}, g^{\alpha(r_V + r_R Z(u) - \sum_{i=1}^{\ell} r_i(t_i - u_i))})$ *and $y = \tilde{V}(u) + Z(u)R(u_1)$.*
- $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{Verify}(\mathsf{com}, u, y, \pi, \mathsf{vp})$: *Parse π as $(\pi_i, \pi_{\alpha i})$ for $i \in [1, \ell + 1]$. Check $e(\pi_i, g^{\alpha}) = e(\pi_{\alpha i}, g)$ for $i \in [1, \ell{+}1]$. Check $e(c_1 c_3^{Z(u)}/g^y, g) = \prod_{i=1}^{\ell} e(\pi_i, g^{t_i - u_i}) \cdot e(g^{\pi_{\ell+1}}, g^{t_{\ell+1}})$. Output* accept *if all the checks pass, otherwise, output* reject.

scheme and preserve the efficiency. Note that though $\dot{V}_d(z)$ is a low degree extension of the input, it can be decomposed to the sum of $\tilde{V}_d(z)$, a multilinear polynomial, and $Z_d(z)\sum_{w \in \{0,1\}} R_d(z_1, w)$. Moreover, $Z_d(u^{(d)})$ and $Z_d(v^{(d)})$ can be computed directly by $\mathcal{V}$. Therefore, in our construction, $\mathcal{P}$ commits to $\tilde{V}_d(z)$ and $\sum_{w \in \{0,1\}} R_d(z_1, w)$ separately, and later opens the sum together given $Z_d(u^{(d)})$ and $Z_d(v^{(d)})$, which is naturally supported because of the homomorphic property of the commitment. Another optimization is that unlike other layers of the circuit, $R_d(z_1, w)$ itself is not opened at two points ($\mathcal{V}$ does not receive $R_d(u^{(d)}, c^{(d)})$ and $R_d(v^{(d)}, c^{(d)})$ in Construction 2). Therefore, it suffices to set $\dot{V}_d(z) = \tilde{V}_d(z) + Z_d(z)R_d(z_1)$, where R_d is a univariate linear polynomial. The full protocol is presented in Construction 3.

Theorem 2.4.3. *Construction 3 is a zero-knowledge verifiable polynomial delegation scheme as defined by Definition 2.2.5, under Assumption 1 and 2.*

The proof of completeness, soundness and zero knowledge is similar to that of the zkVPD protocol in [ZGKPP17a]. We only add an extra univariate linear polynomial $R(x_1)$, which does not affect the proof.

We omit the proof due to space limit. Using the same algorithms proposed in in [ZGKPP18; ZGKPP17a], the running time of KeyGen, Commit and Open is $O(2^\ell)$, Verify takes $O(\ell)$ time and the proof size is $O(\ell)$.

2.4.4 Putting Everything Together

In this section, we present our zero knowledge argument scheme. At a high level, similar to [ZGKPP17c; WTSTW18; ZGKPP17a], $\mathcal{V}$ can use the GKR protocol to verify the correct evaluation of a circuit C on input x and a witness w, given an oracle access to the evaluation of a polynomial defined by x, w on a random point. We instantiate the oracle using the zkVPD protocol. Formally, we present the construction in Construction 4, which combines our zero knowledge GKR and zkVPD protocols. Similar to the protocols in [ZGKPP17a; WTSTW18], Step 6 and 7 are to check that $\mathcal{P}$ indeed uses x as the input to the circuit.

Theorem 2.4.4. *For an input size n and a finite field $\mathbb{F}$, Construction 4 is a zero knowledge argument for the relation*

$$\mathcal{R} = \{(C, x; w) : C \in \mathcal{C}_{\mathbb{F}} \wedge |x| + |w| \leq n \wedge C(x; w) = 1\},$$

as defined in Definition 2.2.2, under Assumption 1 and 2. Moreover, for every $(C, x; w) \in \mathcal{R}$, the running time of $\mathcal{P}$ is $O(|C|)$ field operations and $O(n)$ multiplications in the base group of the bilinear map. The running time of $\mathcal{V}$ is $O(|x|+d\cdot\log|C|)$ if C is log-space uniform with d layers. $\mathcal{P}$ and $\mathcal{V}$ interact $O(d\log|C|)$ rounds and the total communication (proof size) is $O(d\log|C|)$. In case d is $\mathsf{polylog}(|C|)$, the protocol is a succinct argument.

Proof Sketch. The correctness and the soundness follow from those of the two building blocks, zero knowledge GKR and zkVPD, by Theorem 2.4.2 and 2.4.3.

To prove zero knowledge, consider a simulator $\mathcal{S}$ that calls the simulator $\mathcal{S}_{GKR}$ of zero knowledge GKR given in Section 2.4.2 as a subroutine, which simulates the partial view up to the input layer. At the input layer, the major challenge is that $\mathcal{S}$ committed to (a randomly chosen) $\dot{V}_d^*$ at the beginning of the protocol, before knowing the points $u^{(d)}, v^{(d)}$ to evaluate on. If $\mathcal{S}$ opens the commitment honestly, with high probability the evaluations are not consistent with the last message of the GKR (sumcheck in layer $d-1$) and a malicious $\mathcal{V}^*$ can distinguish the ideal world from the real world. In our proof, we resolve this issue by using the simulator $\mathcal{S}_{VPD}$ of our zkVPD protocol. Given the trapdoor trap used in KeyGen, $\mathcal{S}_{VPD}$ is able to open the commitment to any value in zero knowledge, and in particular it opens to those messages that are consistent with the GKR protocol in our scheme, which completes the construction of $\mathcal{S}$.

The complexity of our zero knowledge argument scheme follows from our new GKR protocol with linear prover time, and the complexity of the zkVPD protocol for the input layer analyzed in Section 2.4.3. The masking polynomials g_i, R_i and their commitments and openings introduce no asymptotic overhead and are efficient in practice.

Removing interaction. Our construction can be made non-interactive in the random oracle model using Fiat--Shamir heuristic [FS86]. Though GKR protocol is not constant round, recent results [BSCS16; CCHLRR18] show that applying Fiat-Shamir only incurs a polynomial soundness loss in the number of rounds in GKR. In our implementation, the GKR protocol is on a 254-bit prime field matching the bilinear group used in the zkVPD. The non-interactive version of our system provides a security level of 100+ bits.

Construction 4. *Let λ be the security parameter, $\mathbb{F}$ be a prime field, n be an upper bound on input size, and S be an upper bound on circuit size. We use* $\mathsf{VPD}_1, \mathsf{VPD}_2, \mathsf{VPD}_3$ *to denote the zkVPD protocols for input layer, masking polynomials g_i and R_i described in Construction 2.*

- $\mathcal{G}(1^\lambda, n, S)$*: run* $(\mathsf{pp}_1, \mathsf{vp}_1) \leftarrow \mathsf{VPD}_1.\mathsf{KeyGen}(1^\lambda, \log n)$, $(\mathsf{pp}_2, \mathsf{vp}_2) \leftarrow \mathsf{VPD}_2.\mathsf{KeyGen}(1^\lambda, \log S)$, $(\mathsf{pp}_3, \mathsf{vp}_3) \leftarrow \mathsf{VPD}_3.\mathsf{KeyGen}(1^\lambda)$. *Output* $\mathsf{pk} = (\mathsf{pp}_1, \mathsf{pp}_2, \mathsf{pp}_3)$ *and* $\mathsf{vk} = (\mathsf{vp}_1, \mathsf{vp}_2, \mathsf{vp}_3)$.
- $\langle \mathcal{P}(\mathsf{pk}, w), \mathcal{V}(\mathsf{vk})\rangle(x)$*: Let C be a layered arithmetic circuit over $\mathbb{F}$ with d layers, input x and witness w such that $|x| + |w| \leq n$, $|C| \leq S$ and $C(x; w) = 1$. Without loss of generality, assume $|w|/|x| = 2^m - 1$ for some $m \in \mathbb{N}$.*

  1. *$\mathcal{P}$ selects a random bivariate polynomial R_d with variable degree 2 and commits to the input of C by sending* $\mathsf{com}_d \leftarrow \mathsf{VPD}_1.\mathsf{Commit}(\dot{V}_d, r_V, r_R, \mathsf{pp}_1)$ *to $\mathcal{V}$, where $\tilde{V}_d$ is the multilinear extension of array $(x; w)$ and $\dot{V}_d = \tilde{V}_d + R_d$*
  2. *$\mathcal{V}$ runs* $\mathsf{VPD}_1.\mathsf{CheckComm}(\mathsf{com}_d, \mathsf{vp}_1)$. *If it outputs* reject, *$\mathcal{V}$ aborts and outputs* reject.
  3. *$\mathcal{P}$ and $\mathcal{V}$ execute Step 1-5 of the zero knowledge GKR protocol in Construction 2, with the zkVPDs instantiated with* VPD_2 *and* VPD_3. *If Construction 2 rejects, $\mathcal{V}$ outputs* reject *and aborts. Otherwise, by the end of this step, $\mathcal{V}$ receives two claims of $\dot{V}_d$ at $u^{(d)}$ and $v^{(d)}$.*
  4. *$\mathcal{P}$ runs* $(y_1, \pi_1) \leftarrow \mathsf{VPD}_1.\mathsf{Open}(\dot{V}, r_V, r_R, u^{(d)}, \mathsf{pp}_1)$, $(y_2, \pi_2) \leftarrow \mathsf{VPD}_1.\mathsf{Open}(\dot{V}, r_V, r_R, v^{(d)}, \mathsf{pp}_1)$ *and sends y_1, π_1, y_2, π_2 to $\mathcal{V}$.*
  5. *$\mathcal{V}$ runs* $\mathsf{Verify}(\mathsf{com}_d, u^{(d)}, y_1, \pi_1, \mathsf{vp}_1)$ *and* $\mathsf{Verify}(\mathsf{com}_d, v^{(d)}, y_2, \pi_2, \mathsf{vp}_1)$ *and output* reject *if either check fails. Otherwise, $\mathcal{V}$ checks $\dot{V}_d(u^{(d)}) = y_1$ and $\dot{V}_d(v^{(d)}) = y_2$, and rejects if either fails.*
  6. *$\mathcal{V}$ computes the multilinear extension of input x at a random point $r_x \in \mathbb{F}^{\log|x|}$ and sends r_x to $\mathcal{P}$.*
  7. *$\mathcal{P}$ pads r_x to $r'_x \in \mathbb{F}^{\log|x|} \times 0^{\log|w|}$ with $\log|w|$ 0s and sends $\mathcal{V}$* $(y_x, \pi_x) \leftarrow \mathsf{VPD}_1.\mathsf{Open}(\tilde{V}_d, r_V, r_R, r'_x, \mathsf{pp}_1)$. *$\mathcal{V}$ checks* $\mathsf{Verify}(\mathsf{com}_d, r'_x, y_x, \pi_x, \mathsf{vp}_1)$ *and y_x equals the evaluation of the multilinear extension on x. $\mathcal{V}$ outputs* reject *if the checks fail. Otherwise, $\mathcal{V}$ outputs* accept.

2.5 Implementation and Evaluation

Software. We fully implement Libra, our new zero knowledge proof system in C++. There are around 3000 lines of code for the zkGKR protocol, 1000 lines for the zkVPD protocol and 700 lines for circuit generators. Our system provides an interface to take a generic layered arithmetic circuit and turn it into a zero knowledge proof. We implement a new class for large integers named u512, and use it together with the GMP[Gnu] library for large numbers and field arithmetic. We use the ate-pairing[Ate] library on a 254-bit elliptic curve for the bilinear map used in zkVPD. We plan to open-source our system.

Hardware. We run all of the experiments on Amazon EC2 c5.9xlarge instances with 70GB of RAM and Intel Xeon platinum 8124m CPU with 3GHz virtual core. Our current implementation is not parallelized and we only use a single CPU core in the experiments. We report the average running time of 10 executions.

More gate types with no overhead. We first present a concrete optimization we developed during the implementation to support various types of gates with no extra overhead. In our protocol in Section 2.3 and 2.4, we only consider addition and multiplication gates, as they are enough to represent all arithmetic circuits. However, in practice, the size of the circuit can be reduced significantly if we introduce other types of gate. The GKR protocol still works with these new gates, but they incur an overhead on the prover time for a circuit of the same size. Therefore, in prior work such as [WHGSW16; ZGKPP18], this is considered as a trade-off.

Our protocol supports any gate with fan-in ≤ 2 and degree ≤ 2 with no overhead on the prover. Recall that in the GKR protocol, the values in layer i is represented as a sumcheck of values in layer $i+1$ and the wiring predicates, as shown in Equation 2.2. With a set of gate types $\mathcal{T}$, we can write the polynomial in the sum as

$$\sum_{j \in \mathcal{T}} \tilde{gate}_i^{(j)}(g, x, y) G_i^{(j)}(\tilde{V}_{i+1}(x), \tilde{V}_{i+1}(x)),$$

where $G_i^{(j)}()$ is the computation of gate type j (e.g., for addition gates, $G_i^{(j)}(\tilde{V}_{i+1}(x), \tilde{V}_{i+1}(x)) = \tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(x)$). As the gates have fan-in ≤ 2 and degree ≤ 2, $G_i^{(j)}$ has up to 2 variables and total degree at most 2 for all j. Therefore, each $G_i^{(j)}$ can be expressed explicitly as $a_0 + a_1\tilde{V}_{i+1}(x) + a_2\tilde{V}_{i+1}(y) + a_3\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y) + a_4\tilde{V}_{i+1}(x)^2 + a_5\tilde{V}_{i+1}(y)^2$, at most 6 nonzero monomials. The prover can then combine all the wiring predicates $\tilde{gate}_i^{(j)}(g, x, y)$ for the same monomial through a summation. With this approach, when generating the proof in Algorithm 4 and 5, the prover only allocates one array for each monomial, and initializes all 6 arrays with one scan through all the gates in Init_PhaseOne and Init_PhaseTwo. In this way, the prover time remains the same regardless of the number of gate types.

In our experiments, useful types of gates include subtraction, relay, multiply by constant, $x(1-x)$ for binary check, `NOT,AND,OR,XOR`, etc.

2.5.1 Improvements on GKR protocols

In this section, we compare the performace of our new GKR protocol with linear prover time with all variants of GKR in the literature on different circuits.

Methodology and benchmarks. For fair comparisons, we re-implement all of these variants in C++ with the same libraries. The variants include: (1) $O(C)$ for regular circuits, proposed in [Tha13a], where the two inputs of a gate can be described by two mapping functions with constant size in constant time. See [Tha13a] for the formal definition of regular circuits. (2) $O(C + C' \log C')$ for data-parallel circuits with a small copy of size C', proposed in [Wah+17]. (3) $O(C \log C')$ for circuits with non-connected different copies of size C', proposed in [ZGKPP18]. (4) $O(C \log C)$ for arbitrary circuits, proposed in [CMT12].

We compare our GKR protocol to these variants on the benchmarks below:

- **Matrix multiplication:** $\mathcal{P}$ proves to $\mathcal{V}$ that it knows two matrices whose product equals a public matrix. The representation of this function with an arithmetic circuit is highly regular[6]. We evaluate on different dimensions from 4×4 to 256×256 and the elements in the matrices are 32-bit integers.

[6] We use the circuit representation of matrix multiplication with $O(n^3)$ gates for fair comparisons, not the special protocol proposed in [Tha13a].

Matrix multiplication	Matrix size	4x4	16x16	64x64	256x256
	[Tha13a]	0.0003s	0.006s	0.390s	29.0s
	Ours	0.0004s	0.014s	0.788s	50.0s
Image scaling	#pixels	112x112	176x176	560x560	1072x1072
	[Wah+17]	0.445s	0.779s	7.54s	29.2s
	Ours	0.337s	1.25s	19.8s	79.2s
Image scaling with different parameters	#pixels	112x112	176x176	560x560	1072x1072
	[ZGKPP17c]	5.45s	21.8s	348s	1441s
	Ours	0.329s	1.22s	19.3s	77.2s
Random circuit	#gates per layer	2^8	2^{12}	2^{16}	2^{20}
	[CMT12]	0.008s	0.179s	3.79s	83.1s
	Ours	0.002s	0.039s	0.635s	10.8s

Table 2.2: Prover time of our linear GKR and previous GKR variants.

- **Image scaling:** It computes a low-resolution image by scaling from a high-resolution image. We use the classic Lanczos re-sampling[Tur90] method. It computes each pixel of the output as the convolution of the input with a sliding window and a kernel function defined as: $k(x) = \text{sinc}(x)/\text{sinc}(ax), \text{if} -a < x < a; k(x) = 0,$ otherwise, where a is the scaling parameter and $\text{sinc}(x) = \sin(x)/x$. This function is data parallel, where each sub-circuit computes the same function to generate one pixel of the output image. We evaluate by fixing the window size as 16×16 and increase the image size from 112x112 to 1072x1072. The pixels are 8-bit integers for greyscale images.

- **Image scaling of different parameters:** It is the same computation as above with different scaling parameters in the kernel function for different pixels. The circuit of this function consists of different sub-copies. We evaluate it with the same image sizes as above.

- **Random circuit:** It is randomly generated layered circuit. We randomly sample the type of each gate, input value and the wiring patterns. We fix the depth as 3 and increase the number of gates per layer from 2^8 to 2^{20}.

To be consistent with the next section, all the protocols are executed on a 254-bit prime field. This does not affect the comparison at all, as all the protocols are in the same field. In Table 2.2, we report the prover time of the protocols. The proof size and the verification time of all the variants are similar.

Results. As shown in Table 2.2, the performance of our GKR protocol is comparable to those special protocols for structured circuits, and much better than the state-of-the-art on generic circuits. For example, for matrix multiplication, our protocol is slower by 1.3-$2.4\times$, because the protocol in [Tha13a] writes the wiring of matrix multiplication explicitly and does not need to compute $\tilde{add}$ and $\tilde{mult}$. For image scaling, our protocol is slower by 2.5-$4\times$. This gap would become even smaller when the size of each sub-copy is larger. Here we use a small 16×16 block, while the number of copies is 49-4489.

For image scaling with different parameters and generic random circuits, our protocol has a speedup of 4-8×, and the speedup will increase with the scale of the circuits, as indicated by the complexity.

Besides the speedup on complicated circuits, a significant advantage of our new GKR protocol is on the prover interface of the system. In prior work such as [Wah+17; ZGKPP18], as the protocols are particularly efficient for structured circuits, the circuits must be represented as small copies and the numbers of each copy. Even worse, the structure is explored per layer of the circuit, making the numbers of each copy potentially different in different layers. (E.g., 6 gates may be considered 3 copies with 2 gates and 2 copies with 3 gates in two different layers for efficiency purposes.) This constraint makes the interface of these systems hard to use and generalize. Our result gives a unified solution for arbitrary circuits, and it is the main reason that our prover can take the description of any layered arithmetic circuit potentially generated by other tools like Verilog.

2.5.2 Comparing to Other ZKP Schemes

In this section, we show the performance of Libra as a whole and compare it with several state-of-the-art zero knowledge proof systems.

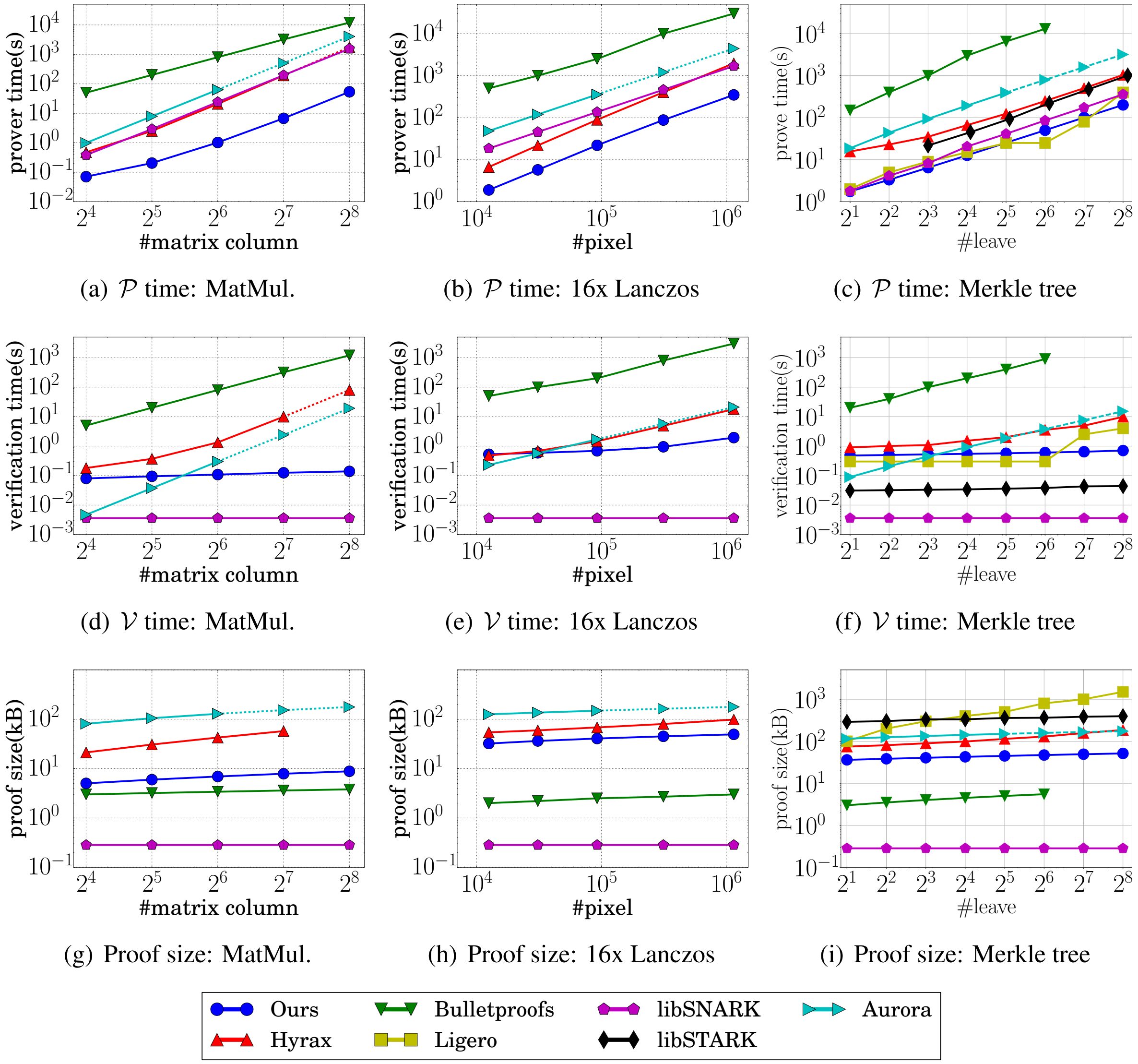
Methodology. We compare with the following systems: libSNARK [BSCTV14c], Ligero[AHIV17], libSTARK[BSBHR19], Hyrax[WTSTW18], Bulletproofs[BBBPWM18] and Aurora [BSCRSVW19]. See Section 2.1 for more explanations of these systems and their asymptotic.

- libSNARK: We use jsnark [Jsn] to write the circuits (rank one constraint system (R1CS)), which compiles them to zero knowledge proofs using the libSNARK backend [Liba].
- Ligero: As the system is not open-source, we use the same number reported in [AHIV17] on computing hashes.
- libSTARK: After communications with the authors of [BSBHR19], we obtain numbers for proving the same number of hashes in the 3rd benchmark below from the authors. The experiments are executed on a server with 512GB of DDR3 RAM (1.6GHz) and 16 cores (2 threads per core) at speed of 3.2GHz.
- Hyrax: We use the open-source implementation of the system at [Hyr].
- Bulletproofs: We use the system re-implemented by [WTSTW18] at [Hyr].
- Aurora: As a recently accepted paper, the system is not available and we extrapolate its performance using the numbers reported in the paper [BSCRSVW19] for circuits with $2^{10} - 2^{20}$ R1CS constrains.

Benchmarks. We evaluate the systems on three benchmarks: matrix multiplication, image scaling and Merkle Tree[Mer87], which are used in [WTSTW18]. Matrix multiplication and image scaling are the same as explained in Section 2.5.1. In the third benchmark, $\mathcal{P}$ proves to $\mathcal{V}$ that it knows the value of the leaves of a Merkle tree[Mer87] that computes to a public root value[BEGKN94]. We use SHA-256 for the hash function. We implement it with a flat circuit where each sub-computation is one instance of the hash function. The consistency of the input and output of corresponding hashes are then checked by the circuit. There are $2M - 1$ SHA256 invocations for a Merkle tree with M leaves. We increase the number of leaves from 16 to 256. We use the SHA-256 implemented by jsnark [Jsn] in R1CS format to run libSNARK and estimate Aurora, and we use the SHA-256 arithmetic circuit implemented by Hyrax to run Hyrax, Bulletproofs and Libra. We only show the performance of Ligero and libSTARK on the third benchmark.

We report the prover time, proof size and verification time in Figure 2.1.

Prover time. As shown in Figure 2.1(a)(b)(c), the prover in Libra is the fastest among all systems in all three benchmarks we tested. Ligero is one of the best existing ZKP systems on prover time as it is purely based on



(a) $\mathcal{P}$ time: MatMul.

(b) $\mathcal{P}$ time: 16x Lanczos

(c) $\mathcal{P}$ time: Merkle tree

(d) $\mathcal{V}$ time: MatMul.

(e) $\mathcal{V}$ time: 16x Lanczos

(f) $\mathcal{V}$ time: Merkle tree

(g) Proof size: MatMul.

(h) Proof size: 16x Lanczos

(i) Proof size: Merkle tree

Figure 2.1: Comparisons of prover time, proof size and verification time between Libra and existing zero-knowledge proof systems.

symmetric key operations. Comparing to Ligero, the prover time of Libra is $1.15\times$ faster on a Merkle tree with 2 leaves and $2\times$ faster with 256 leaves. Comparing to other systems, Libra improves the prover time by $3.4-8.9\times$ vs. Hyrax, $7.1-16.1\times$ vs. Aurora, $10.1-12.4\times$ vs. libSTARK and $65-166\times$ vs. Bulletproof.

Libra is also faster than libSNARK on general circuits by $5-10\times$, as shown in Figure 2.1(a) and 2.1(b). The performance of Libra is comparable to libSNARK on Merkle trees in Figure 2.1(c). This is because (1) most values in the circuit of SHA256 are binary, which is friendly to the prover of libSNARK as the time of exponentiation is proportional to the bit-length of the values; (2) The R1CS of SHA256 is highly optimized by jsnark [Jsn] and real world products like Zcash [Ben+14]. There are only 26,000 constrains in one hash.

In the arithmetic circuit used by Libra, there are 60,000 gates with 38,000 of them being multiplication gates. Even so, Libra is still as fast as libSNARK on a Merkle tree with 2 leaves and 2× faster with 256 leaves. We plan to further optimize the implementation of SHA256 as an arithmetic circuit in the future.

The gap between Libra and other systems will become bigger as the size of the circuit grows, as the prover time in these systems (other than Bulletproof) scales quasi-linearly with the circuit size. The evaluations justify that the prover time in Libra is both optimal asymptotically, and efficient in practice.

Verification time. Figure 2.1(d)(e)(f) show the verification time. Our verifier is much slower than libSNARK and libSTARK, which runs in 1.8ms and 28-44ms respectively in all the benchmarks.

Other than these two systems, the verification time of Libra is faster, as it grows sub-linearly with the circuit size. In particular, our verification time ranges from $0.08 - 1.15$s in the benchmarks we consider. In Figure 2.1(f), the verification time of Libra is 8× slower than Aurora when $M = 2$, and 15× faster when $M = 256$. Libra is 2.5× slower than Ligero with $M = 2$ and 4× faster with $M = 256$. Comparing to Hyrax and Bulletproof, our verification is $1.2 - 9\times$ and $27 - 900\times$ faster respectively. Again, the gap increases with the scale of the circuits as our verification is succinct.

Proof size. We report the proof size in Figure 2.1(g)(h)(i). Our proof size is much bigger than libSNARK, which is 128 bytes for all circuits, and Bulletproof, which ranges in 2-5.5KBs. The proof size in Libra is in the range of 30-60KBs, except for the matrix multiplications where it reduces to 5-9KBs. This is better than Aurora, Hyrax and libSTARK, which also have poly-logarithmic proof size to the circuit. Finally, the proof size in Ligero is $O(\sqrt{C})$ and grows to several megabytes in practice.

Setup time. Among all the systems, only Libra and libSNARK require trusted setup. Thanks to the optimization described in the beginning of this section, it only takes 202s to generate the public parameters in our largest instance with $n = 2^{24}$. Libra only needs to perform this setup once and it can be used for all benchmarks and all circuits with no more inputs. libSNARK requires a per-circuit setup. For example, it takes 1027s for the Merkle tree with 256 leaves, and takes 210s for 64×64 matrix multiplications.

2.5.3 Discussions

In this section, we discuss some potential improvements for Libra.

Improving verification time. As shown in the experiments above, the verification time in Libra is already fast in practice compared to other systems, yet it can be further improved by 1-2 orders of magnitude.

Within the verification of Libra, most of the time (more than 95% in the evaluations above) is spent on our zkVPD protocols using bilinear pairings. In our current protocol, we use the pairing-based zkVPD both for the input layer and for the masking polynomials g_i, R_i in each intermediate layer. Although the masking polynomials are small, the verification of our zkVPD still requires $O(s_i)$ pairings per layer for g_i, which is asymptotically the same as the input layer. For example, for the SHA256 circuit with 12 layers, the zkVPD verification of each g_i is around 46ms, $\frac{1}{16}$ of the total verification time.

However, there are many zkVPD candidates for these masking polynomials. Recall that the size of g_i is only $O(s_i)$, logarithmic on the size of the circuit. We could use any zkVPD with up to linear commitment size, prover time, proof size and verification time while still maintaining the asymptotic complexity of Libra. The only property we need is zero knowledge. Therefore, we can replace our pairing-based zkVPD with any of the zero knowledge proof systems we compare with as a black-box. Ligero and Aurora are of particular interest as their verification requires no cryptographic operations. If we use the black-box of these two systems for the zkVPD of g_i, R_i, the prover time and proof size would be affected minimally, and the verification time would be improved by almost d times, as only the zkVPD of the input layer requires pairings after the change. This is a 1-2 orders-of-magnitude improvement depending on the depth of the circuit. In addition, it also removes the trusted setup in the zkVPD for the masking polynomials. We plan to integrate this approach into our system when the implementations of Ligero and Aurora become available.

Removing trusted setup. After the change above, the only place that requires trusted setup is the zkVPD for the input layer. However, replacing our pairing-based zkVPD with other systems without trusted setup may affect the succinctness of our verification time on structured circuits. For example, using Ligero, Bulletproof and Aurora as a black-box would increase the verification time to $O(n)$, and using Hyrax would increase the proof size and verification time to $O(\sqrt{n})$. Using libSTARK may keep the same complexity, as polynomial evaluation is a special function with short description, but the prover time and memory usage is high in STARK as shown in the experiments. Designing an efficient zkVPD protocol with logarithmic proof size and verification time without trusted setup is left as an interesting future work and we believe this paper serves as an important step towards the goal of efficient succinct zero knowledge proof without trusted setup.

Chapter 3

Orion: Zero Knowledge Proof with Linear Prover Time

Zero-knowledge proof is a powerful cryptographic primitive that has found various applications in the real world. However, existing schemes with succinct proof size suffer from a high overhead on the proof generation time that is super-linear in the size of the statement represented as an arithmetic circuit, limiting their efficiency and scalability in practice. In this paper, we present Orion, a new zero-knowledge argument system that achieves $O(N)$ prover time of field operations and hash functions and $O(\log^2 N)$ proof size. Orion is concretely efficient and our implementation shows that the prover time is 3.09s and the proof size is 1.5MB for a circuit with 2^{20} multiplication gates. The prover time is the fastest among all existing succinct proof systems, and the proof size is an order of magnitude smaller than a recent scheme proposed in Golovnev et al. 2021.

In particular, we develop two new techniques leading to the efficiency improvement. (1) We propose a new algorithm to test whether a random bipartite graph is a lossless expander graph or not based on the densest subgraph algorithm. It allows us to sample lossless expanders with an overwhelming probability. The technique improves the efficiency and/or security of all existing zero-knowledge argument schemes with a linear prover time. The testing algorithm based on densest subgraph may be of independent interest for other applications of expander graphs. (2) We develop an efficient proof composition scheme, code switching, to reduce the proof size from square root to polylogarithmic in the size of the computation. The scheme is built on the encoding circuit of a linear code and shows that the witness of a second zero-knowledge argument is the same as the message in the linear code. The proof composition only introduces a small overhead on the prover time.

3.1 Introduction

Zero-knowledge proof (ZKP) allows a *prover* to convince a *verifier* that a statement is valid, without revealing any additional information about the prover's secret witness of the statement. Since it was first introduced in the seminal paper by Goldwasser, Micali and Rackoff [GMR89], ZKP has evolved from a purely theoretical interest to a concretely efficient cryptographic primitive, leading to many real-world applications in practice. It has been widely used in blockchains and cryptocurrencies to achieve privacy (Zcash [Ben+14; Zca]) and to improve scalability (zkRollup [Zkr]). More recently, it also found applications in zero-knowledge machine learning [ZFZS20; LKKO20; LXZ21; FQZDC21; WYXKW21], zero-knowledge program analysis [FDNZ21], and zero-knowledge middlebox [GAZBW22].

There are three major efficiency measures in ZKP: the overhead of the prover to generate the proof, which is referred to as the *prover time*; the total communication between the prover and the verifier, which is called the *proof size*; and the time to verify the proof, which is called the *verifier time*. Despite its recent progress, the efficiency of ZKP is still not good enough for many applications. In particular, the prover time is one of the major bottlenecks preventing existing ZKP schemes from scaling to large statements. As pointed out by Golovnev et al. in [GLSTW], to prove a statement that can be modeled as an arithmetic circuit with N gates, existing schemes with succinct proof size either perform a fast Fourier transform (FFT) due to the Reed-Solomon code encodings or polynomial interpolations, or a multi-scalar exponentiation due to the use of discrete-logarithm assumptions or bilinear maps, over a vector of size $O(N)$. The former takes $O(N \log N)$ field additions and multiplications and the latter takes $O(N \log |\mathbb{F}|)$ field multiplications, where $|\mathbb{F}|$ is the size of the finite field. With the Pippenger's algorithm [Pip76], the complexity of the multi-scalar exponentiation can be improved to $O(N \log |\mathbb{F}| / \log N)$, which is still super-linear as $\log |\mathbb{F}| = \omega(\log N)$ to ensure security. These operations are indeed the dominating cost of the prover time both asymptotically and concretely. See Section 3.1.3 for more discussions about existing ZKP schemes categorized by the underlying cryptographic techniques.

The only exceptions in the literature are schemes in [BCGGHJ17; BCG20; BCL22; GLSTW]. Bootle et al. [BCGGHJ17] proposed the first ZKP scheme with a prover time of $O(N)$ field operations and a proof size of $O(\sqrt{N})$ using a linear-time encodable error-correcting code. The proof size is later improved to $O(N^{1/c})$ for any constant c via a tensor code in [BCG20], and then to $\mathsf{polylog}(N)$ via a generic proof composition with a probabilistic checkable proof (PCP) in [BCL22]. These schemes are mainly for theoretical interests and do not have implementations with good concrete efficiency. Recently, Golovnev et al. [GLSTW] proposed a ZKP scheme based on the techniques in [BCG20] by instantiating the linear-time encodable code with a randomized construction. However, the security guarantee (soundness error) is only inverse polynomial in the size of the circuit, instead of negligible. Moreover, the proof size of the implemented scheme is $O(\sqrt{N})$ (more details are presented in Section 3.1.3). Therefore, the following question still remains open:

Can we construct a concretely efficient ZKP scheme with $O(N)$ prover time and $\mathsf{polylog}(N)$ proof size?

3.1.1 Our Contributions

We answer the question above positively in this paper by proposing a new ZKP scheme. In particular, our contributions include:

- First, we propose a random construction of the linear-time encodable code that has a constant relative distance with overwhelming probability. Such a code was used in all existing linear-time ZKP schemes [BCGGHJ17; BCG20; BCL22; GLSTW] and thus our new construction also improves their

	Prover time	Proof size	Verifier time*	Soundness	Concrete efficiency
[BCGGHJ17]	$O(N)$	$O(\sqrt{N})$	$O(N)$	$\mathsf{negl}(N)$	✗
[BCG20]	$O(N)$	$O(N^{1/c})$	$O(N^{1/c})$	$\mathsf{negl}(N)$	✗
[BCL22]	$O(N)$	$\mathsf{polylog}(N)$	$\mathsf{polylog}(N)$	$\mathsf{negl}(N)$	✗
[GLSTW]	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\frac{1}{\mathsf{poly}(N)})$	✓
our scheme	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$	$\mathsf{negl}(N)$	✓

Table 3.1: Comparison to existing ZKP schemes with linear prover time. N is the size of the circuit/R1CS and $c \geq 2$ is a constant. * The verifier time is achieved in the preprocessing setting. In addition, the scheme in [GLSTW] achieves $O(\sqrt{N})$ verifier for structured circuits in the non-preprocessing setting.

efficiency. The key technique is a new algorithm to test whether a random graph is a good expander graph based on the densest sub-graph algorithm, which may be of independent interest for other applications of expander graphs [SZT02].

- Second, we propose a new reduction that achieves a proof size of $O(\log^2 N)$ efficiently. Our technique is a proof composition named "code switching" proposed in [RR20]. We develop an efficient instantiation using the encoding circuit of the linear-time encodable code, which reduces the proof size of the schemes in [BCG20; GLSTW] from $O(\sqrt{N})$ to $O(\log^2 N)$ with a small overhead on the prover time.
- Finally, we implement our new ZKP scheme, Orion, and evaluate it experimentally. On a circuit with 2^{20} gates (rank-1-constraint-system (R1CS) with 2^{20} constraints), the prover time is 3.09s, the proof size is 1.5 MBs and the verifier time is 70ms. Orion has the fastest prover time among all existing ZKP schemes in the literature. The proof size is 6.5× smaller than the system in [GLSTW]. The scheme is plausibly post-quantum secure and can be made non-interactive via the Fiat-Shamir heuristic [FS86].

Table 3.1 shows the comparison between our scheme and existing schemes with linear prover time and succinct proof size.

Verifier time. The verifier time in Table 3.1 is achieved in the preprocessing setting (holographic proofs [CHMMVW20]). As all the schemes do not have a trusted setup, their verifier time is $O(N)$ in the worst case, as the verifier has to read the description of the circuit/R1CS. In the preprocessing setting, the verifier time becomes sublinear with the commitment of an indexer describing the circuit. This is the best that can be achieved, and our scheme has a $O(\log^2 N)$ verifier time in this setting using the techniques in [Set20]. In addition, the scheme in [GLSTW] can also achieve a verifier time of $O(\sqrt{N})$ in the non-preprocessing setting if the circuit/R1CS is structured, i.e., the description of the circuit can be computed in sublinear time. Our scheme has an $O(\sqrt{N})$ verifier in this case, but not $O(\log^2 N)$. This is because the encoding circuit we use in the proof compositing is of size $O(\sqrt{N})$ and is not structured.

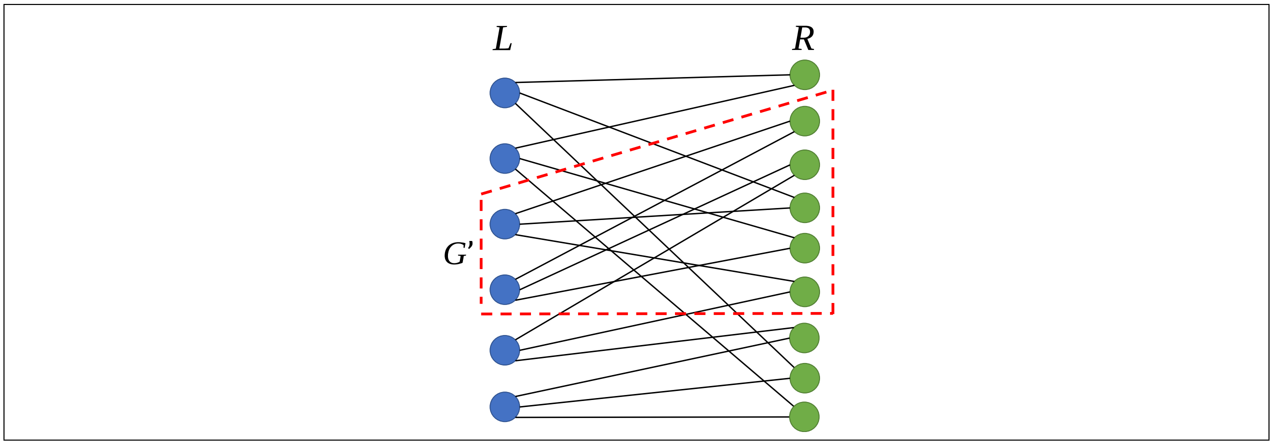


Figure 3.1: An example of lossless expander. $k = 6, k' = 9, g = 3, \delta = 1, \epsilon = \frac{1}{6}$.

3.1.2 Technical Overview

Testing expander graphs via densest sub-graph. All existing ZKP schemes with linear prover time and succinct proof size [BCGGHJ17; BCG20; BCL22; GLSTW] use linear-time encodable codes with a constant relative distance proposed in [Spi96; DI14; GLSTW], which in turn all rely on the existence of good expander graphs. In a good expander graph, any subset of vertices expands to a large number of neighbors. Figure 3.1 shows an example of a bipartite graph where any subset of vertices on the left of size 2 expands to at least 5 vertices on the right. See Section 3.2.1 for formal definitions and constructions. However, how to construct such good expanders remain unclear in practice. Explicit constructions [CRVW02] have large hidden constants in the complexity and thus are not practical. A random graph tends to have good expansion, but the probability that a random graph is not a good expander is inverse polynomial in the size of the graph. The code constructed from a non-expanding graph does not have a good minimum distance, making the ZKP scheme insecure. Therefore, a randomly sampled graph is not good for cryptographic applications.

In this paper, we propose a new algorithm to efficiently test whether a random graph is a good expander or not. With the new testing algorithm, we are able to re-sample the random graph until it passes the test, obtaining a good expander with an overwhelming probability and boosting the soundness error of the ZKP scheme to be negligible. The testing algorithm is based on the densest sub-graph algorithm [Gol84]. The density of a graph $G = (V, E)$ is defined as the number of edges divided by the number of vertices $\frac{|E|}{|V|}$, and the densest sub-graph is simply the sub-graph in a graph with the maximum density. We observe that a good expander graph tends to have a small maximum density. This is because assuming the degree g of each vertex is a constant, e.g. $g = 3$ for all vertices on the left in Figure 3.1, given any subset of vertices of size s in the graph, the total number of edges is fixed as $|E| = gs$ in the sub-graph defined by this subset and its neighbors. For example, any two vertices on the left in Figure 3.1 as highlighted always have 6 outgoing edges. Then we differentiate two cases:

- In a good expander graph, any subset expands to a large number of neighbors, thus the total number of vertices in this sub-graph is large. Therefore, the density of any sub-graph is small;
- In contrast, if the graph is not a good expander, there is at least one subset that does not expand. Taking the sub-graph defined by this subset and its neighbors, again the number of edges is fixed, while the

number of vertices is small. Therefore, the density of this sub-graph is large, which will be detected by the densest subgraph algorithm.

This observation gives us a way to differentiate good expanders. To the best of our knowledge, we are the first to make the connection between expander and the densest subgraph problem.

The real testing algorithm involves random sampling and repeating the densest sub-graph algorithm because of additional conditions of the expander. The formal algorithm, theorem and proofs are presented in Section 3.3.

Proof composition via code-switching. With the expander graph sampled above and the corresponding linear code, we are able to build efficient ZKP schemes following the approaches in [BCGGHJ17; BCG20; GLSTW]. However, the proof size is $O(N^{1/c})$ instead of $\mathsf{polylog}(N)$. To reduce the proof size, a common technique in the literature is proof composition. Instead of sending the proof directly to the verifier, the prover uses a second ZKP scheme to show that the proof of the first ZKP is indeed valid. In particular, in [BCGGHJ17; BCG20; GLSTW], the proof consists of several codewords of the linear-time encodable code, and the checks can be represented as inner products between the messages in the codewords and some public vectors.

Unfortunately, we do not have a second ZKP scheme based on the linear-time encodable code with a $\mathsf{polylog}(N)$ proof size to prove inner products. If we had it, we would already be able to build a ZKP scheme with $\mathsf{polylog}(N)$ proof size in the first place. Instead, we rely on the fact that the proof consists of the codewords of the linear code and construct the second ZKP scheme as follows. One component of the second ZKP scheme is the *encoding circuit* of the linear-time encodable code. It takes the witness of the second ZKP scheme, encodes it and outputs several random locations of the codeword. The verifier checks that these random locations are the same as the proof of the first ZKP scheme, without receiving the entire proof. By the distance of the linear-time encodable code, we show that the witness of the second ZKP must be the same as the message in the proof of the first ZKP with overwhelming probability. After that, the other component of the second ZKP checks the inner product relationship modeled as an arithmetic circuit. A similar proof composition was also used in [RR20]. We view our approach using the encoding circuit as a variant of the proof composition that is efficient in practice, and thus we inherit the name “code switching” from [RR20].

With this idea, we can use any general-purpose ZKP scheme on arithmetic circuits with a $\mathsf{polylog}(N)$ proof size as the second ZKP scheme in the proof composition. The size of this circuit is only $O(\sqrt{N})$, thus the second ZKP does not introduce any overhead on the prover time as long as its prover time is no more than quadratic. In our construction, we use the ZKP scheme in [ZXZS20] as the second ZKP. The scheme is based on the interactive oracle proofs (IOP) and the witness is encoded using the Reed-Solomon code. Therefore, the technique is called code switching. The formal protocols are presented in Section 2.4.

3.1.3 Related Work

Zero-knowledge proof was introduced in [GMR89] and generic constructions based on PCPs were proposed by Kilian [Kil92] and Micali [Mic00] in the early days. Driven by various applications mentioned in the introduction, there has been significant progress in efficient ZKP protocols and systems. Categorized by their underlying techniques, there are ZKP systems based on bilinear maps [PHGR13; BSCGTV13; BFRSBW13; BSCTV14a; Cos+15; WSRBW15; FFGKOP16; GKMMM18; MBKM19; GWC19a; CHMMVW20; KPPS20], MPC-in-the-head [GMO16; Cha+17; AHIV17; KKW18], IP [ZGKPP17d; ZGKPP17b;

WTSTW18; ZGKPP18; XZZPS19c; Zha+21a], discrete logarithm [BBBPWM18; BFS20; Set20; SL20], interactive oracle proofs (IOP) [BSCRSVW19; BSBHR19; ZXZS20; BFHVXZ20; COS20; BDFG20], and lattices [BBCDPGL18; ESLL19; BLNS20; ISW21]. As mentioned in the introduction, these schemes perform either an FFT (such as schemes based on MPC-in-the-head and IOP) or a multi-scalar exponentiation (such as schemes based on discrete-log and bilinear pairing), making the complexity of the prover time superlinear in the size of the circuit.

With the techniques proposed in [XZZPS19c; Zha+21a], the prover time of the schemes based on the interactive proofs (the GKR protocol [GKR08]) is linear if the size of the input is significantly smaller than the size of the circuit. However, the goal of this paper is to make the prover time strictly linear without such a requirement, and our polynomial commitment scheme can also be plugged into these schemes to improve their efficiency.

Schemes with linear prover time. As mentioned before, schemes in [BCGGHJ17; BCG20; BCL22; GLSTW] are the only candidates in the literature with linear prover time and succinct proof size for arithmetic circuits. They all use linear-time encodable codes based on expander graphs and our first contribution applies to all of them. Moreover, our ZKP scheme is based on the polynomial commitment in [GLSTW] and the tensor IOP in [BCG20], and we improve the proof size to $O(\log^2 N)$ through a proof composition. In fact, the scheme in [BCL22] also proposes a proof composition with the PCP in [Mie09]. However, the complexity of the PCP is polynomial time. That is why the scheme in [BCL22] has to be built on the scheme in [BCG20] with a proof size of $O(N^{1/c})$ and is not concretely efficient, while our scheme can be built on top of the efficient scheme in [GLSTW] with a proof size of $O(\sqrt{N})$.

Finally, the scheme in [GLSTW] samples a random graph to build the linear-time encodable code. The scheme achieves a soundness error of $O(\frac{1}{\mathsf{poly}(N)})$ and the authors spent great efforts calculating parameters that achieve a concrete failure probability of 2^{-100} for large circuits in practice [GLSTW, Claim 2 and Figure 2]. Our sampling algorithm provides the provable security guarantee of a negligible soundness error for their scheme. Moreover, we improve the proof size from $O(\sqrt{N})$ to $O(\log^2 N)$ efficiently, solving an open problem left in [GLSTW].

There are two recent schemes that achieve linear prover time for Boolean circuits [RR22; HR22]. We mainly focus on arithmetic circuits in this paper, but our techniques may also apply to these schemes to obtain efficient instantiations.

Schemes with linear proof size. Recently, there is a line of work constructing ZKP based on secure multiparty computation (MPC) techniques [WYKW20; DIO21; BMRS21; YSWW21] and these schemes have demonstrated fast prover time in practice. If one treats a block cipher (e.g., AES) as a constant-time operation because of the CPU instruction, these schemes indeed have a linear time prover (we are using a similar CPU instruction for the hash function SHA-256 in our scheme to achieve linear prover time). However, they have linear proof size in the size of the circuit, are inherently interactive, and are not publicly verifiable, which are not desirable in many applications. We mainly focus on non-interactive ZKP with succinct proof size and public verifiability in this paper.

Expander testing. Testing the properties of expander graphs is a deeply explored area in computer science. Many works [NS07; CS07; GR11] have proposed efficient testing algorithms without accessing the whole graph. However, these algorithms do not directly apply to our testing of lossless expander. For example, the algorithm in [NS07] based on random walks can differentiate good expanders from graphs that are far from expanders, while our scheme can differentiate whether a graph is a lossless expander or not with overwhelming probability. Of course our algorithm accesses the entire graph, which is fine in our application of linear-time encodable code. To the best of our knowledge, we are not aware of any testing algorithm with such properties.

There are also impossibility results on expander testing [KS16]. Due to different definitions of expansion, our testing algorithm cannot distinguish the cases in [KS16, Theorem 1.1] and thus it does not violate the impossibility results.

3.2 Preliminary

We use $[N]$ to denote the set $\{0, 1, 2, ..., N-1\}$. $\mathsf{poly}(N)$ means a function upper bounded by a polynomial in N with a constant degree . We use $\lambda = \omega(\log N)$ to denote the security parameter, and $\mathsf{negl}(N)$ to denote the negligible function in N, i.e. $\mathsf{negl}(N) \leq \frac{1}{\mathsf{poly}(N)}$ for all sufficiently large N and any polynomial. Some papers define $\mathsf{negl}(\lambda)$ as the negligible function. As λ is a function of N, they are essentially the same and $\mathsf{negl}(N) \leq \frac{1}{2^\lambda}$. "PPT" stands for probabilistic polynomial time. $\langle A(x), B(y)\rangle(z)$ denotes an interactive protocol between algorithms A, B with x as the input of A, y as the input of B and z as the common input.

3.2.1 Linear-Time Encodable Linear Code

Definition 3.2.1 (Linear Code). *A linear error-correcting code with message length k and codeword length n is a linear subspace $C \in \mathbb{F}^n$, such that there exists an injective mapping from message to codeword $E_C : \mathbb{F}^k \to C$, which is called the encoder of the code. Any linear combination of codewords is also a codeword. The rate of the code is defined as $\frac{k}{n}$. The distance between two codewords u, v is the hamming distance denoted as $\Delta(u, v)$. The minimum distance is $d = \min_{u,v} \Delta(u, v)$. Such a code is denoted as $[n, k, d]$ linear code, and we also refer to $\frac{d}{n}$ as the relative distance of the code.*

Generalized Spielman code. In our construction, we use a family of linear codes that can be encoded in linear time and has a constant relative distance [Spi96; DI14; GLSTW]. The code was first proposed by Daniel Spielman in [Spi96] over the Boolean alphabet. Druk and Ishai [DI14] generalized it to a finite field $\mathbb{F}$, and introduced a distance boosting technique to achieve the Gilbert-Varshamov bound[Gil52; Var57]. We only use the basic construction over $\mathbb{F}$ without the distance boosting, and thus refer to it as the generalized Spielman code in this paper. The code relies on the existence of lossless expander graphs, which is defined below:

Definition 3.2.2 (Lossless Expander [Spi96]). *Let $G = (L, R, E)$ be a bipartite graph. $0 < \epsilon < 1$ and $0 < \delta$ be some constants. The vertex set consists of L and R, two disjoint subsets, henceforth the left and right vertex set. Let $\Gamma(S)$ be the neighbor set of some vertex set S. We say G is an $(k, k'; g)$-**lossless expander** if $|L| = k, |R| = k' = \alpha k$ for some constant α, and the following property hold:*

1. *Degree: The degree of every vertex in L is g.*
2. *Expansion: $|\Gamma(S)| \geq (1-\epsilon)g|S|$ for every $S \subseteq L$ with $|S| \leq \frac{\delta|L|}{g}$.*

Intuitively speaking, a lossless expander has very strong expansion. As the degree of each left vertex is g, a set of $|S|$ left vertices have at most $g|S|$ neighbors, while the second condition requires that every set expands to at least $(1-\epsilon)g|S|$ vertices for a small constant ϵ. Meanwhile, as the right vertext set has

$|R| = \alpha k$ vertices, such an expansion is not possible if $|S| > \frac{\alpha k}{(1-\epsilon)g}$, thus there is a condition $|S| \leq \frac{\delta k}{g}$ bounding the size of S. An example is shown in Figure 3.1.

Construction of generalized Spielman code. With the lossless expander, we give a brief description of the generalized Spielman code. Let $G = (L, R, E)$ be a lossless expander with $|L| = 2^t, |R| = 2^{t-1}$. Let A_t be a $2^t \times 2^{t-1}$ matrix where $A_t[i][j] = 1$ if there is an edge i, j in G for $i \in [2^t], j \in [2^{t-1}]$; otherwise $A_t[i][j] = 0$. The generalized Spielman code is constructed as follows:

1. Let $E_C^t(x)$ be the encoder function of input length $|x| = 2^t$, and its output will be a codeword of size 2^{t+2}. We use E_C to denote the encoder function when length is clear.

2. If $|x| \leq n_0$ then directly output x, for some constant n_0.

3. Compute $m_1 = xA_t$. Each entry of m_1 can be viewed as a vertex in R, and value of each vertex is the summation of its neighbors in L. The length of m_1 is 2^{t-1}.

4. Recursively apply the encoder E_C^{t-1} on m_1, let $c_1 = E_C^{t-1}(m_1)$.

5. Compute $c_2 = c_1 A_{t+1}$.

6. Output $x \odot c_1 \odot c_2$ as the codeword of size 2^{t+2}. $\odot$ denotes concatenation.

Lemma 3.2.3 (Generalized Spielman code, [DI14])**.** *Given a family of lossless expander, that achieves* $(1 - \epsilon)g|S|$ *expansion with* $|S| \leq \frac{\delta|L|}{g}$*, for input size k, the generalized Spielman code is a* $[4k, k, \frac{\delta}{8g}k]$ *linear code over* $\mathbb{F}$*.*

The code in [GLSTW] is a variant of generalized Spielman code. In their construction, random weights are assigned to each edge of lossless expander at line 3, 5. The output at line 6 is randomized as $(x \otimes r) \odot c_1 \odot c_2$, where $\otimes$ denotes element-wise multiplication and r is a random vector.

Definition 3.2.4 (Tensor code)**.** *Let C be a $[n, k, d]$ linear code, the tensor code $C^{\otimes 2}$ of dimension 2 is the linear code in $\mathbb{F}^{n^2}$ with message length k^2, codeword length n^2, and distance nd. We can view the codeword as a $n \times n$ matrix. We define the encoding function below:*

1. *A message of length $k \times k$ is parsed as a $k \times k$ matrix. Each row of the matrix is encoded using E_C, resulting in a codeword C_1 of size $k \times n$.*

2. *Each column of C_1 is then encoded again using E_C. The result C_2 of size $n \times n$ is the codeword of the tensor code.*

3.2.2 Collision-Resistant Hash Function and Merkle Tree

Let $H : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$ be a hash function. A Merkle Tree is a data structure that allows one to commit to $l = 2^{\mathsf{dep}}$ messages by a single hash value h, such that revealing any bit of the message require $\mathsf{dep} + 1$ hash values.

A Merkle hash tree is represented by a binary tree of depth dep where l messages elements $m_1, m_2, ..., m_l$ are assigned to the leaves of the tree. The values assigned to internal nodes are computed by hashing the value of its two child nodes. To reveal m_i, we need to reveal m_i together with the values on the path from m_i to the root. We denote the algorithm as follows:

1. $h \leftarrow \mathsf{Merkle.Commit}(m_1, ..., m_l)$.
2. $(m_i, \pi_{\mathsf{i}}) \leftarrow \mathsf{Merkle.Open}(m, i)$.
3. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{Merkle.Verify}(\pi_{\mathsf{i}}, m_i, h)$.

To achieve zero-knowledge, we requires the hash function to be hiding, we formally define hiding as follows:

Hiding property The hiding property specifies that given any hash output y, it is infeasible to find an input x such that $y = H(x|r)$ is satisfied. Note, we implicitly assumes for each hash function call on input x, we will append a randomness r.

3.2.3 Zero-Knowledge Arguments

An argument system for an NP relation R is a protocol between a computationally bounded prover $\mathcal{P}$ and a verifier $\mathcal{V}$. At the end of the protocol $\mathcal{V}$ will be convinced that there exits a witness w such that $(x, w) \in R$ for some public input x. We focus on arguments of knowledge which require the prover know the witness w. We formally define zero-knowledge as follows:

Definition 3.2.5 (View)**.** *We denote by* $\mathsf{View}(\langle \mathcal{P}, \mathcal{V} \rangle(x))$ *the view of* $\mathcal{V}$ *in an interactive protocol with* $\mathcal{P}$*. Namely, it is the random variable* $(r, b_1, b_2, ..., b_n, v_1, v_2, ..., v_m)$ *where* r *is* $\mathcal{V}$*'s randomness,* $b_1, ..., b_n$ *are messages from* $\mathcal{V}$ *to* $\mathcal{P}$*, and* $v_1, ..., v_m$ *are messages from* $\mathcal{P}$ *to* $\mathcal{V}$*.*

Definition 3.2.6. *Let* $\mathcal{R}$ *be an NP relation. A tuple of algorithm* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is a zero-knowledge argument of knowledge for* $\mathcal{R}$ *if the following holds.*

- ***Correctness****. For every* pp *output by* $\mathcal{G}(1^\lambda)$ *and* $(x, w) \in R$*,*

$$\Pr[\langle \mathcal{P}(w), \mathcal{V}() \rangle(\mathsf{pp}, x) = \mathsf{accept}] = 1.$$

- ***Knowledge Soundness****. For any PPT adversary* $\mathcal{P}^*$*, there exists a PPT extractor* ε *such that for every* pp *output by* $\mathcal{G}(1^\lambda)$ *and any* x*, the following probability is* $\mathsf{negl}(N)$*:*

$$\Pr[\langle \mathcal{P}^*(), \mathcal{V}() \rangle(\mathsf{pp}, x) = \mathsf{accept}, (x, w) \notin \mathcal{R} | w \leftarrow \varepsilon(\mathsf{pp}, x, \mathsf{View}(\langle \mathcal{P}^*(), \mathcal{V}() \rangle(\mathsf{pp}, x)))]$$

- ***Zero knowledge****. There exists a PPT simulator* $\mathcal{S}$ *such that for any PPT algorithm* $\mathcal{V}^*$*,* $(x, w) \in R$*,* pp *output by* $\mathcal{G}(1^\lambda)$*, it holds that*

$$\mathsf{View}(\langle \mathcal{P}(w), \mathcal{V}^*() \rangle(x)) \approx \mathcal{S}^{\mathcal{V}^*}(\mathsf{pp}, x)$$

Where $\mathcal{S}^{\mathcal{V}^*}(x)$ *denotes that* $\mathcal{S}$ *is given oracle accesses to* $\mathcal{V}^*$*'s random tape.*

We say that $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is a* succinct *argument system if the total communication between* $\mathcal{P}$ *and* $\mathcal{V}$ *(proof size) is* $\mathsf{poly}(\lambda, |x|, \log |w|)$*.*

Definition 3.2.7 (Arithmetic circuit)**.** *An arithmetic circuit* C *over* $\mathbb{F}$ *and a set of variables* $x_1, ..., x_N$ *is a directed acyclic graph as follows:*

1. *Each vertex is called a "gate". A gate with in-degree zero is an input gate and is labeled as a variable x_i or a constant field element in $\mathbb{F}$.*

2. *Other gates have 2 incoming edges. It calculates the addition or multiplication over the two inputs and output the result.*

3. *The size of the circuit is defined as the number of gates N.*

3.2.4 Polynomial Commitment

A polynomial commitment consists of three algorithms:

- $\mathsf{PC.Commit}(\phi(\cdot))$: the algorithm outputs a commitment $\mathcal{R}$ of the polynomial $\phi(\cdot)$.
- $\mathsf{PC.Prove}(\phi, \vec{x}, \mathcal{R})$: given an evaluation point $\phi(\vec{x})$, the algorithm outputs a tuple $\langle \vec{x}, \phi(\vec{x}), \pi_{\vec{x}} \rangle$, where $\pi_{\vec{x}}$ is the proof.
- $\mathsf{PC.VerifyEval}(\pi_{\vec{x}}, \vec{x}, \phi(\vec{x}), \mathcal{R})$: given $\pi_{\vec{x}}, \vec{x}, \phi(\vec{x}), \mathcal{R}$, the algorithm checks if $\phi(\vec{x})$ is the correct evaluation. The algorithm outputs accept or reject.

Definition 3.2.8 ((Multivariate) Polynomial commitment)**.** *A polynomial commitment scheme has the following properties:*

- ***Correctness**. For every polynomial ϕ and evaluation point $\vec{x}$, the following probability holds:*

$$\Pr\left(\begin{array}{r} \mathsf{PC.Commit}(\phi) \rightarrow \mathcal{R} \\ \mathsf{PC.Prove}(\phi, \vec{x}, \mathcal{R}) \rightarrow \vec{x}, y, \pi \\ y = \phi(\vec{x}) \\ \mathsf{PC.VerifyEval}(\pi, \vec{x}, y, \mathcal{R}) \rightarrow \mathsf{accept} \end{array}\right) = 1$$

- ***Knowledge Soundness**. For any PPT adversary $\mathcal{P}^*$ with $\mathsf{PC.Commit}^*$, $\mathsf{PC.Prove}^*$, there exists a PPT extractor $\mathcal{E}$ such that the probability below is negligible:*

$$\Pr\left(\begin{array}{c} \mathsf{PC.Commit}^*(\phi^*) \rightarrow \mathcal{R}^* \\ \mathsf{PC.Prove}^*(\phi^*, \vec{x}, \mathcal{R}^*) \rightarrow \vec{x}, y^*, \pi^* : \phi^* \leftarrow \mathcal{E}(\mathcal{R}^*, \vec{x}, \pi^*, y^*) \wedge y^* \neq \phi^*(\vec{x}) \\ \mathsf{PC.VerifyEval}(\pi^*, \vec{x}, y^*, \mathcal{R}^*) \rightarrow \mathsf{accept} \end{array}\right)$$

- ***Zero-knowledge**. For security parameter λ, polynomial ϕ, any PPT adversary $\mathcal{A}$, there exists a simulator $\mathcal{S} = [\mathcal{S}_0, \mathcal{S}_1]$, we consider following two experiments:*

$\mathsf{Real}_{\mathcal{A},\phi}(\mathsf{pp})$*:*

1. $\mathcal{R} \leftarrow \mathsf{Commit}(\mathsf{pp}, \phi)$
2. $\vec{x} \leftarrow \mathcal{A}(\mathcal{R}, \mathsf{pp})$
3. $(\vec{x}, y, \pi) \leftarrow \mathsf{Prove}(\phi, \vec{x}, \mathcal{R})$
4. $b \leftarrow \mathcal{A}(\pi, \vec{x}, y, \mathcal{R})$
5. *Output* b

$\mathsf{Ideal}_{\mathcal{A},\mathcal{S}^{\mathcal{A}}}(\mathsf{pp})$*:*

1. $\mathcal{R} \leftarrow \mathcal{S}_0(1^{\lambda}, \mathsf{pp})$
2. $\vec{x} \leftarrow \mathcal{A}(\mathcal{R}, \mathsf{pp})$
3. $(\vec{x}, y, \pi) \leftarrow \mathcal{S}_1^{\mathcal{A}}(\vec{x}, \mathsf{pp})$*, given oracle access to* $y = \phi(\vec{x})$
4. $b \leftarrow \mathcal{A}(\pi, \vec{x}, y, \mathcal{R})$
5. *Output* b

For any PPT adversary $\mathcal{A}$, two experiments are identically distributed:

$$\Pr[|\mathsf{Real}_{\mathcal{A},f}(\mathsf{pp}) - \mathsf{Ideal}_{\mathcal{A},\mathcal{S}^{\mathcal{A}}}(\mathsf{pp})| = 1] \leq \mathsf{negl}(N)$$

3.3 Testing Algorithm for Lossless Expander

As explained above, the generalized Spielman code relies on the existence of lossless expanders. On one hand, there are explicit constructions of lossless expanders in the literature [CRVW02]. However, there are large hidden constants in the complexity and the constructions are not practical. On the other hand, a random bipartite graph is a lossless expander with a high probability of $1 - O(\frac{1}{\mathsf{poly}(k)})$, where k is the size of the left vertex set in the bipartite graph. However, this is not good enough for cryptographic applications.

In this section, we propose a new approach to sample a lossless expander with a negligible failure probability. The key ingredient of our approach is a new algorithm to test whether a randomly sampled bipartite graph is a lossless expander or not. We begin the section by introducing the classical randomized construction of a lossless expander and its analysis.

3.3.1 Random Construction of Lossless Expander

As defined in Definition 3.2.2, a lossless expander graph is a g-left-regular bipartite graph $G = (L, R, E)$. Wigderson et al. [HLW06, Lemma1.9] showed that a random bipartite graph is a lossless expander with a high probability. In particular, we have the following lemma:

Lemma 3.3.1 ([HLW06]). *For fixed constant parameters $g, \delta, \alpha, \epsilon$, a random g-left-regular bipartite graph is a $(k, k'; g)$-lossless-expander with probability* $1 - O(\frac{1}{\mathsf{poly}(k)})$.

Proof. Let $G = (L, R, E)$ be a random bipartite graph with k vertices on the left and $k' = O(k)$ vertices on the right, where each left vertex connects to a randomly chosen set of g vertices on the right.

Let $s = |S|$ be the cardinality of a left subset of vertices $S \subseteq L$ such that $s \leq \frac{\delta k}{g}$, and let $t = |T|$ be the cardinality of a right subset of vertices $T \subseteq R$ such that $t \leq (1 - \epsilon)gs$. Let $X_{S,T}$ be an indicator random variable for the event that all the edges from S connect to T. Then for a particular S, if $\sum_{T \in R} X_{S,T} = 0$, then the number of neighboring vertices of S must be larger than $(1 - \epsilon)gs$. Otherwise, if there exists a $T \in R$ such that $X_{S,T} = 1$, i.e., all edges from S connect to T, the graph is not a lossless expander. As the edges are sampled randomly, the probability of this *non-expanding* event is $(\frac{t}{k'})^{sg}$. Therefore, summing over all S and by the union bound, the probability of a non-expanding graph is:

$$\begin{aligned}\Pr[(\sum_{S,T} X_{S,T}) > 0] &\le \sum_{S,T} Pr[X_{S,T} = 1] = \sum_{S,T} (\frac{t}{k'})^{sg} \\ &\le \sum_{s=2}^{\frac{\delta k}{g}} \binom{k}{s}\binom{k'}{t}(\frac{t}{k'})^{sg} \le \sum_{s=2}^{\frac{\delta k}{g}} \binom{k}{s}\binom{k'}{(1-\epsilon)gs}(\frac{(1-\epsilon)gs}{k'})^{sg}\end{aligned}$$

Using the inequality $\binom{k}{s} \le (\frac{ke}{s})^s$, the probability above is

$$\begin{aligned}&\le \sum_{s=2}^{\frac{\delta k}{g}} (\frac{ke}{s})^s (\frac{k'e}{(1-\epsilon)gs})^{(1-\epsilon)gs} (\frac{(1-\epsilon)gs}{k'})^{sg} \\ &= \sum_{s=2}^{\frac{\delta k}{g}} (\frac{ke}{s})^s e^{(1-\epsilon)gs} (\frac{(1-\epsilon)gs}{k'})^{\epsilon gs} \\ &= \sum_{s=2}^{\frac{\delta k}{g}} e^{(1-\epsilon)gs+s} \cdot (\frac{k}{s})^s \cdot (\frac{(1-\epsilon)gs}{k'})^{\epsilon gs} \qquad (3.1)\end{aligned}$$

When s, ϵ, g are constants and $k' = O(k)$, $e^{(1-\epsilon)gs+s}$ is a constant, $(\frac{k}{s})^s$ is $O(\mathsf{poly}(k))$, and $(\frac{(1-\epsilon)gs}{k'})^{\epsilon gs}$ is $O(\frac{1}{\mathsf{poly}(k)})$. Therefore, the overall upper bound is at least $O(\frac{1}{\mathsf{poly}(k)})$. □ □

The derivation above shows that the probability that a random graph is not a lossless expander is upper-bounded by $O(\frac{1}{\mathsf{poly}(k)})$, which is not negligible. Furthermore, we show that the lower-bound of the non-expanding probability is also not negligible through a simple argument here.

We focus on the case where s is a constant. The number of all possible sub-graphs induced by a left subset of vertices S is at most $k'^{sg} = O(\mathsf{poly}(k))$. That is, the size of the entire probability space is bounded by a polynomial. The number of non-expanding graphs is at least 1 (e.g., all edges from S connect to a single vertex in R). Therefore, the non-expanding probability is at least $O(\frac{1}{\mathsf{poly}(k)})$.

Lossless expander in [GLSTW] As explained in Section 3.2.1, in [GLSTW], the authors extended the generalized Spielman code by adding random weights to the edges in the bipartite graph. However, the graph still needs to be a lossless expander in order to achieve a constant relative distance, and the same issue above applies to their construction. In particular, as shown by [GLSTW, Claim 2], the probability of *not* sampling a lossless expander is

$$2^{kH(15/k)+\alpha kH(19.2/(\alpha k))-15g\log\frac{\alpha k}{19.2}},$$

where $H(x) = -x\log x - (1-x)\log(1-x)$. We show that the probability above is not negligible. First, for any constant const,

$$\begin{aligned}xH(\mathsf{const}/x) &= x(-\frac{\mathsf{const}}{x}\log\frac{\mathsf{const}}{x} - (1-\frac{\mathsf{const}}{x})\log(\frac{x-\mathsf{const}}{x}) \\ &= (\mathsf{const}\log(x) - \mathsf{const}\log\mathsf{const}) + (1-\frac{\mathsf{const}}{x})\log(\frac{x-\mathsf{const}}{x}).\end{aligned}$$

By taking the limit, we have $\lim_{x\to\infty} xH(\mathsf{const}/x) = (\mathsf{const}\log(x) - \mathsf{const}\log\mathsf{const}) + 1 \times 0$. Therefore, $xH(\mathsf{const}/x) = O(\log x)$. Applying this fact to the equation above, $kH(15/k) + \alpha kH(19.2/(\alpha k)) = O(\log k)$, and $-15g\log\frac{\alpha k}{19.2} = -O(\log k)$. Therefore, $2^{kH(15/k)+\alpha kH(19.2/(\alpha k))-15g\log\frac{\alpha k}{19.2}}$ is at least $2^{-O(\log k)} = \frac{1}{\mathsf{poly}(k)}$. The failure probability is similar to the upper bound in Equation 3.1.

3.3.2 Algorithm based on Densest Sub-graph

To reduce the non-expanding probability of the random construction, we take a closer look at the equations above. Equation 3.1 shows that the probability that a random bipartite graph is a not lossless expander is upper bounded by $\frac{1}{\mathsf{poly}(k)}$. However, we observe that within the summation, the probability is actually negligible when s is large. In particular, if we decompose the summation in Equation 3.1 into two sums, one for $2 \le s \le \log\log k$, and the other for $s \ge \log\log k$, the second part is

$$\sum_{s=\log\log k}^{\frac{\delta k}{g}} e^{(1-\epsilon)gs+s} \cdot (\frac{k}{s})^s \cdot (\frac{(1-\epsilon)gs}{k'})^{\epsilon gs}. \tag{3.2}$$

Lemma 3.3.2. *Equation 3.2 is negligible if the following conditions are met:*

1. $(1-\epsilon)\delta + \frac{\delta}{g} + \frac{\delta}{g}\log(\frac{g}{\delta}) + \log(\frac{\delta}{\alpha})\epsilon\delta < -0.001$,
2. $\epsilon d > 2$.

Here -0.001 is just any small constant that is less than 0. We give a proof in Appendix 3.7. To provide an intuition on how these parameters are set, we give an example here: $\delta = \frac{1}{11}, \epsilon = \frac{7}{16}, g = 16, k' = \frac{1}{2}k$. We can verify the condition:

1. $\epsilon g = 7 > 2$.
2. $(1-\epsilon)\delta + \frac{\delta}{g} + \frac{\delta}{g}\log(\frac{g}{\delta}) + \log(\frac{\delta}{\alpha})\epsilon\delta = -0.009 < -0.001$.

Sampling lossless expander with negligible failure probability. The observation above shows that the non-expanding probability is dominated by small sub-graphs with size $2 \le s \le \log\log k$. This actually matches our lower bound in Section 3.3.1, as there are only polynomially many such sub-graphs and there exist ones that do not expand. Therefore, in order to reduce the non-expanding probability, we propose a new algorithm that detects small sub-graphs of size $s \le \log\log k$ that do not expand. The algorithm is based on the densest sub-graph problem, and we are the first to make the connection between the densest sub-graph and the lossless expander.

Definition 3.3.3 (Densest Sub-graph Problem)**.** *Let $G = (V, E)$ be an undirected graph, and let $S = (E_S, V_S)$ be a subgraph of G. The density of S is defined to be* $\mathsf{den}(S) = \frac{E_S}{V_S}$. *The densest sub-graph problem is to find S such that it maximizes* $\mathsf{den}(S)$. *We denote the maximum density by* $\mathsf{Den}(G)$.

Theorem 3.3.4. *[Gol84] For any graph $G = (V, E)$, there is a polynomial time algorithm that find the densest sub-graph $G' = (V', E')$ such that $V' \subseteq V$ and G' is the sub-graph. And $\frac{|E'|}{|V'|}$ is maximized. The running time of the algorithm is $O(|V||E|\log|E|\log|V|)$.*

We will use this algorithm as a building block of our testing algorithm. First, we define a notion of perfect expander, and then derive the density of a perfect expander.

Definition 3.3.5 (Perfect expander)**.** *Let $G = (L, R, E)$ be a bipartite graph. We say G is an $(k^*, k'; g)$-**perfect expander** if $|L| = k^*$, $|R| = k'$, the following property holds (where $\Gamma(S)$ denotes the set of neighbors of a set S in G):*

1. *Degree: every vertex $a \in L$, it has constant degree g.*
2. *Expansion: $|\Gamma(S)| \geq (1 - \varepsilon)g|S|$ for every $S \subseteq L$.*

Compared to lossless expander, the perfect expander does not have the upper bound on $|S|$ in the expansion property. Therefore, k' has to be much larger than k^*, unlike the case of lossless expander where $k' = O(k)$. Now we show that the density of a perfect expander is low:

Theorem 3.3.6. *If a bipartite graph is a perfect expander, its density is at most $\frac{g}{1+(1-\epsilon)g}$; otherwise, the density of the graph is larger than $\frac{g}{1+(1-\epsilon)g}$.*

Proof. We first show that the density of a perfect expander is at most $\frac{g}{1+(1-\epsilon)g}$. For any subset $L' \subseteq L$, we prove that among all sub-graphs that L' is the left vertex set, the graph induced by $(L', \Gamma(L'))$ has the maximum density.

To see this, suppose $V' = (L', R'), R' \neq \Gamma(L')$ has density $\frac{|E'|}{|V'|}$ that is the densest sub-graph with L' as its left vertex set.

Case 1: If there exists a vertex $y \in R', y \notin \Gamma(L')$, then there is no edge between y and L'. We can increase the density by removing y from R', as $\frac{|E'|}{|V'|-1} > \frac{|E'|}{|V'|}$. This is a contradiction. Therefore, $R' \subseteq \Gamma(L')$.

Case 2: If there exists an element $y \in \Gamma(L'), y \notin R'$, let $c \geq 1$ be the number of edges between y and L', by adding y to R', the density becomes $\frac{|E'|+c}{|V'|+1} > \frac{|E'|}{|V'|}$. This is a contradiction again and thus $\Gamma(L') \subseteq R'$.

Therefore, we have $\Gamma(L') = R'$ and $V' = (L', \Gamma(L'))$ maximizes the density among all sub-graphs with L' as the left vertex set. Let that sub-graph be G'. By the expansion property of the perfect expander, $\mathsf{den}(G') = \frac{|E'|}{|V'|} \leq \frac{|L'|g}{|L'|+(1-\epsilon)g|L'|} = \frac{g}{1+(1-\epsilon)g}$. Therefore, the maximum density $\mathsf{Den}(G) = \max_{L' \subseteq L}$ $\mathsf{den}(G') \leq \frac{g}{1+(1-\epsilon)g}$.

Next, we show that if a bipartite graph is not a perfect expander, its density is larger than $\frac{g}{1+(1-\epsilon)g}$. Let S^* be the set such that $|\Gamma(S^*)| < (1 - \epsilon)g|S^*|$, then the density of the sub-graph $G' = (V' = (S^*, \Gamma(S^*)), E')$ is $\frac{|E'|}{|V'|} > \frac{g|S^*|}{|S^*|+(1-\epsilon)g|S^*|} = \frac{g}{1+(1-\epsilon)g}$, so $\mathsf{Den}(G) \geq \mathsf{den}(G') > \frac{g}{1+(1-\epsilon)g}$. □

□

3.3.3 Testing Random Lossless Expander

Theorem 3.3.6 suggests a way to test whether a random graph is a lossless expander. As discussed in lemma 3.3.2, when $s \geq \log\log k$, the non-expanding probability is negligible. Thus, it suffices to test whether there is a sub-graph of size $s < \log\log k$ that does not expand. In particular, we are trying to distinguish the following two cases:

Algorithm 8 Distinguisher

```
Let $G = (L, R, E)$ be the random bipartite graph.

for $i \in [(\frac{g}{\delta})^{\log\log k}]$ do
    Sample a random set $L'$, where $|L'| = \frac{\delta k}{g}$.
    Run densest graph algorithm in [Gol84] on the subgraph induced by $L'$: $G^* = ((L', \Gamma(L')), E')$ to find its densest subgraph.
    if $\mathsf{Den}(G^*) > \frac{g}{1+(1-\varepsilon)g}$ then
        return FAIL
return SUCC
```

1. **Yes case**: For $G = (L, R, E), \forall S \subseteq L, |S| \leq \log\log k$, we have $|\Gamma(S)| \geq (1-\epsilon)g|S|$.
2. **No case**: For $G = (L, R, E)$, there exists a subset $S^* \subseteq L, |S^*| \leq \log\log k$, such that $|\Gamma(S^*)| < (1-\epsilon)g|S_0|$.

To distinguish these two cases, we cannot directly apply the densest sub-graph algorithm on the entire bipartite graph, because the expansion property only holds for $|S| \leq \frac{\delta k}{g}$ by Definition 3.2.2 of the lossless expander. The densest sub-graph algorithm would return a large sub-graph with $|S| > \frac{\delta k}{g}$ even if it is a lossless expander, as the density of the large sub-graph could be larger than $\frac{g}{1+(1-\epsilon)g}$ by Theorem 3.3.6.

Instead, we randomly sample sub-graphs $G^* = ((L', \Gamma(L')), E')$ with $\frac{\delta k}{g}$ vertexes in the left vertex set. If there exists a small non-expanding sub-graph with at most $\log\log k$ vertices on the left, the density of this small sub-graph is larger than $\frac{g}{1+(1-\epsilon)g}$ and the probability of it is in the sub-graph G^* is at least $(\frac{\delta}{g})^{\log\log k}$. Once it is contained in G', the densest-sub-graph algorithm will output a sub-graph with density larger than $\frac{g}{1+(1-\epsilon)}$. We will sample G^* $\frac{g}{\delta}^{\log\log k}$ times to amplify the probability. The formal algorithm is presented in Algorithm 8.

Theorem 3.3.7 (Distinguisher)**.** *Algorithm 8 achieves the following properties:*

1. *If G is a **Yes case**, then the algorithm will return* SUCC *with probability* 1.
2. *If G is a **No case**, then the algorithm will return* FAIL *with probability at least* $1 - \frac{1}{e}$.

Proof. By Theorem 3.3.6, if the random graph is in **Yes case**, then the distinguisher will always return SUCC, since for every induced sub-graph G^*, it is a perfect expander. Otherwise, if the random graph contains a subset $S_0 \subseteq L, |S_0| \leq \log\log k$ such that $|\Gamma(S_0)| < (1-\epsilon)g|S_0|$, then with probability at least $(\frac{\frac{\delta k}{g}}{k})^{\log\log k} = (\frac{\delta}{g})^{\log\log k}$, S_0 will be a subset of L' sampled by the algorithm. In this case, L' is not a perfect expander graph and by Theorem 3.3.6, $\mathsf{Den}(G^*) > \frac{g}{1+(1-\epsilon)g}$ and the algorithm will return FAIL. Since we repeat it $\frac{g}{\delta}^{\log\log n}$ times, the probability that we did not successfully sample S_0 is $(1 - (\frac{\delta}{g})^{\log\log k})^{(\frac{g}{\delta})^{\log\log k}}$. By the inequality $(1 - \frac{1}{n})^n \leq \frac{1}{e}$, we have $(1 - (\frac{\delta}{g})^{\log\log k})^{(\frac{g}{\delta})^{\log\log k}} \leq \frac{1}{e}$. □

□

By repeating the distinguisher λ times, we can amplify the detection probability of the No case to $1 - \frac{1}{e^\lambda}$. Finally, we re-sample the random graph until the distinguisher returns SUCC. The successful probability of one sampling is $1 - O(\frac{1}{\mathsf{poly}(k)})$, so the expected number of sampling is a constant. The algorithm runs $\lambda(\frac{g}{\delta})^{\log\log k}$ instances of the densest sub-graph algorithm, and each instance involves a graph with at most $\delta\frac{k}{g}$ vertices and δk edges, so the total running time is $O(\lambda(\frac{g}{\delta})^{\log\log k}k^2\log^2 k) = O(\lambda\mathsf{polylog}(k)k^2)$. The same algorithm can also apply to the lossless expander graph in [GLSTW]. Our sampling algorithm is very efficient in practice. First, it does not involve any cryptographic operations and is done once. Second, $k = \sqrt{N}$ in our protocol of the polynomial commitment in the next section, so the complexity is actually quasi-linear in the size of the zero-knowledge argument instance. Finally, the complexity of the densest sub-graph algorithm in Theorem 3.3.4 is for arbitrary graphs. As observed in our experiments, the algorithm is faster on random bipartite graphs and we conjecture that there is a better complexity analysis, which is left as an interesting future work.

3.4 Our new Zero-Knowledge Argument

In this section, we present the construction of our zero-knowledge argument scheme. Many existing papers show that one can build zero-knowledge arguments from polynomial commitments [WTSTW18; ZXZS20; CHMMVW20; Set20; GWC19a; BFS20; GLSTW]. We adopt the same technique and focus on constructing a polynomial commitment because of its simplicity and efficiency, but our approach can be applied directly to the zero-knowledge arguments for R1CS in [BCG20; BCL22] to improve the prover time and the proof size. We start the section by describing the polynomial commitment scheme in [GLSTW] based on the tensor IOP protocol in [BCG20] with a proof size of $O(\sqrt{N})$.

3.4.1 Polynomial commitment from tensor query

In [GLSTW], Golovnev et al. observed that a polynomial evaluation can be expressed as a tensor product. Here we only consider multilinear polynomial commitments, which can be used to construct zero-knowledge arguments based on the approaches in [ZGKPP17b; WTSTW18; XZZPS19c; ZXZS20; Set20], but our scheme can be extended to univariate polynomials. In particular, given a multilinear polynomial ϕ, its evaluation on input vector $x_0, x_1, ..., x_{\log N-1}$ is:

$$\phi(x_0, x_1, ..., x_{\log N-1}) = \sum_{i_0=0}^{1}\sum_{i_1=0}^{1}...\sum_{i_{\log N-1}=0}^{1} w_{i_0 i_1 ... i_{\log N-1}} x_0^{i_0} x_1^{i_1} ... x_{\log N-1}^{i_{\log N-1}}.$$

The degree of each variable is either 0 or 1 by the definition of a multilinear polynomial, and thus there are N monomials and coefficients with $\log N$ variables. We let $i = \sum_{j=0}^{\log N-1} 2^j i_j$, that is, $i_0 i_1 ... i_{\log N-1}$ is the binary representation of number i. We use w to denote the coefficients where $w[i] = w_{i_0 i_1 ... i_{\log N-1}}$. Similarly we define $X_i = x_0^{i_0} x_1^{i_1} ... x_{\log N-1}^{i_{\log N-1}}$. Let $k = \sqrt{N}$, $r_0 = \{X_0, X_1, ..., X_{k-1}\}$,
$r_1 = \{X_{0\times k}, X_{1\times k}, X_{2\times k}, ..., X_{(k-1)\times k}\}$. Then we have $X = r_0 \otimes r_1$. The polynomial evaluation is reduced to a tensor product $\phi(x_0, x_1, ..., x_{\log N-1}) = \langle w, r_0 \otimes r_1\rangle$. Using the tensor IOP protocol in [BCG20], one can build a polynomial commitment [GLSTW] and we present the protocol in Protocol 9 for completeness. Here we reuse the notation k as it is exactly the message length of the linear code.

Protocol 9 Polynomial commitment from [BCG20; GLSTW]

Public input: The evaluation point $\vec{x}$, parsed as a tensor product $r = r_0 \otimes r_1$;
Private input: the polynomial ϕ, the coefficient of ϕ is denoted by w.
Let C be the $[n, k, d]$-linear code, $E_C : \mathbb{F}^k \to \mathbb{F}^n$ be the encoding function, $N = k \times k$. If N is not a perfect square, we can pad it to the next perfect square.
We use a python style notation to select the i-th column of a matrix $\mathsf{mat}[:, i]$.

```
1:  function PC.Commit($\phi$)
2:      Parse $w$ as a $k \times k$ matrix. The prover computes the tensor code encoding $\mathsf{C}_1, \mathsf{C}_2$ locally as defined in Definition 3.2.4. Here $\mathsf{C}_1$ is a $k \times n$ matrix and $\mathsf{C}_2$ is a $n \times n$ matrix.
3:      for $i \in [n]$ do
4:          Compute the Merkle tree root $\mathsf{Root_i} = \mathsf{Merkle.Commit}(\mathsf{C}_2[:, \mathsf{i}])$.
5:      Compute a Merkle tree root $\mathcal{R} = \mathsf{Merkle.Commit}([\mathsf{Root_0}, ..., \mathsf{Root_{n-1}}])$ and output $\mathcal{R}$ as the commitment.
6:  function PC.Prove($\phi, \vec{x}, \mathcal{R}$)
7:      The prover receives a random vector $\gamma_0 \in \mathbb{F}^k$ from the verifier.
8:      $c_{\gamma_0} = \sum_{i=0}^{k-1} \gamma_0[i]\mathsf{C}_1[i], y_{\gamma_0} = \sum_{i=0}^{k-1} \gamma_0[i]w[i]$.          ▷ Proximity
9:      $c_1 = \sum_{i=0}^{k-1} r_0[i]\mathsf{C}_1[i]$, $y_1 = \sum_{i=0}^{k-1} r_0[i]w[i]$.          ▷ Consistency
10:     Prover sends $c_1, y_1, c_{\gamma_0}, y_{\gamma_0}$ to the verifier.
11:     Verifier randomly samples $t \in [n]$ indexes as an array $\hat{I}$ and send it to prover.
12:     for $\mathsf{idx} \in \hat{I}$ do
13:         Prover sends $\mathsf{C}_1[:, \mathsf{idx}]$ and the Merkle tree proof of $\mathsf{Root_{idx}}$ for $\mathsf{C}_2[:, \mathsf{idx}]$ under $\mathcal{R}$ to verifier
14: function PC.VerifyEval($\pi_{\vec{x}}, \vec{x}, y = \phi(\vec{x}), \mathcal{R}$)
15:     $\forall \mathsf{idx} \in \hat{I}, c_{\gamma_0}[\mathsf{idx}] == \langle \gamma_0, \mathsf{C}_1[:, \mathsf{idx}] \rangle$ and $E_C(y_{\gamma_0}) == c_{\gamma_0}$.          ▷ Proximity
16:     $\forall \mathsf{idx} \in \hat{I}, c_1[\mathsf{idx}] == \langle r_0, \mathsf{C}_1[:, \mathsf{idx}] \rangle$ and $E_C(y_1) == c_1$.          ▷ Consistency
17:     $y == \langle r_1, y_1 \rangle$.          ▷ Tensor product
18:     $\forall \mathsf{idx} \in \hat{I}$, $E_C(\mathsf{C}_1[:, \mathsf{idx}])$ is consistent with $\mathsf{Root_{idx}}$, and $\mathsf{Root_{idx}}$'s Merkle tree proof is valid.

19:     Output accept if all conditions above holds. Otherwise output reject.
```

As shown in the protocol, to commit to a polynomial, $\mathsf{PC.Commit}$ parses the coefficients w as a $k \times k$ matrix and encodes it using the tensor code with dimension 2 as defined in Definition 3.2.4. Then the algorithm constructs a Merkle tree commitment for every column $\mathsf{C}_2[:, i]$ of the $n \times n$ codeword C_2, and finally builds another Merkle tree on top of their roots as the final commitment.

To answer the tensor query, there are two checks in the protocol: a proximity check and a consistency check. The proximity check ensures that the matrix in the commitment is indeed close to a codeword of the tensor code. The consistency check ensures that $y = \langle r_0 \otimes r_1, w \rangle$ assuming $\mathcal{R}$ is a commitment of a codeword.

Proximity check. The proximity check has two steps. First, the verifier sends a random vector γ_0 to the prover, and the prover computes the linear combination of all rows of C_1 and w with γ_0, as in Step 8 in Protocol 9.

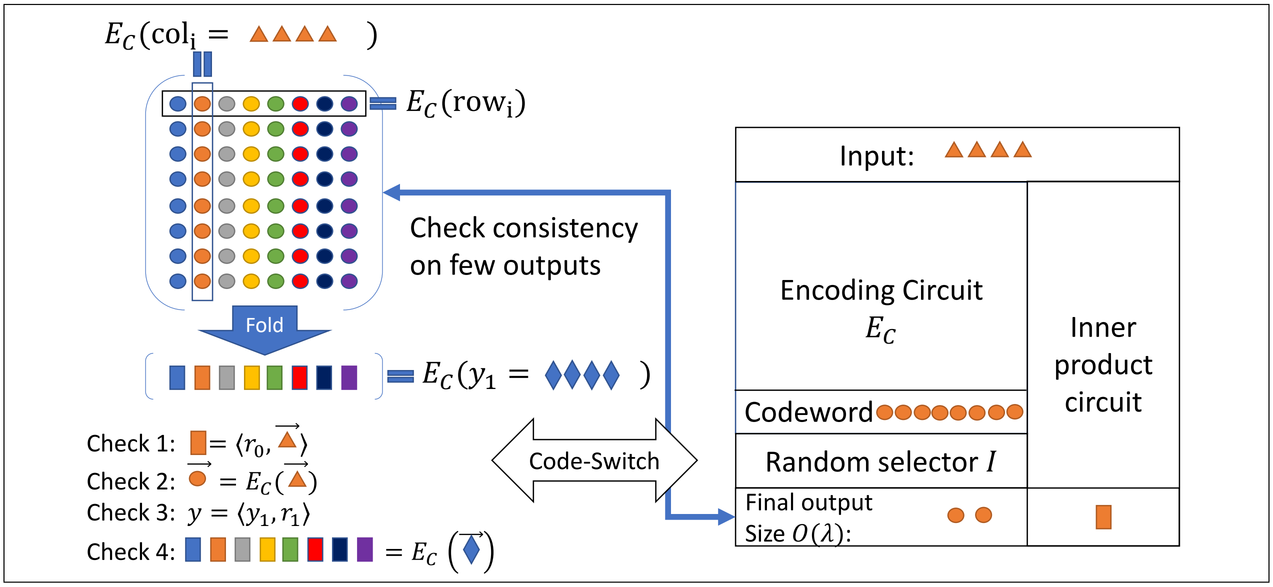

E_C(col$_i$ = ▲▲▲▲)
E_C(row$_i$)
Fold
E_C(y_1 = ◆◆◆◆)
Check consistency on few outputs
Check 1: ■= ⟨r_0, $\vec{▲}$⟩
Check 2: $\vec{●}$ = E_C($\vec{▲}$)
Check 3: $y = \langle y_1, r_1 \rangle$
Check 4: ■■■■■■■■ = E_C($\vec{◆}$)
Code-Switch
Input: ▲▲▲▲
Encoding Circuit E_C
Inner product circuit
Codeword ●●●●●●●●
Random selector I
Final output Size $O(\lambda)$: ●●
■

Figure 3.2: An illustration of code switching. The circuit on the right for Check 1,2 and Check 3,4 is the same.

Because of the property of a linear code, c_{γ_0} is a codeword with message y_{γ_0}, and this step is referred to as the "fold" operation in [BCG20]. Second, the prover shows that c_{γ_0} is indeed computed from the committed tensor codeword. To do so, the verifier randomly selects t columns and the prover opens them with their Merkle tree proofs. The verifier checks that the inner product between each column and the random vector γ_0 is equal to the corresponding element of c_{γ_0} (Step 15). As shown in [BCGGHJ17; BCG20], if the linear code has a constant relative distance, the committed matrix is close to a tensor codeword with overwhelming probability.

Consistency check. The consistency check follows exactly the same steps of the proximity check. Instead of using a random vector from the verifier, the linear combination is done with r_0 of the tensor query $r_0 \otimes r_1$. Similarly, c_1 is a codeword of the linear code with message y_1, and $\phi(x) = \langle y_1, r_1 \rangle$ by the definition of tensor product and polynomial evaluation. As shown in [BCG20], by the check in Step 16, if the committed matrix in $\mathcal{R}$ is close to a tensor codeword, then $y = \phi(x)$ with overwhelming probability. In particular, there exist an extractor to extract a polynomial ϕ from the commitment such that $y = \phi(x)$.

Theorem 3.4.1 (Polynomial commitment [BCG20; GLSTW])**.** *Protocol 9 is a polynomial commitment that is correct and sound as defined in Definition 3.2.8.*

Efficiency. The prover's computation is dominated by encoding the tensor code, which takes $O(N)$ time using a linear-time encodable code such as the generalized Spielman code. The proof size is $O(t\sqrt{N})$, as the prover opens t random columns of size $\sqrt{N}$ to the verifier. The verifier time is also $O(t\sqrt{N})$ to check the inner products and to encode t columns.

3.4.2 Efficient Proof Composition via Code Switching

The proof size of the polynomial commitment in Protocol 9 is $O(\sqrt{N})$ (the complexity hides a security parameter t). There are three steps that incur $O(\sqrt{N})$ proof size in Protocol 9: Step 8, 9, and 13. In this section, we present a new protocol that reduces the proof size to $O(\log^2 N)$ via the technique of proof composition. The idea is to use a second proof system to prove that the checks of these three steps are satisfied, without sending the proofs of these steps to the verifier directly.

To design the second proof system efficiently, our key observation is that the values sent by the prover in these three steps are messages of the linear-time encodable code. That is, y_{γ_0} is the message of c_{γ_0} in Step 8, y_1 is the message of c_1 in Step 9 and $C_1[:, \mathsf{idx}]$ is the message of $C_2[:, \mathsf{idx}]$ for every idx in Step 13. Therefore, the second proof system takes y_{γ_0}, y_1 and $\mathsf{C}_1[:, \mathsf{idx}]$ for $\mathsf{idx} \in I$ as the witness, and performs the following computations:

1. It encodes the witness using the encoding circuit of the linear-time encodable code.

2. It outputs a subset of random indices of the codewords chosen by the verifier. By checking whether the values of these indices are consistent with the commitments by the prover via the Merkle tree, it guarantees that the witness is indeed the same as the messages specified above with overwhelming probability because of the minimum distance property of the code.

3. Finally, it checks that these messages and their codewords satisfy the conditions in line 15, 16 and 17 of Protocol 9.

The idea is illustrated in Figure 3.2, and we formally present the statement of the second proof system in Protocol 10. Note that $\hat{I}$ is the random set chosen by the verifier in Protocol 9, and is only used as a notation for the subscripts in Protocol 10. I is the random set chosen by the verifier for the code switching. In this way, we switch the message encoded using the linear-time encodable code to the witness of the second proof system. In our implementation, we are using an IOP-based zero-knowledge argument with the Reed-Solomon code, thus this can be viewed as an efficient instantiation of the "code switching" technique in [RR20].

We apply any zero-knowledge argument scheme $\mathcal{ZK}$ on the statement and then check the consistency between the output and the Merkle tree commitment $\mathcal{R}$ of the codeword of the linear-time encodable code. We present the new protocol in Protocol 11 and highlight the differences from Protocol 9 in blue. As shown in the protocol, instead of sending $c_1, y_1, c_{\gamma_0}, y_{\gamma_0}$, the prover commits to c_1 and c_{γ_0} in Step 8 and 9. The codeword C_2 was already committed column-wise in $\mathcal{R}$. The prover then proves the constraints of $c_1, y_1, c_{\gamma_0}, y_{\gamma_0}$ and $\mathsf{C}_1[:, \mathsf{idx}]$ using the code switching technique in Step 13. In this way, we are able to reduce the proof size of Protocol 9 to $O(\log^2 N)$.

Theorem 3.4.2. *Protocol 11 is a polynomial commitment that is correct and sound, as defined in Definition 3.2.8 without zero-knowledge property.*

The proof is presented in Appendix 3.8.

Complexity of Protocol 11. The prover time remains $O(N)$. This is because in Step 8 and 9, the prover additionally commits to c_1, c_{γ_0}, which only takes $O(n) = O(\sqrt{N})$ time. In Step 13, the prover invokes another zero-knowledge argument on C_{CS}. C_{CS} consists of $t + 2$ encoding circuits E_C of the linear-time encodable code and t+2 inner products. In Appendxi 3.9, we show that the encoding circuit of the generalized Spielman code is of size $O(k)$. The circuit to compute an inner product is of size $O(k)$, thus the overall circuit size is $O(t \cdot k)$. By using any zero-knowledge argument scheme with a quasi-linear prover time, the prover

Protocol 10 Code Switching Statement C_{CS}

```
    **Witness**: $y_{\gamma_0}, y_1$, $\mathsf{C}_1[:, \mathsf{idx}]\ \forall \mathsf{idx} \in \hat{I}$ in Protocol 9.
    **Public input**: $\gamma_0, r_0, r_1, y$.
    **Public information**: $\hat{I}$ and $I$ chosen by the verifier.
 1: Encode $c_{\gamma_0} := E_C(y_{\gamma_0})$, $c_1 := E_C(y_1)$.
 2: for $\mathsf{idx} \in \hat{I}$ do
 3:     Encode $\mathsf{C}_2[:, \mathsf{idx}] := E_C(\mathsf{C}_1[:, \mathsf{idx}])$
 4: for $\mathsf{idx} \in \hat{I}$ do
 5:     Check if $c_{\gamma_0}[\mathsf{idx}] == \langle \gamma_0, \mathsf{C}_1[:, \mathsf{idx}] \rangle$.        ▷ Proximity
 6:     Check if $c_1[\mathsf{idx}] == \langle r_0, \mathsf{C}_1[:, \mathsf{idx}] \rangle$.                ▷ Consistency
 7: Check if $\langle r_1, y_1 \rangle == y$.                                           ▷ Tensor product
 8: for $0 \leq j < |I|$ do                                                           ▷ Encoder check
 9:     Output $c_1[I[j]]$, $c_{\gamma_0}[I[j]]$.
10:     for $\mathsf{idx} \in \hat{I}$ do
11:         Output $\mathsf{C}_2[I[j], \mathsf{idx}]$
```

time of this step is $O(t \cdot k \log k)$. Since $k = \sqrt{N}$, the prover time is still $O(N)$ dominated by the encoding and the commitment of the $k \times k$ matrix. With the code switching technique, the proof size and becomes $O(t \log^2 N)$.

Since we apply $\mathcal{ZK}$ in a black-box way, the verification time of the protocol will be $O(\sqrt{N})$ due to the size of recursive circuit. However, in the holographic setting with preprocessing, the verifier time can be reduced to $O(\log^2 N)$ using the techniques in [Set20]. Note that the code switching circuit depends on I randomly selected by the verifier, which is not known during preprocessing. ($\hat{I}$ is also not known, but it does not affect the structure of C_{CS}.) Fortunately, we are able to preprocess most of the circuits (Step 1-7 in Protocol 10) and the verifier time is $O(\log^2 N) + O(|I| \cdot |\hat{I}|) = O(\log^2 N)$ in this setting.

3.4.3 Putting Everything Together

In this section, we show how to achieve zero-knowledge on top of our new polynomial commitment in Protocol 11, and sketch how to build a zero-knowledge argument using the polynomial commitment.

Achieving zero-knowledge. We apply a masking technique similar to the one in [BCGGHJ17]. The codeword C_2 is masked by a codeword MSK of a masking polynomial with random coefficients m. We use our proof system to prove $y_{w+m} = \langle (w+m), r_0 \otimes r_1 \rangle$ and $y_m = \langle m, r_0 \otimes r_1 \rangle$ simultaneously, and the final answer of the polynomial evaluation is $y = y_{w+m} - y_m$. We present the protocol in Protocol 12.

Theorem 3.4.3. *Protocol 12 is a zero-knowledge polynomial commitment scheme by definition 3.2.8.*

We present the proof in Appendix 3.10.

Zero-knowledge argument. Finally, we build our zero-knowledge argument system by combining the multivariate polynomial commitment with the sumcheck protocol as in [Set20; GLSTW]. We state the theorem here and refer the readers to [Set20; GLSTW] for the construction and the proof.

Protocol 11 Polynomial commitment with code-switching

```
Public input: The evaluation point $\vec{x}$, parsed as a tensor product $r = r_0 \otimes r_1$;
Private input: the polynomial $\phi$ with coefficients $w$.
function COMMIT($\phi$)
    Parse $w$ as a $k \times k$ matrix. The prover computes the tensor code encoding $\mathsf{C}_1, \mathsf{C}_2$ locally as defined in Definition 3.2.4.
    for $i \in [n]$ do
        Compute the Merkle tree root $\mathsf{Root_i} = \mathsf{Merkle.Commit}(\mathsf{C}_2[:, \mathsf{i}])$.
    Compute a Merkle tree root $\mathcal{R} = \mathsf{Merkle.Commit}([\mathsf{Root}_0, ..., \mathsf{Root}_{\mathsf{n}-1}])$ and output $\mathcal{R}$ as the commitment.
function PROVE($\phi, \vec{x}, \mathcal{R}$)
    The prover receives a random vector $\gamma_0 \in \mathbb{F}^k$ from the verifier.
    $c_1 = \sum_{i=0}^{k-1} r_0[i]\mathsf{C}_1[i]$, $y_1 = \sum_{i=0}^{k-1} r_0[i]w[i]$, $\mathcal{R}_{c_1} = \mathsf{Merkle.Commit}(c_1)$
    $c_{\gamma_0} = \sum_{i=0}^{k-1} \gamma_0[i]\mathsf{C}_1[i], y_{\gamma_0} = \sum_{i=0}^{k-1} \gamma_0[i]w[i]$, $\mathcal{R}_{\gamma_0} = \mathsf{Merkle.Commit}(c_{\gamma_0})$
    The prover computes the answer $y := \langle y_1, r_1 \rangle$. Prover sends $\mathcal{R}_{c_1}, \mathcal{R}_{\gamma_0}, y$ to the verifier.
    The verifier randomly samples $t \in [n]$ indexes as an array $\hat{I}$ and send it to prover.
    The verifier randomly samples another index set $I \subseteq [k], |I| = t$ and sends it to the prover.
    The prover calls the zero-knowledge argument protocol $\mathcal{ZK}.\mathcal{P}$ on $C_{\mathsf{CS}}$. Let $\pi_{zk}$ be the proof of the zero-knowledge argument. The prover sends the output of $C_{\mathsf{CS}}$: $\mathsf{C}_2[I[j], \mathsf{idx}]$ $\forall \mathsf{idx} \in \hat{I}$, $c_1[I[j]], c_{\gamma_0}[I[j]]$ and $\pi_{zk}$ to the verifier.
    The prover sends the Merkle tree proofs of $\mathsf{C}_2[I[j], \mathsf{idx}]$ $\forall \mathsf{idx} \in \hat{I}$ under $\mathsf{Root}_{\mathsf{idx}}$.
    The prover sends the Merkle tree proofs of $\mathsf{Root}_{\mathsf{idx}}$ $\forall \mathsf{idx} \in \hat{I}$ under $\mathcal{R}$.
    The prover sends the Merkle tree proofs of $c_1[I[j]], c_{\gamma_0}[I[j]]$ under $\mathcal{R}_{c_1}, \mathcal{R}_{c_{\gamma_0}}$.
function VERIFYEVAL($\pi_{\vec{x}}, \vec{x}, y = \phi(\vec{x}), \mathcal{R}$)
    The verifier calls the zero-knowledge argument protocol $\mathcal{ZK}.\mathcal{V}$ on $C_{\mathsf{CS}}$.
    The verifier checks the Merkle tree proofs of $\mathsf{C}_2[I[j], \mathsf{idx}]$ $\forall \mathsf{idx} \in \hat{I}$.
    The verifier checks the Merkle tree proofs of $\mathsf{Root}_{\mathsf{idx}}$ $\forall \mathsf{idx} \in \hat{I}$ using $\mathcal{R}$.
    The verifier checks the Merkle tree proofs of $c_1[I[j]], c_{\gamma_0}[I[j]]$ using $\mathcal{R}_{c_1}, \mathcal{R}_{c_{\gamma_0}}$.
Output accept if all checks pass. Otherwise output reject.
```

Theorem 3.4.4. *There exists a zero-knowledge argument scheme by definition 3.2.6 with $O(N)$ prover time, $O(\log^2 N)$ proof size and $O(N)$ verifier time.*

As we are using the IOP-based scheme in [ZXZS20] as the second zero-knowledge argument in the proof composition, our scheme is an IOP with a linear proof size and logarithmic query complexity. The scheme can be made non-interactive via the Fiat-Shamir [FS86] heuristic, and has plausible post-quantum security. Following the frameworks in [CHMMVW20; COS20; Set20; GLSTW], our scheme can be turned into a holographic proof with a $O(\log^2 N)$ verifier time in a straight-forward way.

Protocol 12 zk-Polynomial commitment

```
    **Public input**: The evaluation point $\vec{x}$, parsed as a tensor product $r = r_0 \otimes r_1$;
    **Private input**: the polynomial $\phi$ with coefficients $w$.
function zkCommit($\phi_w$)
    The prover randomly samples $m \in \mathbb{F}^{|w|}$.
    Output $\mathcal{R}_{w+m} = \mathsf{COMMIT}(w + m), \mathcal{R}_m = \mathsf{COMMIT}(m)$.
function zkProve($\phi, \vec{x}, \mathcal{R}$)
    Let $\phi_m$ be the masking polynomial, $\phi_{w+m}$ be the masked polynomial.
    Run $\mathsf{Prove}(\phi_{w+m}, \vec{x}, \mathcal{R}_{w+m})$. Let the random index set used during the protocol be $\hat{I}_0, I_0$.
    Run $\mathsf{Prove}(\phi_m, \vec{x}, \mathcal{R}_m)$. In this step, the verifier samples the random index set $\hat{I}_1, I_1$. used during the protocol such that $\hat{I}_0 \cap \hat{I}_1 = \emptyset \wedge I_0 \cap I_1 = \emptyset$.
function zkVerify($\pi_{\vec{x}}^{w+m}, \pi_{\vec{x}}^{m}, \vec{x}, y_{w+m}, y_m, \mathcal{R}_{w+m}, \mathcal{R}_m$)
    The final polynomial evaluation $\phi(\vec{x})$ should be $y_{w+m} - y_m$.
    Execute $\mathsf{VerifyEval}(\pi_{w+m}, \vec{x}, y_{w+m}, \mathcal{R}_{w+m})$.
    Execute $\mathsf{VerifyEval}(\pi_m, \vec{x}, y_m, \mathcal{R}_m)$.
    Output accept if all checks above passes, otherwise output reject.
```

3.5 Experiments

We have implemented our scheme, Orion, and we present the evaluations of the system and the comparions to existing ZKP schemes in this section.

Settings and parameters. Our polynomial commitment scheme is implemented in C++ with 6000 lines of code. The proof composition uses Virgo in [ZXZS20] and its open-source implementation. We combine the polynomial commitment with a sumcheck protocol to get our zero-knowledge argument following the approach in [Set20] and we implement our own code for this part.

Expander graph used in our implementation We use a modified version of generalized Spielman code in [GLSTW]. The code assigns a random weight to each edge of the expander graph, achieving a better minimum distance. We take a step further and fine-tune the dimensions more aggressively. With our testing algorithm, the failure probability of the expander sampling remains negligible. There are two types of expander graph used in our construction and the parameters are G_1: $\alpha = 0.33, \delta = 0.6, \epsilon = 0.78, g = 6$; G_2: $\alpha = 0.337, g = 6, \delta = g, \epsilon = 0.88$.

Parameters of our linear code. With expanders above, the final relative distance is 0.055. We set the security parameter $\lambda = 128$. This leads to opening $t = \frac{-128}{\log(1-0.055)} = 1568$ columns and locations in Protocol 11.

Hash function and finite field. We use the SHA-256 hash function implemented by [arm]. We use the extension field of GF$((2^{61} - 1)^2)$ as our underlying field to be compatible with the zero-knowledge argument in [ZXZS20].

Environment and method. We use an AWS m6i-32xlarge instance with Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz CPU and 512GB memory to execute all of our experiments. However, the largest instance in our experiment only utilize 16 GB of memory. All experiments are using a single thread except the expander testing algorithm. For each data point, we run the experiments 10 times and report the average.

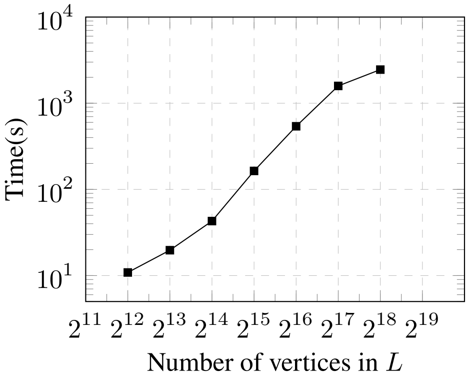


Figure 3.3: Running time of our expander testing algorithm.

3.5.1 Expander Testing

We first show the performance of our expander testing algorithm in Section 3.3. We implemented the densest sub-graph algorithm in [Gol84], which uses network-flow algorithm as a black-box. In our implementation, we use Dinic's algorithm [Din70], the complexity of which is $O(|V|^2|E|)$ on general graphs. However, on random bipartite graphs, the Dinic's algorithm runs significantly faster. As observed in our experiments, it scales almost linearly in the size of the graph.

Figure 3.3 shows the running time of the algorithm. We vary the size of left vertex set L in the random bipartite graph from 2^{12} to 2^{18}, and the size of R is set to be $|L| \times \alpha$. The implementation uses multi-threading utilizing all 128 CPU cores. As shown in the figure, it only takes 163 seconds to test whether a random bipartite graph with $|L| = 2^{15}$ vertices is a lossless expander with a failure probability $\mathsf{negl}(N) = 2^{-128}$. The running time almost grows linearly in $|L|$. As $k = \sqrt{N}$ in our zero-knowledge argument, this is enough for our experiments. As the sampling of the lossless expander is done once, our testing algorithm is very practical.

3.5.2 Polynomial Commitment

In this section, we report the performance of our polynomial commitment scheme and compare it with the scheme Brakedown in [GLSTW], which is the only implemented polynomial commitment scheme with a linear prover time. We use the open-source implementation of Brakedown at [Wa] in the comparison. Our current implementation is for the plain version of the polynomial commitment without zero-knowledge, which is the same as Brakedown.

Figure 3.4 shows the performance of our polynomial commitment and the polynomial commitment in Brakedown. We vary the size of the polynomials from 2^{15} to 2^{29} and measure the prover time, the proof size and the verifier time. As shown in the figure, our prover time is even slightly faster than Brakedown. It only takes 115 seconds for a polynomial with 2^{27} coefficients, while it is 132 seconds in Brakedown. This is because we use more aggressive parameters of the expander code, while still achieving 128-bit of security thanks to our expander testing algorithm. Moreover, the additional proof composition in our scheme involves a second zero-knowledge argument on a circuit of size $O(\sqrt{N})$. In our experiments, this extra zero-knowledge argument takes less than 20% of the total prover time, justifying that our code switching technique only introduces a small overhead on the prover time.

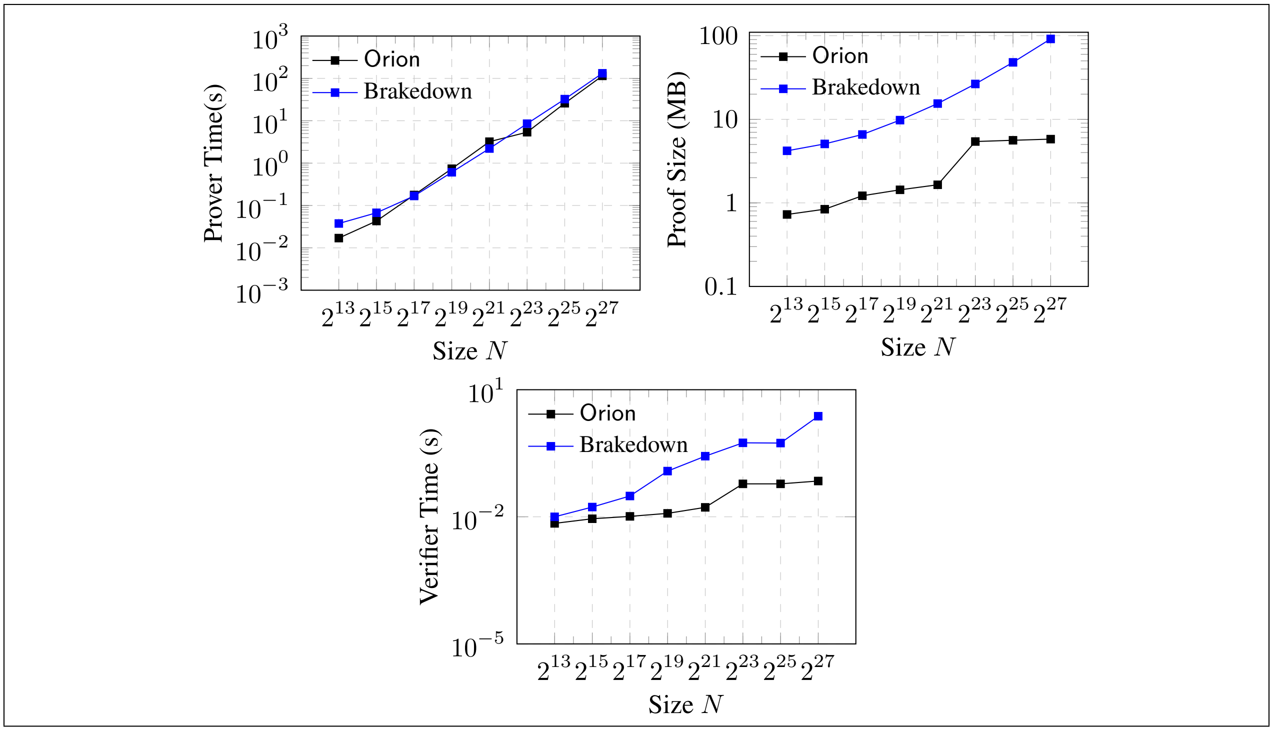


Figure 3.4: Performance of polynomial commitments.

Our proof size and verifier time is significantly smaller than Brakedown. The proof size is only 6 MBs for a polynomial of size 2^{27}, 16× smaller than Brakedown. The verifier time is 70ms for $N = 2^{27}$, 33× faster than Brakedown. The result demonstrates the improvement of the $O(\log^2 N)$ proof size and verifier time in our scheme.

Note that there is a jump from $N = 2^{21}$ to $N = 2^{23}$ in the proof size and verifier time. This is because in our implementation, instead of directly parsing the coefficients into $\sqrt{N} \times \sqrt{N}$ matrix, we optimize the dimensions for better performance. When $N < 2^{23}$, it is not meaningful to do code-switching on the columns. The prover only does the code-switching on the row (Protocol 11 Step 8 and 9), but opens the columns directly. We observe that this gives the best prover time and the proof size. When $N \geq 2^{23}$, the prover does the code-switching for both the row and the columns (Protocol 11, Step 8–13). Therefore, the proof size and the verifier time have a big increase because of the larger column size and the additional code-switching protocol.

3.5.3 Zero-knowledge Arguments

Finally, we present the performance of our zero-knowledege argument scheme for R1CS as a whole in this section. We focus the comparison to existing schemes that work on R1CS and have transparent setup and plausible post-quantum security. They include Brakedown [GLSTW], Aurora [BSCRSVW19] and Ligero [AHIV17]. We use the implementation of Brakedown at [Wa], and the open-source code of Ligero and Aurora at [Aur] in the experiments.

We randomly generate the R1CS instances and vary the number of constraints from 2^{15} to 2^{20}. As shown in Figure 3.5, Orion has the fastest prover among all schemes. It only takes 3.09 seconds to generate the proof

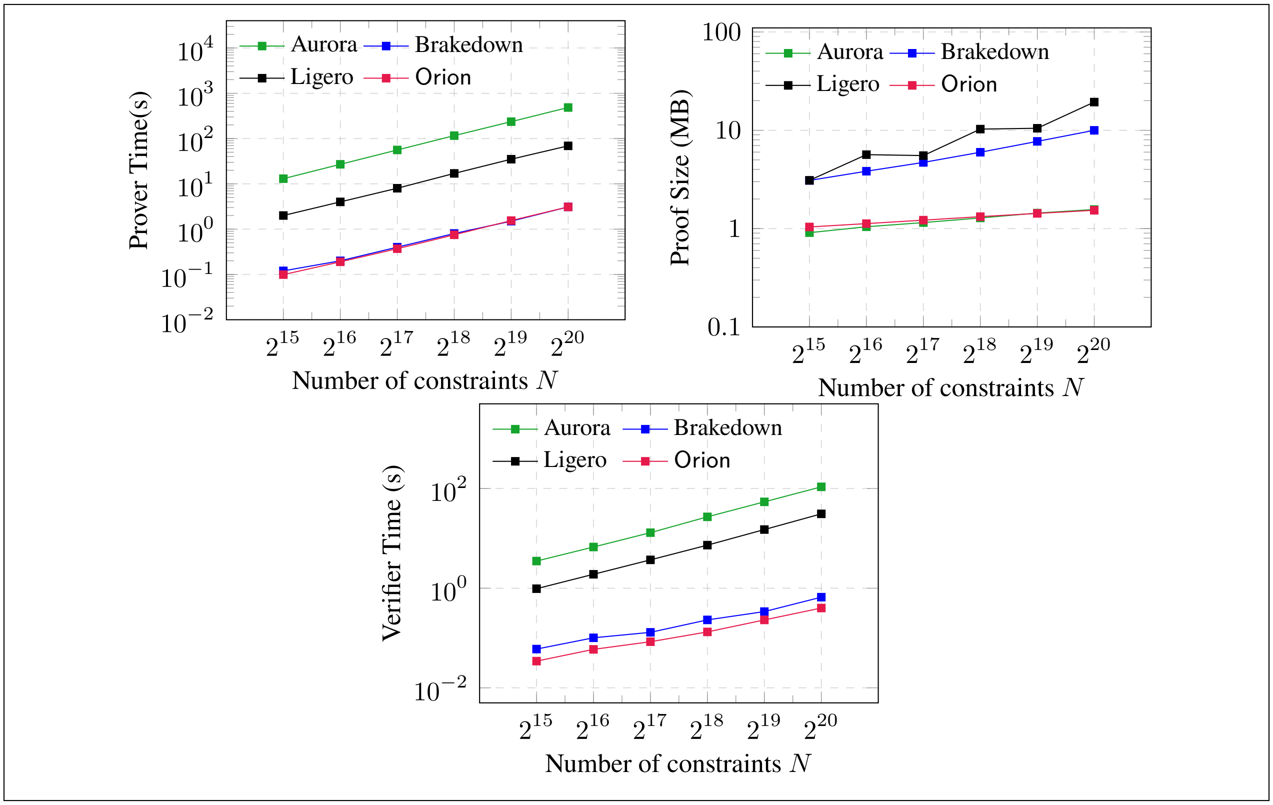


Figure 3.5: Performance of zero-knowledge arguments on R1CS.

for $N = 2^{20}$. This is slightly faster than Brakedown for the same reason as explained in Section 3.5.2. It is 20× faster than Ligero and 142× faster than Aurora because of the linear prover time and the simplified reduction via polynomial commitments.

The proof size of Orion is significantly smaller than Brakedown and Ligero. It is only 1.5 MB for $N = 2^{20}$, 6.5× smaller than Brakedown and 12.5× smaller than Ligero. The proof size is even comparable to Aurora, which has $O(\log^2 N)$ proof size and uses the Reed-Solomon code with a much better minimum distance than our linear code. The result justifies the improvement of our code switching.

We only implemented and compared with the variants of the protocols with a linear verifier time. As explained in the introduction, the verifier time of all schemes grow linearly with N in the worst case, and the comparisons are similar to the prover time. One can reduce the verifier time to sublinear in the holographic setting using the techniques in [CHMMVW20; COS20; Set20].

Other related schemes. There are several other existing transparent zero-knowledge argument schemes. Hyrax [WTSTW18], Virgo [ZXZS20] and Virgo++ [Zha+21a] work on layered arithmetic circuits and STARK [BSBHR19] works on an algebraic intermediate representation that is close to a RAM program. It is hard to compare directly to R1CS, but we expect our prover time to be faster than these systems for similar computations based on the results shown in prior papers [ZXZS20; Zha+21a]. Spartan and schemes in [SL20] are using the same framework of polynomial commitment and sumcheck as in our scheme. However, they are based on discrete-log and bilinear pairing and thus are not post-quantum secure. As shown in [GLSTW], their prover time is slower than Brakedown while the proof size is better (tens of KBs). Finally, Bulletproofs [BBBPWM18] and Supersonic [BFS20] are based on discrete-log and group of unknown order. Their prover time is orders of magnitude slower than schemes mentioned above, while providing the smallest proof size (1-2 KBs) because of the underlying cryptographic techniques.

3.6 Appendix

3.7 Proof of Lemma 3.3.2

Proof. When $s \geq \log\log k$, we have following:

1. $e^{(1-\epsilon)gs+s} = e^{O(s)} = e^{c_0 s}$ for some constant c_0.
2. $(\frac{(1-\epsilon)gs}{k'})^{\epsilon gs} \leq (\frac{gs}{k'})^{\epsilon gs}$

We take the expression in the summation and simplify it:

$$e^{(1-\epsilon)gs+s} \cdot (\frac{k}{s})^s \cdot (\frac{(1-\epsilon)gs}{k'})^{\epsilon gs} \leq e^{c_0 s}(\frac{k}{s})^s(\frac{gs}{k'})^{\epsilon gs}$$

Let $f(x) = e^{c_0 x}(\frac{k}{x})^x(\frac{gx}{k'})^{\epsilon gx}$, then its derivative $f'(x) = e^{c_0 x}(\frac{k}{x})^x(\frac{gx}{k'})^{\epsilon gx} \cdot (c_0 + \epsilon g \log\frac{gx}{k'} + \epsilon g + \log\frac{k}{x} - 1)$. Let $g(x) = (c_0 + \epsilon g \log\frac{gx}{k'} + \epsilon g + \log\frac{k}{x} - 1)$, we know that when $x > 2$, $f'(x)$ is positive (negative or zero) if and only if $g(x)$ is positive (negative or zero). Taking the derivative of $g(x)$, $g'(x) = \frac{\epsilon g - 1}{x} > 0$ so $f(x)$ is a convex function. Therefore, the maximum of $f(x)$ is $\max_{x \in [\log\log k, \frac{\delta k}{g}]}(f(x)) = \max(f(\log\log k), f(\frac{\delta k}{g}))$.

We then compute these two values at the boundaries:

1. $f(\log\log k) = \log^{c_0}(k)(\frac{k}{\log\log k})^{\log\log k}(\frac{g\log\log k}{k'})^{\epsilon g\log\log k}$, since $k' = \alpha k, \epsilon g > 2$, the equation is

   $$\leq \log^{c_0}(k)(\frac{k}{\log\log k})^{\log\log k}(\frac{\frac{g}{\alpha}\log\log k}{k})^{2\log\log k} = \tilde{O}((\frac{\log\log k}{k})^{\log\log k}),$$

   which is negligible.

2. $f(\frac{\delta k}{g}) = e^{c_0\frac{\delta k}{g}}(\frac{g}{\delta})^{\frac{\delta}{g}k}(\frac{\delta k}{k'})^{\epsilon\delta k} = e^{(\frac{c_0\delta}{g} + \frac{\delta}{g}\log(\frac{g}{\delta}))k + \log(\frac{\delta}{\alpha})\epsilon\delta k}$.
   It is negligible if $\frac{c_0\delta}{g} + \frac{\delta}{g}\log(\frac{g}{\delta}) + \log(\frac{\delta}{\alpha})\epsilon\delta < -0.01$. Therefore, we set $c_0 = (1-\epsilon)g + 1$, and we have $\frac{c_0\delta}{g} + \frac{\delta}{g}\log(\frac{g}{\delta}) + \log(\frac{\delta}{\alpha})\epsilon\delta = (1-\epsilon)\delta + \frac{\delta}{g} + \frac{\delta}{g}\log(\frac{g}{\delta}) + \log(\frac{\delta}{\alpha})\epsilon\delta < -0.001$

The reasoning above shows that every single value in the summation is negligible as the maximum is negligible, and there are linear number of values in the summation, so the summation is negligible.

□ □

3.8 Proof of Theorem 3.4.2

Proof. **Correctness.** It follows the correctness of Protocol 9, the zero-knowledge argument $\mathcal{ZK}$ on C_{CS}, and the Merkle trees.

Soundness. By Step 18 of Protocol 11, $\mathcal{E}$ first extracts the witness $w^* \in \mathbb{F}^{(t+2)k}$ of $\mathcal{ZK}$ on C_{CS} using $\mathcal{E}_{\mathcal{ZK}}$. Parse w^* as $y^*_{\gamma_0}, y^*_1$ and $\mathsf{C}^*_1[:, \mathsf{idx}]$ for $\mathsf{idx} \in \hat{I}$, each of length k. Let c_{γ_0}, c_1 and $\mathsf{C}_2[:, \mathsf{idx}]$ for $\mathsf{idx} \in \hat{I}$ be vectors committed by $\mathcal{P}$ under $\mathcal{R}_{\gamma_0}, \mathcal{R}_{c_1}, \mathsf{Root}_{\mathsf{idx}}$ in Step 8, 9 and 4 in Protocol 11. By the check in Step 21, we have

$$\Pr\left(\Delta(c_1, E_C(y^*_1)) > \frac{d}{2}\right) \leq \mathsf{negl}(N).$$

To see this, since the minimum distance of the code is $d = O(k)$, if the vector in $\mathcal{R}_{c_1}$ is at least $\frac{d}{2}$-far from the codeword of y^*_1, then the probability that c_1 and $E_C(y^*_1)$ agrees on any idx is $\frac{d}{2k}$. Therefore, the probability to pass all $t = O(\lambda)$ checks in I in Step 21 is at most $(1 - \frac{d}{2k})^t$, which is $\mathsf{negl}(N)$. Similarly,

$$\Pr\left(\Delta(c_{\gamma_0}, E_C(y^*_{\gamma_0})) > \frac{d}{2}\right) \leq \mathsf{negl}(N),$$

and

$$\Pr\left(\Delta(\mathsf{C}_2[:, \mathsf{idx}], E_C(\mathsf{C}^*_1[:, \mathsf{idx}])) > \frac{d}{2}\right) \leq \mathsf{negl}(N), \forall \mathsf{idx} \in \hat{I}.$$

This technique is exactly the proximity check. Therefore, $y^*_{\gamma_0}, y^*_1$ and $\mathsf{C}^*_1[:, \mathsf{idx}]$ for $\mathsf{idx} \in \hat{I}$ are indeed the only messages within the distance of $\frac{d}{2}$ of c_{γ_0}, c_1 and $\mathsf{C}_2[:, \mathsf{idx}]$ for $\mathsf{idx} \in \hat{I}$ respectively, except for $\mathsf{negl}(N)$ probability.

Moreover, by the soundness of $\mathcal{ZK}$ on C_{CS},

$$\Pr\left(y \neq \langle y^*_1, r_0\rangle\right) \leq \mathsf{negl}(N),$$
$$\Pr\left(E_C(y^*_{\gamma_0})[\mathsf{idx}] \neq \langle \mathsf{C}^*_1[:, \mathsf{idx}], \gamma_0\rangle\right) \leq \mathsf{negl}(N), \forall \mathsf{idx} \in \hat{I}.$$

and

$$\Pr\left(E_C(y^*_1)[\mathsf{idx}] \neq \langle \mathsf{C}^*_1[:, \mathsf{idx}], r_0\rangle\right) \leq \mathsf{negl}(N), \forall \mathsf{idx} \in \hat{I}.$$

Therefore, $w = y^*_1, y^*_{\gamma_0}, (\mathsf{C}^*_1[:, \mathsf{idx}] \forall \mathsf{idx} \in \hat{I})$, and $E_C(y^*_1), E_C(y^*_{\gamma_0})$ passes the PC.VerifyEval in Protocol 9. By Theorem 3.4.1, $\mathcal{E}$ calls the extractor $\mathcal{E}_{PC}$ to extract the coefficients of a polynomial ϕ such that $\phi(\vec{x}) = y$, where $x = r_0 \otimes r_1$, except with negligible probability. This completes the proof of knowledge soundness. □

□

3.9 Encoding circuit

Recall the construction of generalized Spielman code in Preliminary section 3.2.1, we prove the following:

Lemma 3.9.1 (Size of the encoder circuit)**.** *The size of the encoder circuit for input size $k = 2^t$, is at most $8dk$. And the circuit depth is $O(\log N)$*

Proof. We prove by induction:

1. If $k \leq n_0$, the lemma holds.
2. Assume for all $k^* \leq 2^{t-1}$ the lemma holds, we prove for $k = 2^t$ the lemma holds:

   a) The step $m_1 = xA_t$ can be done in dk steps, since A_t represents an expander graph with dk edges, so A_t is sparse and have only dk non-zeros.
   b) The step $c_1 = E_C^{t-1}(m_1)$ costs at most $8d\frac{k}{2} = 4dk$ by induction.
   c) The step $c_2 = c_1 B_{t+1}$ costs at most $2dk$ since B_{t+1} represents an expander with $2dk$ edges.
   d) In total the cost is $7dk \leq 8dk$.

The circuit depth is $O(t) = O(\log N)$ from the construction.

□

3.10 Proof of Theorem 3.4.3

Proof. The correctness and the soundness follow Theorem 3.4.2. Here we give the proof for zero-knowledge.

The simulator $\mathcal{S}$ is constructed in Protocol 13. Next we prove that every message sent by the simulator is indistinguishable from the real-world execution as follows:

1. In Step 8, $\mathcal{S}$ directly runs Prove without any modification, so it is indistinguishable from the real-world execution.

2. In Step 14, $\mathcal{S}$ sends two hashes and the result y from the oracle access. by the hiding property of the Merkle tree, they are indistinguishable from the real-world execution.

3. In Step 15, $\mathcal{S}$ calls the simulator of $\mathcal{ZK}$, making $\pi_{\mathcal{ZK}}$ indistinguishable from the real world without knowing the witness of the zero-knowledge argument.

4. In Step 16, 17, 18, at most $O(|\hat{I}_0|)$ entries of the codeword are queried by the verifier, where each entry is a combination of the message $w_{\mathcal{S}}$. Since $|\hat{I}_0| < k$, these queries are uniformly distributed. The same analysis applies to the real-world execution, which also outputs uniformly random values.

Next we need to show that the simulated proof actually passes verification. In Protocol 13 step 13, $\mathcal{S}$ creates a simulated codeword that only agrees with the random codeword $E_C(w+\vec{m})$ on queried points. The simulated codeword c_1^* and the message are computed by solving a set of linear equations. Let G be the generator matrix of E_C, We have following constraints:

1. $\forall i \in I_0, (y_1^* G)[i] == c_1[i]$,

2. $\langle y_1^*, r_1 \rangle == y$

There are k variables in y_1^* but only $|I_0| + 1$ equations, so $\mathcal{S}$ can solve this equation using the Gaussian elimination algorithm and get a valid y_1^*, then computes $c_1^* := y_1^* G$ or $c_1^* := E_C(y_1^*)$.

Finally, the verification of $\mathcal{ZK}$ and the three Merkle tree checks in step 16,17, 18 all pass. The former is because of the the simulator $\mathcal{S}_{\mathcal{ZK}}$. Step 16, 17 naturally passes because C_2 is consistent with $\mathcal{R}_{w_{\mathcal{S}}}$. Step

Protocol 13 Simulators

```
1:  function $\mathcal{S}_0(\mathsf{pp})$
2:      Randomly sample two vectors $w^*, m$.
3:      Output $\mathcal{R}_{w^*+m} := \mathsf{COMMIT}(w^* + m), \mathcal{R}_m := \mathsf{COMMIT}(m)$.
4:  function $\mathcal{S}_1^{\mathcal{A}}(\vec{x}, \mathsf{pp})$
5:      The simulator receives a random vector $\gamma_0 \in \mathbb{F}^k$ from the verifier.
6:      The simulator reads $\mathcal{A}$'s random tape to get $\hat{I}_0, I_0, \hat{I}_1, I_1$.
7:      The simulator will abort if $\hat{I}_0 \cap \hat{I}_1 \neq \emptyset \vee I_0 \cap I_1 \neq \emptyset$.
8:      The simulator runs $\mathsf{Prove}(\phi_m, \vec{x}, \mathcal{R}_m)$ over random tape $I_1, \hat{I}_1$.
    Next, the simulator simulates $\mathsf{Prove}(\phi_{w+m}, \vec{x}, \mathcal{R}_{w+m})$ without knowing the real polynomial $w$.
9:      The simulator makes an oracle query to obtain $y := \phi_w(\vec{x})$.
10:     Let $w_{\mathcal{S}} := w^* + m$, $\mathsf{C}_1, \mathsf{C}_2$ encodes $w_{\mathcal{S}}$.
11:     $c_{\gamma_0} = \sum_{i=0}^{k-1} \gamma_0[i]\mathsf{C}_1[i], y_{\gamma_0} = \sum_{i=0}^{k-1} \gamma_0[i] w_{\mathcal{S}}[i]$, $\mathcal{R}_{\gamma_0} = \mathsf{Merkle.Commit}(c_{\gamma_0})$.
12:     The simulator computes $c_1 = \sum_{i=0}^{k-1} r_0[i]\mathsf{C}_1[i]$
13:     The simulator creates $c_1^*, y_1^*$, such that $\forall i \in I_0, c_1^*[i] == c_1[i]$, $\langle y_1^*, r_1 \rangle == y$, and $E_C(y_1^*) == c_1^*$, $\mathcal{R}_{c_1^*} = \mathsf{Merkle.Commit}(c_1^*)$. $y_1^*$ can be computed by solving a system of linear equations.
14:     The prover sends $\mathcal{R}_{c_1^*}, \mathcal{R}_{\gamma_0}, y$ to the verifier.
15:     The prover calls the zero-knowledge argument protocol simulator $\mathcal{ZK}.\mathcal{S}$ on $C_{\mathsf{CS}}$. Let $\pi_{zk}$ be the proof of the zero-knowledge argument. The prover sends the output of $C_{\mathsf{CS}}$: $\mathsf{C}_2[I_0[j], \mathsf{idx}]$ $\forall \mathsf{idx} \in \hat{I}_0, c_1[I_0[j]], c_{\gamma_0}[I_0[j]]$ and $\pi_{zk}$ to the verifier.
16:     The prover sends the Merkle tree proofs of $\mathsf{C}_2[I_0[j], \mathsf{idx}]$ $\forall \mathsf{idx} \in \hat{I}_0$ under $\mathsf{Root}_{\mathsf{idx}}$.
17:     The prover sends the Merkle tree proofs of $\mathsf{Root}_{\mathsf{idx}}$ $\forall \mathsf{idx} \in \hat{I}_0$ under $\mathcal{R}_{w_{\mathcal{S}}}$.
18:     The prover sends the Merkle tree proofs of $c_1[I_0[j]], c_{\gamma_0}[I_0[j]]$ under $\mathcal{R}_{c_1^*}, \mathcal{R}_{c_{\gamma_0}}$.
```

18 will pass because c_1^* is constructed to agree with c_1 on index set I_0. Therefore, all the checks in the verification are satisfied.

□

□

Chapter 4

Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs

In the past decade, blockchains have seen various financial and technological innovations, with cryptocurrencies reaching a market cap of over 1 trillion dollars. However, scalability is one of the key issues hindering the deployment of blockchains in many applications. To improve the throughput of the transactions, zkRollups and zkEVM techniques using the cryptographic primitive of zero-knowledge proofs (ZKPs) have been proposed and many companies are adopting these technologies in the layer-2 solutions. However, in these technologies, the proof generation of the ZKP is the bottleneck and the companies have to deploy powerful machines with TBs of memory to batch a large number of transactions in a ZKP.

In this work, we improve the scalability of these techniques by proposing new schemes of fully distributed ZKPs. Our schemes can improve the efficiency and the scalability of ZKPs using multiple machines, while the communication among the machines is minimal. With our schemes, the ZKP generation can be distributed to multiple participants in a model similar to the mining pools. Our protocols are based on Plonk, an efficient zero-knowledge proof system with a universal trusted setup. The first protocol is for data-parallel circuits. For computation of M sub-circuits of size T each, using M machines, the prover time is $O(T \log T + M \log M)$, while the prover time of the original Plonk on a single machine is $O(MT \log(MT))$. Our protocol incurs only $O(1)$ communication per machine, and the proof size and verifier time are both $O(1)$, the same as the original Plonk. Moreover, we show that with minor modifications, our second protocol can support general circuits with arbitrary connections while preserving the same proving, verifying, and communication complexity.

We implement Pianist (Plonk vIA uNlimited dISTribution), a fully distributed ZKP system using our protocols. Pianist can generate the proof for 8192 transactions in 313 seconds on 64 machines. This improves the scalability of the Plonk scheme by 64×. The communication per machine is only 2.1 KB, regardless of the number of machines and the size of the circuit. The proof size is 2.2 KB and the verifier time is 3.5 ms. We further show that Pianist has similar improvements for general circuits. On a randomly generated circuit with 2^{25} gates, it only takes 5s to generate the proof using 32 machines, $24.2\times$ faster than Plonk on a single machine.

4.1 Introduction

Blockchain technology has paved the way for innovative services such as decentralized finance, NFTs, and GameFi. The cryptocurrency market has experienced significant growth, surpassing 1 trillion USD in value since Bitcoin's inception 13 years ago [Coi]. Techniques like zkRollups and zkEVM have been proposed to boost blockchain efficiency and bridge the transaction throughput gap between digital and traditional scenarios. Implementing zkRollups could potentially increase transaction throughput by over 100 times, as estimated by Vitalik Buterin [Vit]. Numerous companies have incorporated these techniques into their products, including zkSync [Zks], Starkware [Sta], Hermez [Her], Aztec [Azt], Scroll [Scr], and others.

zkRollups and zkEVM rely on zero-knowledge proofs (ZKPs), a cryptographic primitive that allows a prover to convince a verifier the correctness of computations. More specifically, they use Zero-Knowledge Succinct Non-interactive Argument of Knowledge (ZK-SNARK) systems, which ensures that the proof size is significantly smaller than the size of computation and enables faster validation. By utilizing ZKPs, a single server can validate multiple transactions, compute state transitions, and generate a proof that is posted on the blockchain. Instead of re-executing all transactions, nodes can verify transactions and smart contracts by checking the proof and updating their status. This approach greatly increases the transaction throughput of the blockchain.

However, the proof generation remains a significant bottleneck for existing ZKP schemes when applied to large-scale statements such as zkRollups and zkEVM. For instance, our experiments show that the Plonk system [GWC19b], a widely-used ZKP protocol in the industry, can only scale to a circuit with 2^{25} gates on a machine with 200 GB of memory. As a result, companies like Starkware [Sta] and Scroll [Scr] must deploy powerful clusters with terabytes of memory to generate proofs for zkRollups and zkEVM. In this paper, we tackle this issue by proposing fully distributed ZKP schemes that enhance both efficiency and scalability through distributed proof generation across multiple machines. Crucially, our schemes require minimal communication among machines, with each machine only exchanging a constant number of values with the master machine. This approach allows us to distribute ZKP generation in zkRollups and zkEVM among multiple participants, in a similar model to existing mining pools. More transactions can be batched into a single ZKP within a fixed period, without necessitating participants to stay online and communicate with each other with high overhead. Participants can potentially share the reward for generating the ZKP, akin to miners in current proof-of-work blockchains. Furthermore, our scheme can be generalized to create proofs for arbitrary general circuits, leading us to the name "fully distributed ZKPs".

Our distributed schemes are built upon Plonk [GWC19b]. Instead of using univariate polynomials to represent the constraints of a computation, we devise a protocol based on a bivariate constraint system. First, we claim that this protocol can cater to data-parallel circuits, allowing each machine to generate the witness for its corresponding sub-circuit. Second, we further generalize it to compute proofs for general circuits with aribitrary connections, assuming the witness has already been distributed among the machines. In both cases, our schemes demonstrate that the efficiency and scalability can be improved by a factor of M using M machines, the proof size remains $O(1)$, and the communication complexity per machine is only $O(1)$.

Our contributions. We have the following contributions:

- We propose two fully distributed ZKP protocols for data-parallel circuits and general circuits, respectively. To construct the schemes, we first propose a distributed polynomial interactive oracle proof (polynomial IOP) and then combine it with a polynomial commitment scheme (PCS) that is distributively computable as well. The polynomial IOP is a bivariate variant of Plonk's [GWC19b] constraint system. To "compile" both IOP schemes by polynomial commitments, we use the bivariate variant of the KZG [PST13] scheme and demonstrate that it is distributively computable. The use of the Lagrange polynomial in our scheme is inspired by a sub-scheme in Caulk [ZBKMNS22].

- We further show that our protocols are robust in the presence of malicious machines. We formalize the notion as *Robust Collaborative Proving Scheme* (RCPS), for the collaborative generation of proofs among sub-provers in a malicious environment. In this setting, the master node is able to verify partial proofs and messages received from other machines before aggregating them to compute the final proof. We show that our protocols are robust under this definition with an additional step of verification. This property is crucial for the applications of distributed zkRollups and zkEVM to exclude malicious participants without ruining the distributed proof generation.

- We implement the fully distributed ZKP system, Pianist, for both data-parallel and general circuits. For the data-parallel version, we report experimental results for the blockchain application of zkRollups. Utilizing rollup circuits generated by the Circom compiler [Cira], we show that Pianist can scale to 8192 transactions on 64 machines with a prover time of 313 seconds. In comparison, the original Plonk scheme can only scale to 32 transactions with a prover time of 95 seconds on a single machine. The communication between each machine and the master machine is only 2144 bytes, and the proof size is 2208 bytes. We observe similar improvements for general circuits. On a circuit of size 2^{25}, it only takes 5s to generate the proof using 32 machines, which is 24.2× faster than Plonk on a single machine, with 2336 Bytes communication and 2816 Bytes proof size.

Organization of the paper. We review the related work in Section 4.1.1 and present the preliminaries in Section 4.2. To explain our protocols, we first introduce our distributed polynomial IOP schemes in Section 4.3 for data-parallel circuits and general circuits. Then in Section 4.4, we present a bivariate variant of the polynomial commitment in [KZG; PST13] to compile our polynomial IOP schemes to SNARKs. In Section 4.5, we formalize the notion of robust collaborative proving scheme (RCPS) and show that our scheme is able to detect malicious machines. We showcase the performance of our system in Section 4.6, and present additional discussions in Section 4.7.

4.1.1 Related works

Zero-knowledge proofs (ZKP) were first introduced by Goldwasser, Micali, and Rackoff in their seminal paper [GMR]. Driven by real-world applications such as blockchains [Ben+14; KMSWP; Xie+22], there has been a rapid development of efficient zkSNARK systems in recent years [PHGR13; BSCTV14c; Gro16; WTSTW18; ZGKPP17a; BSBHR19; BBBPWM18; AHIV17; BSCRSVW19; XZZPS19c; CHMMVW20; GWC19b; Set20; ZXZS20; Zha+21b]. Despite such progress, it remains challenging to scale ZKP protocols to large statements due to their high overhead on the prover running time and memory usage.

Distributed ZKPs. To scale existing ZKP protocols to large-scale circuits, distributed algorithms provide a promising direction. Wu et al. proposed the first distributed zero-knowledge proof protocol called DIZK in [WZCPS18]. DIZK scales the pairing-based zkSNARK in [Gro16] to handle circuits that are 100 times larger on 128 machines compared to a single machine. However, DIZK incurs a high communication cost that is linear in the total size of the circuit among different machines because the scheme runs a distributed number theoretic transformation (NTT) algorithm among the machines using the Map-Reduce framework. Additionally, the recent work of zkBridge [Xie+22] proposed deVirgo, a distributed ZKP protocol based on the ZKP scheme in [ZXZS20], to build bridges between two blockchains using ZKPs. The protocol achieves linear improvement on both the prover time and scalability in the number of machines. However, deVirgo also incurs a linear communication cost among the machines, and the proof size grows with the number of machines. This seems inevitable due to the use of the FRI protocol in [BSBHR18] with Merkle trees [Mer87]. By contrast, our schemes offer optimal linear scalability in prover time and minimal communication among distributed machines simultaneously. We provide comparisons in Table 4.1.

PCD and IVC *Proof-Carrying Data* (PCD [CT10; BCCT13]) is a cryptographic technique that breaks down computation into a sequence of steps. In each step, the prover convinces the verifier not only of the current step's correctness but also of all previous steps. It is an alternative solution for data-parallel circuits when memory is limited. There are generally two ways to achieve PCD: one is from succinct verification, and the other is from accumulation. In the succinct verification approach, for each step, the prover generates a proof for the current step computation and verification for the proof generated from the previous step, as seen in[BCCT13; BSCTV14b; COS20], etc. The accumulation approach postpones and accumulates the verification of SNARK proofs (or some expensive part of it) at each recursion step and proves it all at once at the last step, as demonstrated in [BCMS20; Hal; BCLMS20; KST22; KS22], etc. Although there is no direct correspondence for general circuits, some of these techniques, including but not limited to [KST22; KS22], claim to achieve *Incremental Verifiable Computation* (IVC). IVC focuses on dividing long-running computations into stages that can be verified incrementally. For instance, Nova [KST22] supports proof generation when the computation involves a nondeterministic function f and the result of $f^n(z_0)$. These techniques are widely employed in various applications, however, we identify several drawbacks when compared to our proposed solution. See details in Section 4.7.

Distributed computation from proof aggregation Similar to our approach, aPlonk [ABST22] is a distributed solution based on Plonk that requires prover nodes to share the same Fiat-Shamir randomness, necessitating synchronization several times during the proving process. In their scheme, they propose a multi-polynomial commitment to combine parties' polynomial commitments and attest to the batch opening using a generalized *Inner-Product Argument* (IPA) from [BMV19]. Additionally, they delegate the verification of the constraint system through all evaluations to the prover. We also include the discussion for their protocol in Section 4.7.

Scheme	$\mathcal{P}_i$ time	Comm.	$\lvert\pi\rvert$ & $\mathcal{V}$ time	Robust
DIZK [WZCPS18]	O$(T \log^2 T)$	O(N)	O(1)	✗
deVirgo [Xie+22]	O$(T \log T)$	O(N)	O$(\log^2 N)$	✗
Pianist	O$(T \log T)$	O(M)	O(1)	✓

Table 4.1: Comparisons of our schemes to existing distributed ZKP protocols given M distributed machines on the circuit with M sub-circuits and total N gates, where each sub-circuit has $T = \frac{N}{M}$ gates. $\mathcal{P}_i$ time denotes the prover time per machine, Comm. denotes the total communication among machines, $\lvert\pi\rvert$ denotes the proof size, and $\mathcal{V}$ time denotes the verifier time.

4.2 Preliminaries

Our construction follows the framework proposed in [BFS20] and achieves SNARK by first compiling a public-coin Polynomial IOP into a doubly-efficient public-coin interactive argument of knowledge using a polynomial commitment scheme. Subsequently, the non-interactive property is achieved through the Fiat-Shamir transform. We present the notations and corresponding definitions below

4.2.1 Notations

In our distributed setting, the size of the entire circuit is N, and there are M machines (or users acting as sub-provers) participating in this protocol. Consequently, each party is responsible for generating a proof for a sub-circuit of size $T = \frac{N}{M}$.

We use bivariate polynomials to help construct the constraint system in the scheme. In our constraint system, for the i-th party, it holds its local witness vector $\vec{a}_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,T-1})$. We can transform this witness vector into a univariate polynomial $a_i(X) = \sum_{j=0}^{T-1} a_{i,j} L_j(X)$, where $L_j(X)$ is the Lagrange polynomial defined from the T-th roots of unity, with the close-form $L_j(X) = \frac{\omega_X^j}{T} \cdot \frac{X^T - 1}{X - \omega_X^j}$. Furthermore, we aggregate the witness polynomial from all parties as a bivariate polynomial $A(Y, X) = \sum_{i=0}^{M-1} a_i(X) R_i(Y)$, where $R_i(Y)$ is also the Lagrange polynomial defined by the M-th roots of unity, with the close-form $R_i(Y) = \frac{\omega_Y^i}{M} \cdot \frac{Y^M - 1}{Y - \omega_Y^i}$

Unless specifically stated, for polynomials, we use lowercase letters such as a, b, c to denote the univariate polynomial storing local information, and uppercase letters A, B, C to denote the bivariate polynomial aggregating information throughout the entire circuit. In addition, we use lowercase letters x, y to denote a specific assignment or evaluation for the polynomial, and uppercase letters X, Y to denote unassigned variables.

4.2.2 Interactive Argument

Definition 4.2.1 (Interactive Argument)**.** *We say that* $\mathsf{ARG} = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is an interactive argument of knowledge for a relation* $\mathcal{R}$ *if it satisfies the following completeness and knowledge properties.*

- ***Completeness****: For every adversary* $\mathcal{A}$

$$\Pr\left[\begin{array}{c|c} (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \textit{ or} & \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda) \\ \langle \mathcal{P}(\mathsf{pp}, \mathbb{x}, \mathbb{w}), \mathcal{V}(\mathsf{pp}, \mathbb{x}) \rangle = 1 & (\mathbb{x}, \mathbb{w}) \leftarrow \mathcal{A}(\mathsf{pp}) \end{array}\right] = 1$$

- ***Witness-extended emulation****:* ARG *has witness-extended emulation with knowledge error* κ *if there exists an expected polynomial-time algorithm* $\mathcal{E}$ *such that for every polynomial-size adversary* $\mathcal{A}$ *it* *holds that*

$$\left|\Pr\left[\mathcal{A}(\mathsf{aux},\mathsf{tr}) = 1 : \begin{array}{c}\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)\\ (\mathbb{x},\mathsf{aux}) \leftarrow \mathcal{A}(\mathsf{pp})\\ \mathsf{tr} \leftarrow \langle \mathcal{A}(\mathsf{aux}), \mathcal{V}(\mathsf{pp},\mathbb{x})\rangle\end{array}\right] - \Pr\left[\begin{array}{c}\mathcal{A}(\mathsf{aux},\mathsf{tr}) = 1\\ \textit{and if } \mathsf{tr} \textit{ is accepting}\\ \textit{then } (\mathbb{x},\mathbb{w}) \in \mathcal{R}\end{array} : \begin{array}{c}\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)\\ (\mathbb{x},\mathsf{aux}) \leftarrow \mathcal{A}(\mathsf{pp})\\ (\mathsf{tr},\mathbb{w}) \leftarrow \mathcal{E}^{\mathcal{A}(\mathsf{aux})}(\mathsf{pp},\mathbb{x})\end{array}\right]\right| \le \kappa(\lambda)$$

Above $\mathcal{E}$ has oracle access to (the next-message functions of) $\mathcal{A}(\mathsf{aux})$.

If the interactive argument of knowledge protocol ARG is public-coin, is has been shown that by the Fiat-Shamir transform [FS86], we can derive a non-interactive argument of knowledge from ARG. If the scheme further satisfies the following property:

- **Succinctness**. The proof size is $|\pi| = \mathsf{poly}(\lambda, \log|C|)$ and the verification time is $\mathsf{poly}(\lambda, |\mathbb{x}|, \log|C|)$,

then it is a *Succinct Non-interactive Argument of Knowledge (SNARK)*.

For the applications of zkRollups and zkEVM, we only need a SNARK that is complete, sound, and succinct. Our constructions can be made zero-knowledge via known transformations with random masks and we omit the details in this paper.

4.2.3 Polynomial Interactive Oracle Proof

Definition 4.2.2 (Public-coin Polynomial Interactive Oracle Proof [BFS20]). *Let $\mathcal{R}$ be a binary relation and $\mathbb{F}$ be a finite field. Let $X = (X_1, \ldots, X_\mu)$ be a vector of μ indeterminates. A (μ, d) Polynomial IOP for $\mathcal{R}$ over $\mathbb{F}$ with soundness error ϵ and knowledge error δ consists of two stateful PPT algorithms, the prover $\mathcal{P}$, and the verifier $\mathcal{V}$, that satisfy the following requirements:*

- ***Protocol syntax**. For each i-th round there is a prover state $\mathsf{st}_i^{\mathcal{P}}$ and a verifier state $\mathsf{st}_i^{\mathcal{V}}$. For any common input x and $\mathcal{R}$ witness w, at round 0 the states are $\mathsf{st}_0^{\mathcal{P}} = (x, w)$ and $\mathsf{st}_0^{\mathcal{V}} = x$. In the i-th round (starting at $i = 1$) the prover outputs a single proof oracle $\mathcal{P}(\mathsf{st}_{i-1}^{\mathcal{P}}) \to \pi_i$, which is a polynomial $\pi_i(X) \in \mathbb{F}[X]$. The verifier deterministically computes the query matrix $i \in \mathbb{F}^{\mu\times\ell}$ from its state and a string of public random bits $\mathsf{coins}_i \leftarrow \{0,1\}^*$, i.e, $\mathcal{V}(\mathsf{st}_{i-1}^{\mathcal{V}}, \mathsf{coins}_i) \to \Sigma_i$. This query matrix is interpreted as a list of ℓ points in $\mathbb{F}^\mu$ denoted $(\sigma_{i,1}, \ldots, \sigma_{i,\ell})$. The oracle π_i is queried on all points in this list, producing the response vector $(\pi_i(\sigma_{i,1}), \ldots, \pi_\ell(\sigma_{i,\ell})) = a_i \in \mathbb{F}^{1\times\ell}$. The updated prover state is $\mathsf{st}_i^{\mathcal{P}} \leftarrow (\mathsf{st}_{i-1}^{P}), \Sigma_i)$ and verifier state is $\mathsf{st}_i^{V} \leftarrow (\mathsf{st}_{i-1}^{\mathcal{V}}, \Sigma_i, a_i)$. Finally, $\mathcal{V}(\mathsf{st}_t^{\mathcal{V}})$ returns 1 or 0.*

  (Extensions: multiple and prior round oracles; various arity. The syntax can be naturally extended such that multiple oracles are sent in the i-th round; that the verifier may query oracles sent in the i-th round or earlier; or that some of the oracles are polynomials in fewer variables than μ.)

Furthermore, a Polynomial IOP is stateless if for each $i \in [t]$, $\mathcal{V}(\mathsf{st}_{i-1}^{\mathcal{V}}, \mathsf{coins}_i) = \mathcal{V}(i, \mathsf{coins}_i)$.

4.2.4 Polynomial Commitment Scheme (PCS)

Definition 4.2.3 (Polynomial commitment scheme (PCS))**.** *A Polynomial commitment scheme* Γ *is a tuple* $\Gamma = (\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *of PPT algorithms where:*

- $\mathsf{KeyGen}(1^\lambda, \mathcal{F}) \to \mathsf{pp}$ *generates public parameters* pp*;*
- $\mathsf{Commit}(f, \mathsf{pp}) \to \mathsf{com}_f$ *takes a secret polynomial* $f(\mathbf{X})$ *where* $X = (X_0, \dots, X_{\mu-1})$ *and outputs a public commitment* com_f*;*
- $\mathsf{Open}(\mathsf{com}_f, \mathbf{x}, \mathsf{pp}) \to (z, \pi_f)$ *evaluates the polynomial* $y = f(\mathbf{X})$ *on a point* $\mathbf{x}$ *and generate a proof* π_f*;*
- $\mathsf{Verify}(\mathsf{com}_f, \mathbf{x}, z, \pi_f, \mathsf{pp}) \to b \in \{1, 0\}$ *is a protocol between the prover* $\mathcal{P}$ *and verifier* $\mathcal{V}$*, verifying whether* $f(\mathbf{x})$ *is* z *through* pp*,* com_f *and* π_f*;*

which satisfies the following properties:

- ***Completeness.*** *For any polynomial* $f \in \mathcal{F}$ *and* $\mathbf{x} \in \mathbb{F}^\mu$*, the following probability is* 1.

$$\Pr\left[\mathsf{Verify}(\mathsf{com}_f, \mathbf{x}, z, \pi_h, \mathsf{pp}) = 1 \;:\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \mathcal{F}) \\ \mathsf{com}_f \leftarrow \mathsf{Commit}(f, \mathsf{pp}) \\ (z, \pi_f) \leftarrow \mathsf{Open}(f, \mathbf{x}, \mathsf{pp}) \end{array}\right]$$

- ***Knowledge soundness.*** *For any PPT adversary* $\mathcal{P}^*$*, there exists a PPT extractor* $\mathcal{E}$ *with access to* $\mathcal{P}^*$*'s messages during the protocol, the following probability is* $\mathsf{negl}(\lambda)$.

$$\Pr\left[\begin{array}{rl} \mathsf{Verify}(\mathsf{com}^*, \mathbf{x}^*, z^*, \pi^*, \mathsf{pp}) = 1 & \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \mathcal{F}) \\ \wedge \mathsf{com}^* = \mathsf{Commit}(f^*, \mathsf{pp}) & : (z^*, \mathbf{x}^*) \leftarrow \mathcal{P}^*(1^\lambda, \mathsf{pp}) \\ \wedge f^*(\mathbf{x}^*) \neq z^* & (com^*, \pi^*) \leftarrow \mathcal{P}^*(1^\lambda, \mathsf{pp}) \\ & f^* \leftarrow \mathcal{E}^{\mathcal{P}^*(\cdot)}(1^\lambda, \mathsf{pp}) \end{array}\right]$$

It is worth noting that in [BFS20], although they demonstrate that if the polynomial commitment protocol satisfies witness-extended emulation, the compiled interactive argument also inherits this knowledge property. However, they also point out that it has been proven in [Lin01] that every knowledge sound protocol satisfies witness-extended emulation as well.

4.3 Constraint System And Distributed Polynomial IOP Protocol

In this and the following sections, we demonstrate how to construct our distributively computable SNARK for data-parallel circuits (which accommodate various sub-circuits) and arbitrary general circuits. In both settings, we distribute the input and computation across M machines, each capable of evaluating one sub-circuit C_i of size $T = \frac{N}{M}$ locally. In this section, we first present the constraint system, and then design an IOP protocol proving the constraints. We prove that our IOP protocol has knowledge soundness and can be transformed into a distributed double-efficient interactive argument of knowledge after compiling with a distributive computable bivariate PCS. In the next section, we will instantiate our protocol with bivariate KZG and provide a detailed analysis of proving time, verification time, proof size, and communication complexity.

Before diving into the details, we first explain our intuition. We opt for the distributed system to avoid the substantial overhead introduced by recursive proof (see Section 4.7 for a detailed discussion). PCD-and-IVC-based solutions rely on recursive proofs because they handle each sub-circuit in a separate proof waiting to be aggregated. Instead, we treat all sub-circuits as a whole and exploit the succinctness of SNARK, resulting in a small proof size and verification time. However, DIZK [WZCPS18] shows that directly applying distribution techniques to the original univariate SNARK system leads to linear communication costs due to the significantly interleaving network required to run the NTT algorithm. Taking both hazards into account, we propose a solution leveraging bivariate polynomial constraints to both ”split” the NTT instances, avoiding substantial communication, and ”combine” the proof for each sub-circuit as a whole, eliminating the need for expensive aggregation costs. The details are as follows.

4.3.1 Arithmetic Constraint System for Each Party

Our constraint system inherits the original Plonk [GWC19b]. The original Plonk works for a fan-in-two arithmetic circuit, where each gate takes at most two inputs. In Plonk, the left input, the right input, and the output of each gate are encoded by three univariate polynomials respectively. The verifier can check the computation of each gate by a polynomial equation, which we refer to as the *gate constraint*. Additionally, the verifier also checks that the input and output of the gates are connected correctly as defined by the structure of the circuit, which we refer to as the *copy constraint*.

Gate Constraint For the i-th party, let $a_{i,j}, b_{i,j}$ and $o_{i,j}$ be the left input, right input, and output of gate j of the sub-circuit C_i, for $j = 0, \dots, T-1$. We define a polynomials $a_i(X) = \sum_{j=0}^{T-1} a_{i,j} L_j(X)$ where $\{L_j(X)\}_j$ is the Lagrange polynomials defined from the T-th roots of unity. The coefficient representation of $a_i(X)$ can be computed using polynomial interpolation and the complexity is $O(T \log T)$ via NTT algorithm. Similarly, we define polynomials $b_i(X)$ and $o_i(X)$ using $(b_{i,j})_j$ and $(o_{i,j})_j$. If gate j is an addition gate, then $a_{i,j} + b_{i,j} = o_{i,j}$, and thus $a_i(\omega_X^j) + b_i(\omega_X^j) = o_i(\omega_X^j)$; if gate j is a multiplication gate, then $a_{ij} \cdot b_{ij} = o_{ij}$, and thus $a_i(\omega_X^j) \cdot b_i(\omega_X^j) = o_i(\omega_X^j)$. Following the design of Plonk, we can write the relationship of all gates as one polynomial in Equation 4.1.

$$\begin{aligned} g_i(X) := {} & q_{a,i}(X)a_i(X) + q_{b,i}(X)b_i(X) + q_{o,i}(X)o_i(X) \\ & + q_{ab,i}(X)a_i(X)b_i(X) + q_{c,i}(X) = 0. \end{aligned} \tag{4.1}$$

Here the polynomials $q_{a,i}(X)$, $q_{b,i}(X)$, $q_{o,i}(X)$, $q_{ab,i}(X)$, $q_{c,i}(X)$ are defined by the structure of C_i satisfying

- **Addition gate:** $q_{a,i}(\omega_X^j) = 1$, $q_{b,i}(\omega_X^j) = 1$, $q_{o,i}(\omega_X^j) = -1$, $q_{ab,i}(\omega_X^j) = 0$, $q_{c,i}(\omega_X^j) = 0$.
- **Multiplication gate:** $q_{a,i}(\omega_X^j) = 0$, $q_{b,i}(\omega_X^j) = 0$, $q_{o,i}(\omega_X^j) = -1$, $q_{ab,i}(\omega_X^j) = 1$, $q_{c,i}(\omega_X^j) = 0$.
- **Public input:** $q_{a,i}(\omega_X^j) = 0$, $q_{b,i}(\omega_X^j) = 0$, $q_{o,i}(\omega_X^j) = -1$, $q_{ab,i}(\omega_X^j) = 0$, $q_{c,i}(\omega_X^j) = \mathsf{in}_{i,j}$ if the j-th gate in C_i is a public input gate with the value of $\mathsf{in}_{i,j}$.

In this way, the correct evaluation of the circuit is equivalent to Equation 4.1 being 0 for all $X \in \Omega_X$, where Ω_X denotes the set $\left\{\omega_X^0, \cdots, \omega_X^{T-1}\right\}$.

Copy Constraint In addition to checking the gate constraint, the verifier also needs to check that the connections of wires are correct as defined by the circuit. In particular, there are redundancies in the vectors $a_{i,j}, b_{i,j}$ and $o_{i,j}$, since the output of one gate is the input of other gates in the circuit. The method used in Plonk is derived from a product argument, which can show that if a set of values $\{f_i\}_{i\in\mathcal{I}}$ are identical, then the following two sets are equal:

$$\{(f_i, i)\}_{i\in\mathcal{I}} = \{(f_i, \sigma(i))\}_{i\in\mathcal{I}}$$

where σ defines a cycle connecting all indexes. The protocol reduces the argument to 2 polynomial equations.

The details of the permutation argument is as follows: $\forall X \in \Omega_X, a_i(\sigma_i(X)) = a_i'(X)$, where $a_i(X)$ and $a_i'(X)$ are two univariate polynomials in $\mathbb{F}$ and σ_i is a public permutation from Ω_X to Ω_X. Particularly, in the protocol checking the consistency of $a_i(X), b_i(X), o_i(X)$ in the gate constraint, given two random points $\eta, \gamma \in \mathbb{F}$ from the verifier, the prover defines the running product polynomial $z_i(X)$ on $\mathbb{F}$ defined as follows:

$$z_i(\omega_X^j) := \prod\nolimits_{k=0}^{j-1} \frac{f_i(\omega_X^k)}{f_i'(\omega_X^k)} \tag{4.2}$$

where for simplicity, the notation of $f_i(X)$ and $f_i'(X)$ are used to indicate

$$\begin{aligned} f_i(X) :=&(a_i(X) + \eta\sigma_{a,i}(X) + \gamma)(b_i(X) + \eta\sigma_{b,i}(X) + \gamma) \\ &(o_i(X) + \eta\sigma_{c,i}(X) + \gamma), \\ f_i'(X) :=&(a_i(X) + \eta k_a X + \gamma)(b_i(X) + \eta k_b X + \gamma) \\ &(o_i(X) + \eta k_o X + \gamma), \end{aligned} \tag{4.3}$$

where $k_a = 1$, k_b is any quadratic non-residue, and k_o is a quadratic non-residue not contained in $k_b\Omega_X$. The j-th cell in a_i, b_i, c_i is denoted by ω_X^j, $k_1\omega_X^j$, $k_2\omega_X^j$, respectively, and $\sigma_{a,i}(\omega_X^j)$ denotes the destination that the j-th cell in a_i is mapped to ($\sigma_{b,i}$ and $\sigma_{c,i}$ are defined similarly). The goal of the permutation argument is to prove $\prod_{k=0}^{T-1} \frac{f_i(\omega_X^k)}{f_i'(\omega_X^k)} = 1$, leading to the following constraints:

$$p_{i,0}(X) := L_0(X)(z_i(X) - 1) \tag{4.4}$$

$$p_{i,1}(X) := z_i(X)f_i(X) - z_i(\omega_X X)f_i'(X) \tag{4.5}$$

which equals 0 when $X \in \Omega_X$.

Finally, since the constraints $g_i(X), p_{i,0}(X)$ and $p_{i,1}(X)$ all equal 0 when $X \in \Omega_X$, then given a random challenge λ from the verifier, there must exist a quotient polynomial $h_i(X)$ satisfying

$$g_i(X) + \lambda p_{i,0}(X) + \lambda^2 p_{i,1}(X) = V_X(X)h_i(X), \tag{4.6}$$

where $V_X(X) = X^T - 1$.

Polynomial IOP protocol for Plonk. In the original Plonk, the IOP process sends oracles to the verifier in three rounds. Suppose the verifier knows the structure of the circuit and have oracles of $\left\{q_{\{a,b,o,ab,c\}}(X), \sigma_{\{a,b,o\}}(X)\right\}$. In the first round, the prover sends the polynomial oracles for $a(X), b(X), o(X)$. In the second round, after receiving random challenge η, γ from the verifier, the prover constructs the oracle $z(X)$ for the verifier. In the remaining round, with the randomness λ from the verifier, the prover computes the quotient polynomial $h(X)$ and sends its oracle to the verifier. After having access to all the oracles, the verifier queries them on a random point $X = \alpha$ and an extra point $X = \omega_X \cdot \alpha$ for $z(X)$. With the evaluation, the verifier can verify all the constraints.

4.3.2 Constraint System for Data-parallel Circuit

In this section, we show how to aggregate the polynomials from all separated sub-circuits into a single bivariate polynomial and remain the constraint structure. Inheriting the general-purpose arithmetic constraints from Plonk, it is clear that we not only have a constraint system proving data-parallel circuits but also for a more general case: we allow the sub-circuits to be different.

Intuitively, we can use the powers of another variable Y to randomly combine the polynomials from different parties. For example $A(Y,X) = \sum_{i=0}^{M-1} Y^i a_i(X)$, where $a_i(X)$ is hold by the i-th party. However, when we leverage this formula in the polynomial formula, such as $\left(\sum_{i=0}^{M-1} Y^i a_i(X)\right)\left(\sum_{i=0}^{M-1} Y^i b_i(X)\right) - \left(\sum_{i=0}^{M-1} Y^i c_i(X)\right)$, the cross-terms with the form $Y^i a_i(X) \cdot Y^j b_j(X)$ for $i \neq j$ in the expansion will be very annoying. To avoid the cross-term, instead, we combine the polynomials with Lagrange polynomials $R_i(Y)$. This idea is inspired by a sub-scheme in the recent work Caulk [ZBKMNS22]. In particular, for each univariate polynomial in Equation 4.1, Equation 4.4, Equation 4.5 and Equation 4.6, i.e., $s_i \in \{q_{a,i}, q_{b,i}, q_{o,i}, q_{ab,i}, q_{c,i}, \sigma_{a,i}, \sigma_{b,i}, \sigma_{o,i}, a_i, b_i, o_i, z_i, h_i\}$, we define a bivariate polynomial as

$$S(Y,X) = \sum_{i=0}^{M-1} R_i(Y) s_i(X). \tag{4.7}$$

Then we have an aggregated gate constraint:

$$\begin{aligned} G(Y,X) &:= Q_a(Y,X)A(Y,X) + Q_b(Y,X)B(Y,X) \qquad (4.8)\\ &\quad + Q_{ab}(Y,X)A(Y,X)B(Y,X) \\ &\quad + Q_o(Y,X)O(Y,X) + Q_c(Y,X) \\ P_0(Y,X) &:= L_0(X)(Z(Y,X) - 1) \qquad (4.9)\\ P_1(Y,X) &:= Z(Y,X) \prod_{S \in \{A,B,O\}} (S(Y,X) + \eta\sigma_a(Y,X) + \gamma) \qquad (4.10)\\ &\quad - Z(Y,\omega_X X) \prod_{S \in \{A,B,O\}} (S(Y,X) + \eta k_s X + \gamma) \end{aligned}$$

Then after transforming Equation 4.6, we have

$$G(Y,X) + \lambda P_0(Y,X) + \lambda^2 P_1(Y,X) - V_X(X)H_X(Y,X) \tag{4.11}$$

which equals 0 for all $Y \in \Omega_Y$. It is no hard to see that this is equivalent to Equation 4.6 holds for all $i \in [M]$, because by the definition of the Lagrange polynomial $R_i(Y)$, there is only one non-zero term $g_i(X)$, $p_{i,0}(X)$ and $p_{i,1}(X)$ in Equation 4.6 when $Y = \omega_Y^i$. Therefore, evaluating Equation 4.11 at $Y = \omega_Y^i$ is exactly the same as Equation 4.6 for C_i.

Finally, to check Equation 4.11 vanishes on $Y \in \Omega_Y$, we compute $H_Y(Y,X)$ such that

$$\begin{aligned} &G(Y,X) + \lambda P_0(Y,X) + \lambda^2 P_1(Y,X) - V_X(X)H_X(Y,X) \\ &= V_Y(Y)H_Y(Y,X) \end{aligned} \tag{4.12}$$

where $V_Y(Y) = Y^M - 1$. This concludes the bivariate constraint system in our solution.

A sketch of the distributed IOP for the previous system. In Protocol 3, we introduce the polynomial IOP protocol for the previous constraint system (excluding the orange characters). From this protocol, we observe that, aside from sending and assisting the verifier with querying the polynomial oracles, the prover only needs to distributively maintain each oracle. We will later prove that this property is sufficient to construct a distributed SNARK proof generation. We also observe that this property trivially holds for all polynomials except $H_Y(Y, X)$. To circumvent this obstacle, the prover receives an opening point α from the verifier and only sends the univariate oracle $H_Y(Y, \alpha)$. We claim that after this modification, the protocol remains knowledge-sound and can be distributively computed. For simplicity, we provide the strict proof in Section 4.3.4 after explaining the system for general circuits.

Remark 4.3.1. For the witness generation, since all sub-circuits are separated, each party can generate its witness locally.

Remark 4.3.2. Although we assume the sub-circuits are independent of each other, it is easy to observe that if we introduce custom gates and rotation along with the variable Y, then we can support some simple connections among different sub-circuits. In addition, we can also introduce local lookup arguments in our constraint system. Further discussion on custom gates and lookup arguments are in Section 4.7.

4.3.3 Constraint System for General Circuit

In this section, we show the great potential of our system by generalizing it to generate proofs for arbitrary circuits. Recall that in the original Plonk, it leverages $\sigma_a(X)$, $\sigma_b(X)$, and $\sigma_c(X)$ to navigate the next wire in the circuit with equal value, and computes the running product polynomial $z(X)$ as a helper polynomial to prove the product of $\prod_{k=0}^{j-1} \frac{f(\omega_X^k)}{f'(\omega_X^k)} = 1$. Similarly, we present how to indicate the position to the next wire and how to construct the product proof through the whole circuit.

Since we need to indicate which sub-circuit the next wire locates, we define $\{(\sigma_{Y,s,i}(X), \sigma_{X,s,i}(X))\}_{s\in\{a,b,o\}}$ as: if for the i-th party, for the j-th entry in the polynomial s is mapped to the i'-th party, j'-th entry in the polynomial s' polynomial, then $\left(\sigma_{Y,s,i}(\omega_X^j), \sigma_{X,s,i}(\omega_X^j)\right) = \left(\omega_Y^{i'}, k_{s'}\omega_X^{j'}\right)$. Therefore, we need to prove that

$$\prod_{i=0}^{M-1} \prod_{j=0}^{T-1} \frac{f_i(\omega_X^j)}{f_i'(\omega_X^j)} = 1 \tag{4.13}$$

where

$$\begin{aligned} f_i(X) :=& (a_i(X) + \eta_Y \sigma_{Y,a,i}(X) + \eta_X \sigma_{X,a,i}(X) + \gamma) \\ & (b_i(X) + \eta_Y \sigma_{Y,b,i}(X) + \eta_X \sigma_{X,b,i}(X) + \gamma) \\ & (o_i(X) + \eta_Y \sigma_{Y,o,i}(X) + \eta_X \sigma_{X,o,i}(X) + \gamma) \\ f_i'(X) :=& (a_i(X) + \eta_Y Y + \eta_X X + \gamma) \\ & (b_i(X) + \eta_Y Y + \eta_X k_1 X + \gamma) \\ & (o_i(X) + \eta_Y Y + \eta_X k_2 X + \gamma) \end{aligned}$$

Then we show how to construct the constraints for the product argument. Similarly, each party remains the running product $z_i(X)$, however for the one after the last entry, $z_i^* = z_i(\omega_X^{T-1})\frac{f_i(\omega_X^{T-1})}{f_i'(\omega_X^{T-1})}$ no longer equals 1. Therefore, comparing with Equation 4.4 and Equation 4.5, we have the following constraints instead:

$$\begin{aligned} p_{i,0}(X) :=& L_0(X)(z_i(X) - 1) && (4.14) \\ p_{i,1}(X) :=& (1 - L_{T-1}(X)) && (4.15) \\ & \cdot \Big(z_i(X)f_i(X) - z_i(\omega_X X)f_i'(X)\Big) \end{aligned}$$

After constructing z_i, each party will send the product of their slices z_i^* to the master node, which then generates another helper polynomial $W(X)$ to denote the running product through $(z_0^*, \ldots, z_{M-1}^*)$. Therefore, we have two more constraints that for $0 \le i < M$:

$$\begin{aligned} p_{i,2} :=& w_0 - 1 \text{ which is 0 for all } i && (4.16) \\ p_{i,3}(X) :=& L_{N-1}(X) && (4.17) \\ & \cdot \Big(w_i z_i(X) f_i(X) - w_{(i+1)\%M} f_i'(X)\Big) \end{aligned}$$

Therefore we compute $h_i(X)$ and $H_X(Y,X)$ through the following equation instead:

$$\begin{aligned} h_i(X) &= \frac{g_i(X) + \lambda p_{i,0} + \lambda^2 p_{i,1} + \lambda^4 p_{i,3}}{X^T - 1} \\ H_X(Y,X) &= \sum_{i=0}^{M-1} R_i(Y) h_i(X) \end{aligned} \qquad (4.18)$$

Finally, by multiplying polynomials with $R_i(Y)$, the permutation argument becomes

$$\begin{aligned} P_0(Y,X) :=& L_0(X)(Z(Y,X) - 1) && (4.19) \\ P_1(Y,X) :=& (1 - L_{N-1}(X)) && (4.20) \\ & \cdot \Big(Z(Y,X)F(Y,X) - Z(Y,\omega_X X)F'(Y,X)\Big) \\ P_2(Y) :=& R_0(Y)(W(Y) - 1) && (4.21) \\ p_3(Y,X) :=& L_{N-1}(X) && (4.22) \\ & \cdot \Big(W(Y)Z(Y,X)F(Y,X) - W(\omega_Y Y)F'(Y,X)\Big) \end{aligned}$$

where $F(Y,X)$ and $F'(Y,X)$ are just notations to denote

$$\begin{aligned} F(Y,X) :=& \prod_{S \in \{A,B,O\}} (S(Y,X) + \eta_Y \sigma_{Y,s}(Y,X) \\ & + \eta_X \sigma_{X,s}(Y,X) + \gamma) \\ F'(Y,X) :=& \prod_{S \in \{A,B,O\}} (S(Y,X) + \eta_Y Y + \eta_X k_s X + \gamma) \end{aligned}$$

By combining with the same gate constraint as for data-parallel circuits, we finally have the equation to define $H_Y(Y,X)$, which concludes our constraint system for general circuits.

$$\begin{aligned} & G(Y,X) + \lambda P_0(Y,X) + \lambda^2 P_1(Y,X) + \lambda^3 P_2(Y) \\ & + \lambda^4 P_3(Y,X) = V_X(Y,X)H_X(Y,X) + V_Y(Y)H_Y(Y,X) \end{aligned} \qquad (4.23)$$

4.3.4 Distributedly Computable Polynomial IOP Protocol

We present our polynomial IOP protocol in Protocol 3. The text in orange denotes the additional steps for general circuits. We have the following theorem:

Theorem 4.3.3. *Protocol 3 is a polynomial IOP protocol for $\mathcal{R}$ with negligible knowledge error.*

The proof is in Appendix 4.8.

Theorem 4.3.4. *Protocol 3 is a Polynomial IOP protocol that can be compiled into a distributedly computable double efficient non-interactive proof that has witness-extended emulation, using a distributed computable PCS, with only a constant increase in communication and $O(N \log T + M \log M)$ additional proving time compared to the PCS used.*

Proof. We prove the theorem as follows:

Security. In [BFS20], they provide a detailed proof demonstrating that if the polynomial commitment scheme Γ has witness-extended emulation, and if the t-round Polynomial IOP for a relation $\mathcal{R}$ has negligible knowledge error, an interactive argument for $\mathcal{R}$ with witness-extended emulation exists.

We present a sketch of the proof: for an arbitrary adversary prover $\mathcal{P}^{\mathsf{IP}}$ for the IP scheme, we can construct an adversary prover $\mathcal{P}^{\mathsf{IOP}}$. With DKZG that guarantees witness-extended emulation, it enables $\mathcal{P}^*\mathsf{IOP}$ to simulate the transcript with $\mathcal{P}^*\mathsf{IP}$ to extract polynomials. After sending the oracles to $\mathcal{V}\mathsf{IOP}$ and receiving challenges, $\mathcal{P}_{\mathsf{IOP}}$ can rewind the transcript with $\mathcal{P}_{\mathsf{IP}}$ to insert the same randomness from $\mathcal{V}_{\mathsf{IOP}}$. Consequently, due to the knowledge soundness of PCS, the reduction succeeds with high probability. Then from the knowledge soundness proven in Theorem 4.3.3, an upper bound of the knowledge error for the IP protocol is achieved.

We kindly refer to [BFS20] for the complete proof.

Efficiency. To analyze the extra communication, for all polynomials except $H_Y(Y, X)$ and $W(Y)$, it is divided into slices and stored in each party. Then with a PCS which can generate commitments and proofs in this scenario, we can handle all oracle constructions and queries to those polynomials. As for $W(Y)$, it is computed by $\mathcal{P}_0$ from z_i^* received from the i-th parties. Therefore, it can be easily computed from constant message exchange between each node to the master. While the difficulty occurs when computing $H_Y(Y, X)$, instead of computing the full description, $\mathcal{P}_0$ only deals with it after receiving the first opening coordinate $X = \alpha$ and computes $H_{Y,\alpha}(Y) = H_Y(Y, \alpha)$. Therefore, for $0 \leq i < M$, $\mathcal{P}_i$ sends $s_i(\alpha)$ for $s \in \{q_a, q_b, q_o, q_{ab}, q_c, a, b, c, z, h_x\}$ and recover the corresponding polynomial $S(Y, \alpha)$, $\mathcal{P}_0$ can compute $G(Y, \alpha)$, $P_{\{0,1,2,3\}}(Y, \alpha)$ and compute $H_{Y,\alpha}(Y)$ according to Equation 4.23. Additionally considering the distribution of random challenges, the compiled polynomial IOP protocol only has a constant number of more communication than PCS.

As for the proving time, it requires at most $O(T \log T)$ together to compute $z_i(X), h_i(X)$ for each party and $O(M \log M)$ for $\mathcal{P}_0$ to compute $W(Y)$ and $H_{Y,\alpha}(Y)$, the extra proving time is up to $O(T \log T + M \log M)$ for a single machine and $O(N \log T + M \log M)$ in total. □

Protocol 3 (**Polynomial IOP for Data-parallel and General Circuits**). Suppose the circuit structure is known by $\mathcal{P}$ and $\mathcal{V}$, therefore, $\mathcal{V}$ knows the following oracles:

- $\{Q_a(Y,X), Q_b(Y,X), Q_o(Y,X), Q_{ab}(Y,X), Q_c(X)\}$.
- $\left\{\sigma_{Y,a}(Y,X), \sigma_{Y,b}(Y,X), \sigma_{Y,o}(Y,X), \sigma_{X,a}(Y,X), \sigma_{X,b}(Y,X), \sigma_{X,o}(Y,X)\right\}$.

When generating proof for a new instance, $\mathcal{P}$ and $\mathcal{V}$ go through the following rounds:

1. $\mathcal{P}$ sends the oracles of $\{A(Y,X), B(Y,X), C(Y,X)\}$ to $\mathcal{V}$.
2. After receiving η_Y, η_X, γ from $\mathcal{V}$, $\mathcal{P}$ sends the oracle of $Z(Y,X)$ and $W(Y)$ to $\mathcal{V}$.
3. After receiving λ from $\mathcal{V}$, $\mathcal{P}$ computes $H_X(Y,X) = \sum_{i=0}^{M-1} R_i(Y) \cdot \frac{g_i(X)+\lambda p_{i,0}(X)+\lambda^2 p_{i,1}(X)+\lambda^4 p_{i,3}(X)}{X^T-1}$ and sends the oracle to $\mathcal{V}$.
4. After receiving α from $\mathcal{V}$, $\mathcal{P}$ computes $H_{Y,\alpha}(Y) = \frac{G(Y,\alpha)+\lambda P_0(Y,\alpha)+\lambda^2 P_1(Y,\alpha)+\lambda^3 P_2(Y)+\lambda^4 P_3(Y,\alpha)-(\alpha^T-1)H_X(Y,\alpha)}{Y^M-1}$ and sends the oracle to $\mathcal{V}$.
5. $\mathcal{V}$ queries all oracles on $X = \alpha, Y = \beta$ and assign the evaluations to the corresponding polynomials in Equation 4.12 or Equation 4.23. If this equation holds, then $\mathcal{V}$ output `1`, otherwise `0`.

4.4 Fully Distributed SNARK

In Theorem 4.3.4, we show that with a distributed PCS, we can build a fully distributed double-efficient interactive argument of knowledge protocol from distributed polynomial IOP. In this section, we instantiate Theorem 4.3.4 by a distributed bivariate KZG.

4.4.1 Distributed KZG

In this section, we present a distributedly computable PCS based on a bivariate variant the KZG scheme in [KZG; PST13].

In our distributed setting, the total size of the polynomial is N, and there are M machines of $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ with part of the polynomial on each machine of size $T = N/M$. The goal of the fully distributed polynomial commitments is to accelerate the prover time by B times while keeping the communication complexity among the machine's minimum. Moreover, both the proof size and the verifier time should remain the same as the original polynomial commitment schemes. We present the distributed protocol in Protocol 4.

Theorem 4.4.1. *Given polynomial $f(Y,X) \in \mathbb{F}^M \times \mathbb{F}^{\frac{N}{M}}$, Protocol 4 is PCS satisfying completeness and knowledge soundness. The total proving computation consists of $O(N)$ group operations, while $O\left(\frac{N}{M}\right)$ group operations for each node and $O\left(\frac{N}{M}+M\right)$ group operations for the master node. The total commu-*

Protocol 4 (**Distributed Bivariate Polynomial Commitment**). Suppose $\mathcal{P}$ has M machines of $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and suppose $\mathcal{P}_0$ is the master node. Given the bivariate polynomial $f(Y, X) = \sum_{i=0}^{M-1} f_{i,j} R_i(Y) L_j(X)$, each machine holds $f_i(X) = \sum_{j=0}^{T-1} f_{i,j} L_j(X)$ The protocol proceeds as follows.

- $\mathsf{DKZG.KeyGen}(1^\lambda, M, T)$: Generate $\mathsf{pp} = \left(g, g^{\tau_X}, g^{\tau_Y}, (U_{i,j})_{\substack{0 \le i < M,\\ 0 \le j < T}} = \left(g^{R_i(\tau_Y) L_j(\tau_X)} \right)_{\substack{0 \le i < M,\\ 0 \le j < T}} \right)$, with trapdoor τ_Y and τ_X. Let $\mathcal{P}, \mathcal{V}$ hold pp.
- $\mathsf{DKZG.Commit}(f, \mathsf{pp})$: In the commitment phase, each $\mathcal{P}_i$ computes the commitment $\mathsf{com}_{f_i} = \prod_{j=0}^{T-1} U_{i,j}^{f_{i,j}}$ and sends it to $\mathcal{P}_0$, where $f_{i,j}$ is the j-th entry in the evaluation representation of $f_i(X)$. After receiving commitments from others, $\mathcal{P}_0$ computes $\mathsf{com}_f = \prod_{i=0}^{M-1} \mathsf{com}_{f_i}$.
- $\mathsf{DKZG.Open}(f, \beta, \alpha, \mathsf{pp})$:

  1. Each $\mathcal{P}_i$ computes $f_i(\alpha)$ and $q_0^{(i)}(X) = \frac{f_i(X) - f_i(\alpha)}{X - \alpha}$. $\mathcal{P}_i$ computes $\pi_0^{(i)} = g^{R_i(\tau_Y) q_0^{(i)}(\tau_X)}$ using the public parameters and sends $f_i(\alpha), \pi_0^{(i)}$ to $\mathcal{P}_0$.
  2. After receiving $\left\{ \left(f_i(\alpha), \pi_0^{(i)} \right) \right\}_{0 \le i < M}$, $\mathcal{P}_0$ computes $\pi_0 = \prod_{i=0}^{M-1} \pi_0^{(i)}$, and also recover $f(Y, \alpha) = \sum_{i=0}^{M-1} R_i(Y) f_i(\alpha)$.
  3. $\mathcal{P}_0$ computes $f(\beta, \alpha)$ and $q_1(Y) = \frac{f(Y,\alpha) - f(\beta,\alpha)}{Y - \beta}$. $\mathcal{P}_0$ computes $\pi_1 = g^{q_1(\tau_Y)}$ and sends $z = f(\beta, \alpha)$ and $\pi_f = (\pi_0, \pi_1)$ to $\mathcal{V}$.

- $\mathsf{DKZG.Verify}(\mathsf{com}_f, \beta, \alpha, z, \pi_f, \mathsf{pp})$: $\mathcal{V}$ parses $\pi_f = (\pi_0, \pi_1)$, and checks if $e\left(\mathsf{com}_f / g^z, g\right) \overset{?}{=} e\left(\pi_0, g^{\tau_X - \alpha}\right) e\left(\pi_1, g^{\tau_Y - \beta}\right)$. It outputs $\mathtt{1}$ if the check passes, and $\mathtt{0}$ otherwise.

nication between $\mathcal{P}_i$ and $\mathcal{P}_0$ is $O(1)$. The commitment and proof size are both $O(1)$ group elements. The verification cost is $O(1)$ group operations.

Proof. We prove the theorm as follows:

For security. We kindly refer to [ZGKPP17d] for a full proof of the knowledge soundness for the multivariate KZG protocol.

For efficiency. For the proving complexity, to commit the polynomial $f(Y, X)$, each prover node $\mathcal{P}_i$ need to compute com_{f_i}, which costs $O\left(\frac{N}{M}\right)$ group operations, and the master prover products them up in $O(M)$ group operations. To open the polynomial on a point (β, α), each node needs to evaluate $f_i(\alpha)$ and compute $\pi_0^{(i)}$, from which the master node derives $f(Y, \alpha)$, and π_0, with the same number of group operations as computing the commitment. Finally P_0 computes π_1 in $O(M)$ group operations. For the communication, $\mathcal{P}_i$ only sends com_{f_i}, $f_i(\alpha)$ and $\pi_0^{(i)}$ to $\mathcal{P}_0$, and receives random challenge α from $\mathcal{P}_0$, thus the communication complexity is constant. It is easy to observe that the proof size and verification time are both constant. □

4.4.2 Using DKZG to Compile Protocol 3

We show our full instantiation in Protocol 5. From this protocol, we have the following theorem for general circuits, which implies the security and efficiency of the data-parallel setting.

Theorem 4.4.2. *Given a general circuit C with N gates, Protocol 5 is a double-efficient public-coin interactive argument of knowledge protocol with witness-extended emulation for the relation of $C(\mathbb{x}; \mathbb{w}) = 1$ when splitting C into M parts $(C_0, \ldots, C_{M-1})$ each with $T = \frac{N}{M}$ gates. The total proving computation consists of $O(N \log T + M \log M)$ field operations and $O(N)$ group operations, with each $\mathcal{P}_i$ computes $O(T \log T)$ field operations and $O(T)$ group operations, while $\mathcal{P}_0$ computes $O(T \log T + M \log M)$ field operations plus $O(M + T)$ group operations. The communication is $O(1)$ per machine. The final proof size is $O(1)$. The verification cost is $O(1)$ given the access to the commitments of the public polynomials defined by the circuit in the preprocessing model.*

Proof. We prove the theorem as follows:

For security. Following the security proof in Theorem 4.3.4, by combining the knowledge soundness of DKZG in Theorem 4.4.1 and polynomial IOP in Theorem 4.3.3, we prove that Protocol 5 is a double efficient public-coin interactive argument of knowledge protocol with negligible knowledge error.

For efficiency. The complexity for the efficiency is directly implied by Theorem 4.3.4 and Theorem 4.4.1. □

Since Protocol 5 operates in the public-coin setting, it can be transformed into a SNARK protocol using the Fiat-Shamir transform.

4.5 Robust Collaborative Proving System

In the previous sections, we propose a distributed ZKP protocol that divides the proving computation across multiple machines and generates a constant-size proof, with constant communication and minimal overhead in terms of proving and verification time compared to prior work. In this section, we introduce the definition of the *Robust Collaborative Proving System (RCPS)* scheme and then propose a scheme in Protocol 5, demonstrating the potential of our protocol in a malicious environment where each prover node might sabotage the entire proof by intentionally generating a bad proof.

Definition 4.5.1 (Robust Collaborative Proving System (RCPS))**.** *Suppose for the computation C, we define a system for M participants to be Robust Collaborative Proving System (RCPS) and has the following functionalities:*

- $\mathsf{Setup}\left(1^\lambda\right) \to \mathsf{pp}$
- $\mathsf{SplitCircuit}\left(C, M\right) \to \mathcal{C} = \left(C_0, \ldots, C_{M-1}\right)$
- $\mathsf{MasterKeyGen}\left(\mathcal{C}, M, \mathsf{pp}\right) \to \left(\mathsf{mpk}, \mathsf{vk}\right)$.
- $\mathsf{KeyGen}_i\left(C_i, \mathsf{mpk}\right) \to \mathsf{pk}_i$.
- $\mathsf{SplitWitness}\left(C, \mathbb{x}\right) \to \left(\mathbb{w}_0, \ldots, \mathbb{w}_{M-1}\right)$.
- $\mathsf{CoProve}_i^{(\mathcal{P}_0)}\left(\mathsf{pk}_i, \mathbb{x}_i \in \mathbb{x}, \mathbb{w}_i\right) \to \pi_0^{(i)}$

- $\mathsf{Test}_i\left(\mathbb{x}_i, \pi_0^{(i)}, \mathsf{mpk}\right) \to b_i \in \{1, 0\}$
- $\mathsf{Merge}\left((b_0, \ldots, b_{M-1}), \left(\pi_0^{(0)}, \ldots, \pi_0^{(M-1)}\right)\right) \to \pi$.
- $\mathsf{Verify}(\mathbb{x}, \pi, \mathsf{vk}) \to \{1, 0\}$.

It satisfies the following properties: ***completeness****,* ***witness-extended emulation****,* ***partial correctness****, and* ***partial witness-extended emulation****. Because the verifier's view is identical to an interactive argument, we directly inherit the definition of the completeness, witness-extended emulation in Definition 4.2.1 and only present the definition of partial correctness and witness-extended emulation:*

- ***Partial correctness.*** *Given* pp *and the circuit partition for C, for each $0 \le i < M$, if $C_i(\mathbb{x}_i; \mathbb{w}_i) = 1$ and $\pi_0^{(i)} = \mathsf{CoProve}_i^{(\mathcal{P}_0)}(\mathsf{pk}_i, \mathbb{x}_i \in \mathbb{x}, \mathbb{w}_i)$, then*

$$\Pr\left[\mathsf{Test}_i\left(\mathbb{x}_i, \pi_0^{(i)}, \mathsf{mpk}\right) = 1\right] = 1.$$

- ***Partial witness-extended emulation.*** *With a valid* pp*, a circuit partition $\mathcal{C}$ for C, with $|\mathcal{C}| = M$, and* $\mathsf{mpk} \leftarrow \mathsf{MasterKeyGen}(\mathcal{C}, M, \mathsf{pp})$*, an RCPS has partial witness-extended emulation if that: for each $0 \le i < M$, and any PPT adversary $\mathcal{A}_i$, there exists a PPT extractor $\mathcal{E}_i$ with access to $\mathcal{A}_i$'s messages during the protocol such that $(\mathbb{x}_i, \mathsf{aux}_i) \leftarrow \mathcal{A}_i(\mathsf{mpk})$, $(\mathbb{w}_i, \mathsf{tr}_i) \leftarrow \mathcal{E}_i^{\mathcal{A}_i(\mathsf{aux}_i)}(\mathbb{x}_i, \mathsf{mpk})$, and*

$$\begin{aligned}&|\Pr[\mathcal{A}_i(\mathsf{aux}_i; \mathsf{tr}_i) = 1 \wedge \mathsf{tr} \textit{ is accepting} \Rightarrow C_i(\mathbb{x}_i, \mathbb{w}_i) = 1]\\ &- \Pr[\mathcal{A}_i(\mathsf{aux}_i; \mathsf{tr}_i) = 1]| \le \mathsf{negl}(\lambda)\end{aligned}$$

We further show that with additional verification, our protocol for data-parallel circuits is an RCPS. The full protocol is presented in Protocol 5 and after receiving the partial proofs and messages from P_i, P_0 performs the checks at the end of Step 4). In this way, the protocol is secure in the presence of malicious machines. P_0 can identify the cheating party and exclude her from the final proof.

Theorem 4.5.2. *For a data-parallel circuit C consisting of M independent sub-circuits, Protocol 5 is an RCPS with completeness, witness-extended emulation, partial correctness, and partial witness-extended emulation.*

Please refer to the proof in Appendix 4.9.

4.6 Experiments

We have implemented the fully distributed ZKP system, Pianist and we present the implementation details and evaluation results in this section.

Software and hardware. Our implementation is based on the Gnark [Gna] library written by Golang. Our scheme is implemented using 3700+ lines of code in Go. The bilinear map is instantiated using a BN254 curve. It provides around 100 bits of security and the pairing instruction is supported in Solidity, the programming language of Ethereum smart contracts. The experiments were executed on AWS m6i.16xlarge machines with 64 vCPUs and 256 GiB memory. We used the multi-threading enabled by the Gnark library. We opened 2–64 machines over the two regions of California and Oregon.

Design of the experiments. The goal of the experiments is to evaluate and demonstrate the following three advantages of Pianist:

1. **Linear scalability**: we measure the running time and memory usage and demonstrate that Pianist has linear scalability in the number of machines. The running time decreases linearly as the machine number grows. The maximum size of the circuit supported by the system grows linearly with the number of machines.

2. **Minimum communication and synchronization**: we measure the communication between the machines to demonstrate that Pianist only incurs $O(1)$ communication per machine in $O(1)$ round.

3. **Constant proof size and verifier time**: we report the proof size and the verifier time and show that they remain small in practice.

These three properties are critical for blockchain applications where with our new system, users can contribute to ZKP generations in these applications in a model similar to mining pools.

4.6.1 Evaluations of Pianist for zkRollups

We first present the performance of Pianist on data-parallel circuits in the application of zkRollups. We use the rollup circuit by Polygon Hermez [Her]. The circuit is compiled using Circom [Cira] and the output format is the rank-1-constraint-system (R1CS). As Pianist and the original Plonk do not support R1CS directly, we further compile the R1CS to Gnark's Plonk constraint. The number of R1CS constraints is about 86k per transaction and the final Plonk circuit we use in our experiments is about 660k per transaction. This transformation introduces a big overhead compared to manually designed circuits. In practice, the size of the Plonk circuit can be reduced significantly with special gates and lookup arguments. For example, Scroll [Scr] designed more than 2000 custom gates for the zkEVM circuit. Unfortunately, we could not find any open-source code of the Plonk circuit for zkRollups (even with custom gates). However, the big overhead of the transformation does not defeat the purpose of our experiments. No matter how many transactions can be supported on a single machine, we show that Pianist can scale it to M times more using M machines with small communication.

Prover time. We run our distributed proof generation on 2-64 machines and Figure 4.1 shows the result. The x-axis is the number of transactions to batch in the zkRollups and the y-axis is the prover time. We report the prover time of each $\mathcal{P}_i$ in our scheme and the running time of the original Plonk scheme on a single machine as a baseline. We introduced additional optimizations to Plonk on a single machine to improve the memory usage, and the performance shown in all of our experiments are based on the optimized version. We run each case to the maximum number of transactions until the machines run out of memory. As shown in the figure, with 64 machines, Pianist can prove up to 8192 transactions in 313s, while the original Plonk can only scale to 32 transactions with a prover time of 95s. The number of transactions and thus the maximum circuit size scales linearly in the number of machines. Moreover, given a fixed number of transactions, the prover time is accelerated by the number of machines. For example, it only takes 17.5s to prove 32 transactions using 4 machines, $5.4\times$ faster than on a single machine. In addition, the additional time on P_0 to generate the final proof is only 2-16ms in all of our experiments, which is extremely fast compared to the prover time of each machine.

Constant communication, proof size, and verifier time. In our experiments, each machine only sends 1984 bytes of messages for data-parallel circuits (or 2080 for general circuits) in 4 rounds to the master node and receives 160 bytes for data-parallel (or 256 for general) circuits from the master node, regardless of the total number of machines. Because of this, the bandwidth and the network delay of the machines do not affect the results at all. This feature enables large-scale zkRollups with the help of users globally in a model similar to a mining pool, as the nodes do not have to stay online and deal with massive communication with other nodes in a Map-Reduce framework as in [WZCPS18]. The proof size is 27 $\mathbb{G}_1$ (or 34 for general circuits) and 15 (or 20 for general circuits) $\mathbb{F}$ elements (2208 bytes or 2816 for general circuits) and the verifier time is 3.5ms in all cases regardless of the number of transactions. Compared to the original Plonk, we use bivariate polynomials, which increase the proof size by 18 (or 25 for general circuits) $\mathbb{G}_1$, 7 (or 12 for general circuits) $\mathbb{F}$ elements, and the verifier time by two pairings.

4.6.2 Evaluations on General Circuits

In this section, we further demonstrate that Pianist supports the distributed proof generation of general circuits with arbitrary connections. We vary the total size of the circuit from 2^{21} to 2^{25}, and randomly sample the type and the connection of each gate. The circuit is evaluated and the witness is distributed evenly to multiple machines. In practice, the memory usage of the circuit evaluation is not the bottleneck and the evaluation of the entire circuit can be executed on each machine individually.

Prover time. In Figure 4.2, the x-axis is the number of machines and the y-axis is the prover time of each machine P_i. As shown in the Figure, the running time is decreasing with the number of machines. In particular, for a random circuit of size 2^{25}, it takes 121s to generate the proof using Plonk on a single machine (with our optimizations), while it takes 76.9s on 2 machines in Pianist, $1.57\times$ faster than Plonk. It is further reduced to 5s using 32 machines, which is $24.2\times$ faster than Plonk.

In addition, Table 4.2 shows the additional time on P_0 to merge proofs and messages from P_is. As shown in the table, this step only takes several milliseconds in all instances.

Overhead vs. Plonk To show that the overhead of proving time between Pianist and Plonk, we illustrate the case with the same circuit size as 2^{21} per instance in Figure 4.4. From this result, we show that the overhead of Pianist is negligible.

Memory usage. Figure 4.3 shows the memory usage of the machines. As shown in the figure, in Pianist, the memory usage on each machine decreases with the number of machines. For example, for a circuit of size 2^{24}, it takes 70.7GB of memory to run the original protocol on a single machine, while it only takes 31.7GB on each machine to run Pianist using two machines. It is further improved to 1.92GB using 32 machines, which is $36.8\times$ smaller than Plonk. The improvement is critical for zkRollups and zkEVM as the memory consumption of existing ZKP systems is large. Pianist is able to increase the scalability of these schemes linearly in the number of machines, thus batching more transactions in one ZKP with the help of the fully distributed proof generations.

Communication, proof size, and verifier time. Similar to the case of data-parallel circuits, they all remain $O(1)$ for general circuits. In particular, the communication is 2336 bytes per machine, the proof size is 2816 bytes and the verifier time is 3ms.

Circuit Size	8 Nodes	16 Nodes	32 Nodes
2^{21}	2.764 ms	3.576ms	4.629ms
2^{22}	2.975 ms	3.666ms	4.800ms
2^{23}	3.073 ms	3.687ms	5.009ms
2^{24}	3.120 ms	3.692ms	5.705ms

Table 4.2: Extra time to merge proofs on $\mathcal{P}_0$.

4.7 Discussions

Comparison with PCD and IVC As highlighted in Section 4.1.1, *Proof-Carrying Data* (PCD)[CT10; BCCT13] generates a proof at each step for newly received transactions. To ensure the correctness of previous proofs, there are two approaches: either aggregating a recursive proof for the previous succinct verification circuits in the current proof[BCCT13; BSCTV14b; COS20], or merging each proof into an accumulator and verifying them all at once [BCMS20; Hal; BCLMS20; KST22; KS22]. By using either the aggregated verification circuit or the accumulator, the prover does not need to store the entire circuit, making these methods suitable alternatives when memory is limited.

Nonetheless, these techniques share some common drawbacks. First, they all utilize recursive proofs, which depend on the assumption that the *random oracle* (RO) used in the Fiat-Shamir transform can be efficiently instantiated. Second, their proofs are generated sequentially, potentially imposing an upper bound on TPS. Third, either the succinct verification aggregation or the accumulator demands extra effort from the prover, which will at least increase linearly as the number of steps grows, resulting in reduced practical performance. Our work, on the other hand, avoids these issues since we do not employ recursive proofs.

We also mentioned *Incremental Verifiable Computation* (IVC) for incrementally verifying stages of long-run computation, such as Nova [KST22] and SuperNova [KS22]. While Nova and SuperNova are suitable for real-world applications like zkRollups and zkEVM, they expose the output of each stage, which cannot guarantee a zero-knowledge property throughout the entire process. We argue that our work not only supports general circuits, offering a more powerful computation model but can also achieve zero-knowledge properties using common techniques.

Comparison with aPlonk [ABST22] In Section 4.1.1, we mentioned an alternative solution, aPlonk, which is based on Plonk and generalized IPA under the same settings (distributed, shared Fiat-Shamir randomness) as our approach. However, due to the use of IPA, their final verification cost is logarithmic with respect to the number of parties. Additionally, they only propose a solution for data-parallel circuits and their solution involves recursive proofs. In contrast, our proof does not require recursive circuits, our verification cost is independent of the number of parties, and our approach is more flexible when generalizing to circuits with connections and general circuits. Consequently, our solution delivers better performance, both theoretically and practically.

Custom gates. A key advantage of the Plonk scheme lies in its support for custom gates. Users can define their own gate constraints, differing from Equation 4.1, by altering term forms and introducing rotations. Custom gates may increase the degree and total number of terms in Equation 4.1, but typically reduce the overall gate count in the circuit, leading to significant improvements in prover time in practice. As mentioned earlier, Scroll [Scr] designed over 2000 custom gates to enhance Plonk's performance in their zkEVM implementation. Our new schemes are fully compatible with custom gates by following the outline introduced in Section 4.3.

Additionally, as mentioned in Section 4.3, rotations can be introduced for the variable Y, and simple, regular connections among different sub-circuits can be established based on the data-parallel setting. For instance, in a zkEVM context, if we treat a block of instructions as a sub-circuit, we can define the constraint $S_{\mathsf{pc}}(X)f_{\mathsf{pc}}(Y,X)+\Delta = S_{\mathsf{pc}}(X)f_{\mathsf{pc}}(\omega_Y Y, X)$ to represent the program counter change between the previous and current blocks, where $S_{\mathsf{pc}}(X)$ serves as a selector to indicate the row recording the program counter.

Lookup arguments. Lookup arguments play a crucial role in the implementation of zkRollups and zkEVM, as they help construct proofs for RAM and chiplet computations. Since these lookup arguments are compiled into polynomial equations, we assert that they are compatible with our system. We can identify two primary use cases for lookup arguments:

1. **Lookup arguments with local tables**. In this scenario, each sub-circuit possesses its own lookup arguments, independent of other sub-circuits. This setup is well-suited for situations where each machine runs a program with its local memory, for example.

2. **Lookup arguments with global tables**. This configuration allows applications to define global lookup tables, such as range tables or chiplet computing. We argue that, by leveraging the latest lookup argument research [Hab22; EFG22] based on logarithmic derivatives, this can be easily implemented. These protocols eliminate the need for cumbersome permutations of input and table vectors, requiring only the counting of occurrences and the execution of a sumcheck protocol. Furthermore, by employing rotation on the variable Y, global tables can be distributed across different machines, thus reducing the workload for the master node.

Protocol 5 (**Distributedly Computable Double-efficient Public-coin Interactive Argument of Knowledge**). $\mathcal{P}$ is a prover with M machines of $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$, with master node $\mathcal{P}_0$. Given a fan-in two arithmetic circuit C of size N with M sub-circuits, each of size $T = N/M$. $\mathcal{P}$ wants to convince $\mathcal{V}$ that $C_i\left(\mathbb{x}^{(i)}; \mathbb{w}^{(i)}\right) = 1$ for all $i \in [M]$, where $\mathbb{x}^{(i)}$ is the public input and $\mathbb{w}^{(i)}$ is the witness of C_i. Each $\mathcal{P}_i$ holds the sub-circuit C_i.

- **Key generation and preprocessing procedure:** Let $(\mathsf{pk} = \{\mathsf{pk}_i\}_{0\leq i<M}, \mathsf{vk})$. Run $\mathsf{DKZG.KeyGen}(1^\lambda, M, T)$ and generate $g,\ g^{\tau_X},\ g^{\tau_Y},\ \mathbf{U} = (\mathbf{U_i})_{0\leq i<M} = \left(g^{R_i(\tau_Y)L_j(\tau_X)}\right)_{\substack{0\leq i<M\\0\leq j<T}}$, and derive $\mathbf{V} = \left(g^{R_i(\tau_Y)}\right)_{0\leq i<M}$. Compute commitments com_S for each of the following polynomial set $\mathcal{S}_{\mathsf{pp}}$: $\mathcal{S}_{\mathsf{pp}} = \{Q_a(Y,X), Q_b(Y,X), Q_o(Y,X), Q_{ab}(Y,X), Q_c(Y,X), \sigma_{Y,a}(Y,X), \sigma_{Y,b}(Y,X), \sigma_{Y,o}(Y,X), \sigma_{X,a}(Y,X), \sigma_{X,b}(Y,X), \sigma_{X,o}(Y,X)\}$ Let $s_i(X)$ be defined by $S(Y,X) = \sum_{i=0}^{M-1} R_i(Y)s_i(X)$, we define $\mathsf{pk}_0 = \left(\left(\mathbf{V}, \mathbf{U}_0, \{s_0(X)\}_{S\in\mathcal{S}_{\mathsf{pp}}}\right)\right)$, $\mathsf{pk}_i = \left(\mathbf{U}_i, \{s_i(X)\}_{S\in\mathcal{S}_{\mathsf{pp}}}\right)$, $\mathsf{vk} = \left(g^{\tau_X}, g^{\tau_Y}, \{\mathsf{com}_S\}_{S\in\mathcal{S}_{\mathsf{pp}}}\right)$

- **Proving procedure:**

  1. Each $\mathcal{P}_i$ evaluates C_i and defines polynomials $a_i(X)$, $b_i(X)$, $o_i(X)$. $\mathcal{P}$ invokes the distributed algorithm Commit in Protocol 4 to obtain $\mathsf{com}_A, \mathsf{com}_B, \mathsf{com}_O$ as commitments of $A(Y,X), B(Y,X), O(Y,X)$ and sends them to $\mathcal{V}$.
  2. After receiving random points η_Y, η_X and γ from $\mathcal{V}$, each $\mathcal{P}_i$ computes $z_i(X)$, and sends the last entry $z_{i,T-1}$ to the $\mathcal{P}_0$, from which $\mathcal{P}_0$ computes $W(Y) = \sum_{i=0}^{M-1} w_i R_i(Y)$. Then, $\mathcal{P}$ invokes the Commit algorithm in Protocol 4 to obtain $\mathsf{com}_Z, \mathsf{com}_W$ and sends them to $\mathcal{V}$.
  3. After receiving λ from $\mathcal{V}$, $\mathcal{P}_0$ shares it to $\mathcal{P}_i$. Each $\mathcal{P}_i$ computes $h_i(X)$ according to Equation 4.6 or Equation 4.18. Then, $\mathcal{P}$ invokes algorithm Commit in Protocol 4 to obtain $\mathsf{com}_{H_X} = \left\{\mathsf{com}_{H_{X,0}}, \mathsf{com}_{H_{X,1}}, \mathsf{com}_{H_{X,2}}, \mathsf{com}_{H_{X,3}}\right\}$ as commitments of $H_X(Y,X) = \sum_{i=0}^{M-1} R_i(Y)h_i(X)$, and sends them to $\mathcal{V}$. The form of com_{H_X} due to the fact that $H_X(Y,X)$ has degree $3T-2$ or $4T-2$ with respect to X.
  4. After receiving the random point α from $\mathcal{V}$, $\mathcal{P}_0$ sends α to each $\mathcal{P}_i$. We define $\mathcal{S}_{\mathsf{wit}} = \{A, B, O, Z\}$ Then $\mathcal{P}$ process the following computation:
     - for each polynomial $S \in \mathcal{S}_{\mathsf{pp}} \cup \mathcal{S}_{\mathsf{wit}}$, run Step 1 & 2 in $\mathsf{DKZG.Open}$ function, computing $S(Y,\alpha)$ and the first entry of π_S as $\pi_S[0]$.
     - Furthermore, $\mathcal{P}_0$ also recovers $Z(Y, \omega_X\cdot\alpha)$ and $\pi'_Z[0]$ when running $\mathsf{DKZG.Open}$ for opening $Z(Y,X)$ on $X = \omega_X\cdot\alpha$.
     - With all $\{S(Y,\alpha)\}_{S\in\mathcal{S}_{\mathsf{wit}}\cup\mathcal{S}_{\mathsf{pp}}}$ and $Z(Y,\omega_X\alpha)$, $\mathcal{P}$ computes $H_{Y,\alpha}(Y)$ according to Equation 4.12 or Equation 4.23 and the univariate commitment $\mathsf{com}_{H_Y} = \left\{\mathsf{com}_{H_{Y,0}}, \mathsf{com}_{H_{Y,1}}, \mathsf{com}_{H_{Y,2}}, \mathsf{com}_{H_{Y,3}}\right\}$ as commitments of $H_Y(Y,X) = \sum_{i=0}^{M-1} h_{y,\alpha,i}R_i(Y)$, and sends them to $\mathcal{V}$. The degree of $H_{Y,\alpha}(Y)$ has degree $3T-2$ or $4T-2$ with respect to Y.
     - Finally, $\mathcal{P}$ sends $\{\pi_S[0]\}_{S\in\mathcal{S}_{\mathsf{wit}}\cup\mathcal{S}_{\mathsf{pp}}} \cup \left\{\pi'_Z[0], \mathsf{com}_{h_Y}\right\}$ to $\mathcal{V}$.

     In the malicious setting: $\mathcal{P}_i$ for $i \neq 0$ might be dishonest, $\mathcal{P}_0$ needs to check whether the other nodes have sent malicious proof. Since $\mathcal{P}_0$ has received the commitments $\left(\mathsf{com}_{s_i}\right)_{0\leq i<M}$ for each $s \in \mathcal{S}_{\mathsf{wit}} \cup \left\{h_{X,0}, h_{X,1}, h_{X,2}, h_{X,3}\right\}$ and $\{w_i\}_{0\leq i<M}$ when distributively committing polynomials, and the corresponding evaluations $\{s_i(\alpha)\}$ and opening proof $\left\{\pi_{0,s}^{(i)}\right\}$ when distributively opening the polynomials with α, $\mathcal{P}_0$ can verify Equation 4.1, Equation 4.14, Equation 4.15, Equation 4.16, Equation 4.17 with evaluations and the following paring check: $e(\mathsf{com}_{s_i}/g^{s_i(\alpha)}, g) \overset{?}{=} e(\pi_{0,s}^{(i)}, g^{\tau_X-\alpha})$ **If for some i^*, the check doesn't pass, $\mathcal{P}_0$ discards all messages sent from $\mathcal{P}_{i^*}$ and replaces them with dummy messages.**
  5. After receiving β from $\mathcal{V}$, $\mathcal{P}$ executes Step 3 in the Open algorithm in Protocol 4 to compute the evaluations on $S(\beta,\alpha)$ and $\pi_S[1]$ for $S \in \mathcal{S}_{\mathsf{wit}} \cup \mathcal{S}_{\mathsf{pp}}$ and $Z(\beta, \omega_X\alpha)$, $H_X(\beta,\alpha)$. $\mathcal{P}$ also computes $H_Y(\beta), \pi_{H_Y}$ and $W(\beta), \pi_W$. In the end, $\mathcal{P}$ sends all the evaluations and proofs to $\mathcal{V}$.

- **Verification procedure:** $\mathcal{V}$ verifies the following steps:

  1. $\mathcal{V}$ verifies the evaluation and proof $S(\beta,\alpha), \pi_S$ for $S \in \mathcal{S}_{\mathsf{wit}} \cup \mathcal{S}_{\mathsf{pp}} \cup \{Z(Y,\omega_X X), H_X(Y,X)\}$, together with $H_Y(\beta)$, $W(\beta)$ with corresponding proofs by invoking the Verify algorithm in Protocol 4.
  2. With the evaluations, $\mathcal{V}$ verifies the gate constraint Equation 4.8.
  3. With the evaluations, $\mathcal{V}$ verifies the copy constraints according to Equation 4.9, Equation 4.10, or Equation 4.19, Equation 4.20, Equation 4.21, Equation 4.22.

  If all checks pass, $\mathcal{V}$ outputs 1; otherwise $\mathcal{V}$ outputs 0.

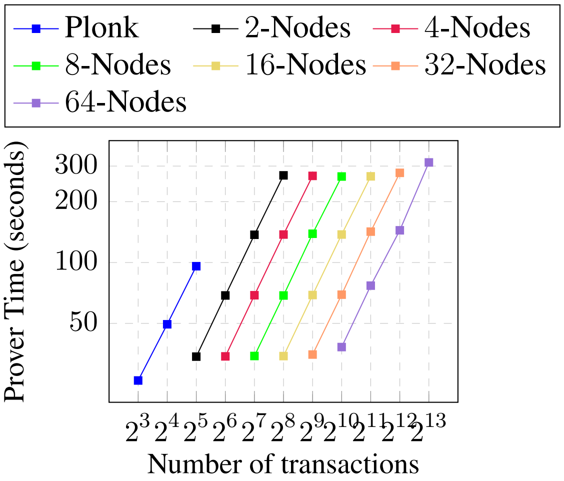


Figure 4.1: Prover time of Pianist for zkRollups transaction verification.

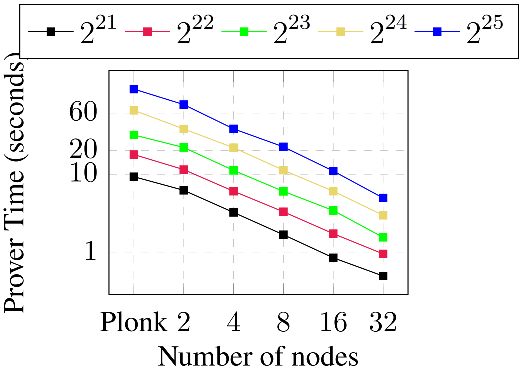


Figure 4.2: Prover time of random circuits

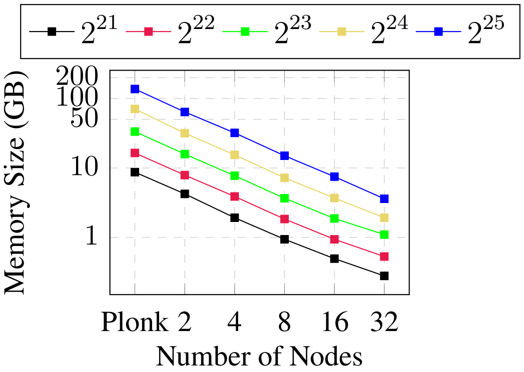


Figure 4.3: Memory consumption of random circuit

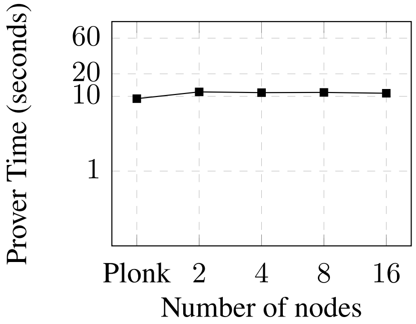


Figure 4.4: Comparison between the prover time of a single node in Pianist and Plonk for sub-circuit with the same size

4.8 Proof of Theorem 4.3.3

Theorem 4.3.3. Protocol 3 is a Polynomial IOP protocol with negligible knowledge error.

Proof. For simplicity, we only prove the theorem in the general-circuit setting. We assume the adversary $\mathcal{P}^*$ has unbounded power. At the beginning, both $\mathcal{P}^*$ and $\mathcal{V}$ hold the following precomputed polynomial set $\mathcal{S}_{\mathsf{pp}}$:

1. $\{Q_a(Y,X), Q_b(Y,X), Q_o(Y,X), Q_{ab}(Y,X), Q_c(Y,X)\}$
2. $\{\sigma_{Y,a}(Y,X), \sigma_{Y,b}(Y,X), \sigma_{Y,o}(Y,X)\ \sigma_{X,a}(Y,X), \sigma_{X,b}(Y,X), \sigma_{X,o}(Y,X)\}$

With arbitrary invalid witness $\mathbf{a}^*, \mathbf{b}^*, \mathbf{o}^* \in \mathbb{F}^{M\times T}$ generated by $\mathcal{P}^*$, the possibility that $\mathcal{V}$ outputs $\mathtt{1}$ is negiligible.

1. For each $\mathbf{f} \in \{\mathbf{a}^*, \mathbf{b}^*, \mathbf{o}^*\}$, $\mathcal{P}^*$ define the polynomial $F(Y,X) = \sum_{i=0}^{M-1}\sum_{j=0}^{T-1} f_{i,j}L_j(X)R_i(Y) = \sum_{i=0}^{M-1} f_{i,j}L_j(X)$, $Z^*(Y,X)$ and $W^*(Y)$ are derived from the witnesses.
2. In the round after $\mathcal{V}$ sends β to $\mathcal{P}^*$, which is the last round, $\mathcal{V}$ queries all oracles in $\mathcal{S}_{\mathsf{pp}}$ and $\mathcal{S}_{\mathsf{wit}} = \{A^*(Y,X), B^*(Y,X), O^*(Y,X), Z^*(Y,X), W^*(Y)\}$ with random challenge α and β and verifies the Equation 4.23. Since $\mathcal{V}$ has received all oracles before sending β to $\mathcal{P}$, by the Schwartz-Zipple Lemma [Sch80; Zip79], it implies that there exists $Q_0(Y) := G_\alpha(Y) + \sum_{i=0}^{3}\lambda^{i+1}P_{i,\alpha}(Y) - \left(\alpha^T - 1\right)H_X(Y,\alpha)$ where

$$
\begin{aligned}
G_\alpha(Y) &:= Q_a(Y,\alpha)A^*(Y,\alpha) + Q_b(Y,\alpha)B^*(Y,\alpha)\\
&\quad + Q_{ab}(Y,\alpha)A^*(Y,\alpha)B^*(Y,\alpha)\\
&\quad + Q_o(Y,\alpha)O^*(Y,\alpha) + Q_c(Y,\alpha)\\
P_{0,\alpha}(Y) &:= L_0(\alpha)(Z^*(Y,\alpha) - 1)\\
P_{1,\alpha}(Y) &:= (1 - L_{N-1}(\alpha))\\
&\quad \cdot \left(Z^*(Y,\alpha)F_\alpha(Y) - Z^*_{\alpha,\mathtt{nxt}}(Y)F'_\alpha(Y)\right)\\
P_{2,\alpha}(Y) &:= R_0(Y)(W^*(Y) - 1)\\
P_{3,\alpha}(Y) &:= L_{N-1}(\alpha)\left(W^*(Y)Z^*(Y,\alpha)F_\alpha(Y)\right.\\
&\qquad\qquad \left. - W^*(\omega_Y Y)F'_\alpha(Y)\right)\\
F_\alpha(Y) &:= \prod_{S\in\{A^*,B^*,O^*\}} (S(Y,\alpha) + \eta_Y\sigma_{Y,s}(Y,\alpha)\\
&\qquad\qquad + \eta_X\sigma_{X,s}(Y,\alpha) + \gamma)\\
F'_\alpha(Y) &:= \prod_{\substack{S\in\\ \{A^*,B^*,O^*\}}} (S(Y,\alpha) + \eta_Y Y + \eta_X k_s\alpha + \gamma)
\end{aligned}
\tag{4.24}
$$

such that

$$Q_0(Y) = \left(Y^M - 1\right)H_{Y,\alpha}(Y) \tag{4.25}$$

which means, for $0 \le i < M$, $Q(\omega_Y^i) = 0$. From the form of $F \in \{A^*, B^*, O^*, Z^*, H_X\}$, $F(Y,X) = \sum_{i=0}^{M-1} f_i(X)R_i(Y)$ and $W^*(Y) = \sum_{i=0}^{M-1} w_i^* R_i(Y)$, we know that for $0 \le i < M$

after assigning the value $a_i^*(\alpha), b_i^*(\alpha), o_i^*(\alpha), z_i^*(\alpha), z_i^*(\omega_X\alpha)$ and w_i to the corresponding polynomials of Y in Equation 4.24, we will derive $q_{0,i} = Q_0(\omega_Y^i) = 0$.

In the following rounds, we go through the proof for $0 \leq i < M$**.**

3. In the round after $\mathcal{V}$ sends α to $\mathcal{P}$. Since $\mathcal{V}^*$ has oracles for all polynomials, before sending α to $\mathcal{P}^*$, which is equivalent to have oracles of $s_i(X) \in (a_i^*(X), b_i^*(X), c_i^*(X), z_i^*(X), h_{X,i}(X))$ since $s_i(r) = S(\omega_Y^i, r)$. In addition, $\mathcal{V}$ has the oracle $W(Y)$ from which he can query $w_i = W(\omega_Y^i)$. Again from the Schwartz-Zipple Lemma, there exists $q_i(X) := g_i(X) + \lambda p_{i,0}(X) + \lambda^2 p_{i,1}(X) + \lambda^4 p_{i,3}(X)$, where
$$\begin{aligned} g_i(X) &:= q_{a,i}(X)a_i^*(X) + q_{b,i}(X)b_i^*(X) + q_{c,i}(X) \\ &\quad + q_{ab,i}(X)a_i^*(X)b_i^*(X) + q_{o,i}(X)o_i^*(X) \\ p_{0,i}(X) &:= L_0(X)(z_i^*(X) - 1) \\ p_{1,i}(X) &:= (1 - L_{N-1}(X)) \\ &\quad \cdot \Big(z_i^*(X)f_i(X) - z_i^*(\omega_X X)f_i'(X)\Big) \\ p_{3,i}(X) &:= L_{N-1}(X)\Big(w_i z_i^*(X)f_i(X) - w_{i+1}f_i'(X) \\ f_i(X) &:= \prod_{s\in\{a^*,b^*,o^*\}} (s_i(X) + \eta_Y\omega_Y^i \\ &\qquad + \eta_X\sigma_{X,s,i}(X) + \gamma) \\ f_i'(X) &:= \prod_{\substack{s\in \\ \{a^*,b^*,o^*\}}} (s_i(X) + \eta_Y\omega_Y^i + \eta_X k_s X + \gamma) \end{aligned} \tag{4.26}$$
such that
$$q_i(X) = \Big(X^T - 1\Big)h_{X,i}(X) \tag{4.27}$$
which means, for $0 \leq i < M$, $Q(\omega_Y^i) = 0$. From the form of $f \in \{a^*, b^*, o^*, z^*, h_X\}$, $f_i(Y, X) = \sum_{j=0}^{T-1} f_{i,j}L_j(X)$, we know that for $0 \leq j < T$ after assigning the value $a_{i,j}^*, b_{i,j}^*, o_{i,j}^*, z_{i,j}^*, z_{i,j+1}^*$ to the corresponding polynomials of X in Equation 4.26, we will derive $q_{i,j} = q_i(\omega_X^j) = 0$.

4. In the round after $\mathcal{V}$ sends γ to $\mathcal{P}^*$, since $\mathcal{V}^*$ has received oracles $(a_i^*(X), b_i^*(X), c_i^*(X), z_i^*(X), w_i^*, w_{i+1}^*)$ before sending λ to $\mathcal{P}$, from Schwartz-Zipple Lemma, it implies for each $0 \leq j < T$, $g_i(\omega_X^j) = p_{0,i}(\omega_X^j) = p_{1,i}(\omega_X^j) = p_{2,i}(\omega_X^j) = 0$.

After combine the claims for $0 \leq i < M$, it implies $(\mathbf{a}^*, \mathbf{b}^*, \mathbf{c}^*) \in (\mathbb{F}^{M\times T}, \mathbb{F}^{M\times T}, \mathbb{F}^{M\times T})$ is a valid witness for gate constraints. From the constraints related to Z and W, suppose $(\sigma_{Y,i,j}, \sigma_{X,i,j})$ correctly describe the permutation cycles similar to the permutation cycles in Plonk, we prove the argument that
$$\left\{\begin{matrix} (a_{i,j}, \sigma_{Y,a,i,j}, \sigma_{X,a,i,j}) \\ (b_{i,j}, \sigma_{Y,b,i,j}, \sigma_{X,b,i,j}) \\ (o_{i,j}, \sigma_{Y,o,i,j}, \sigma_{X,o,i,j}) \end{matrix}\right\} = \left\{\begin{matrix} (a_{i,j}, \omega_Y^i, k_a\omega_X^j) \\ (b_{i,j}, \omega_Y^i, k_b\omega_X^j) \\ (o_{i,j}, \omega_Y^i, k_o\omega_X^j) \end{matrix}\right\}$$
Therefore, $(\mathbf{a}^*, \mathbf{b}^*, \mathbf{c}^*) \in (\mathbb{F}^{M\times T}, \mathbb{F}^{M\times T}, \mathbb{F}^{M\times T})$ also satisfies the copy constraints which is equivalent to the copy constraints in original Plonk. The possibility $\mathcal{P}^*$ successfully cheats $\mathcal{V}$ is bounded by $\frac{5M+5T+O(1)}{|\mathbb{F}|}$.

□

4.9 Proof of Theorem 4.5.2

Theorem 4.5.2. For a data-parallel circuit C consisting of M independent sub-circuits, Protocol 5 is an RCPS with completeness, witness-extended emulation, partial correctness, and partial witness-extended emulation.

Proof. We prove this theorem by first utilizing Protocol 5 in a data-parallel setting to implement RCPS and subsequently demonstrating that the instantiation possesses the witness-extended emulation property.

Implementation. We implement the functionalities in Definition 4.5.1 as follows:

- $\mathsf{Setup}\left(1^\lambda\right) \to \mathsf{pp}$: run $\mathsf{DKZG.KeyGen}(1^\lambda, M_{\mathsf{max}}, T_{\mathsf{max}})$ and generate all bases needed for creating and verifying polynomial commitments.
- $\mathsf{SplitCircuit}\left(C, M\right) \to \mathcal{C} = (C_0, \ldots, C_{M-1})$: split C into M independent sub-circuits, denoted as $C_0, \ldots, C_{M-1}$.
- $\mathsf{MasterKeyGen}\left(\mathcal{C}, M, \mathsf{pp}\right) \to (\mathsf{mpk}, \mathsf{vk})$: call $\mathsf{DKZG.Commit}(S, \mathsf{pp})$ to compute $\left\{\left(\mathsf{com}_S, \mathsf{com}_{s_i}\right)\right\}_{s\in\mathcal{S}_{\mathsf{pp}}}$. We define $\mathsf{mpk} = \left(\mathsf{com}_{s_i}\right)_{\substack{s\in\mathcal{S}_{\mathsf{pp}}\\ 0\le i<M}, \mathsf{pp}}$. Set vk as in the **key generation and preprocessing procedure** from Protocol 5.
- $\mathsf{KeyGen}_i\left(C_i, \mathsf{mpk}\right) \to \mathsf{pk}_i$: generate pk_i for $\mathcal{P}_i$ following the **key generation and preprocessing procedure** in Protocol 5.
- $\mathsf{SplitWitness}\left(C, \mathbb{x}\right) \to (\mathbb{w}_0, \ldots, \mathbb{w}_{M-1})$: each prover $\mathcal{P}_i$ holds the witness $\mathbb{w}_i$ in C_i.
- $\mathsf{CoProve}_i^{(\mathcal{P}_0)}\left(\mathsf{pk}_i, \mathbb{x}_i \in \mathbb{x}, \mathbb{w}_i\right) \to \pi_0^{(i)}$: execute the **proving procedure** between $\mathcal{P}_0$ and other nodes in Protocol 5 from Step 1 to Step 4, collecting all messages from $\mathcal{P}_i$ as $\pi_0^{(i)}$.
- $\mathsf{Test}_i\left(\mathbb{x}_i, \pi_0^{(i)}, \mathsf{mpk}\right) \to b_i \in \{1, 0\}$: $\mathcal{P}_0$ performs the checking procedure, as shown in the **malicious setting** in Protocol 5. If all checks pass, output 1; otherwise, output 0.
- $\mathsf{Merge}\left((b_0, \ldots, b_{M-1}), \left(\pi_0^{(0)}, \ldots, \pi_0^{(M-1)}\right)\right) \to \pi$: $\mathcal{P}_0$ replaces $\pi_0^{(i)}$ with a dummy proof if $b_i = 0$, and run Step 5 in Protocol 5 to generate the proof π.
- $\mathsf{Verify}\left(\mathbb{x}, \pi, \mathsf{vk}\right) \to \{1, 0\}$: $\mathcal{V}$ runs the **verification procedure** in Protocol 5.

Completeness and witness-extended emulation properties are inherited from Protocol 5 as proven in Theorem 4.4.2. Partial correctness is straightforward. Consequently, we only need to prove that the given implementation possesses witness-extended emulation. We note that the underlying polynomial IOP is derived from the constraints in Equation 4.1, Equation 4.4, and Equation 4.5, compiled with a variant of KZG protocol. As a result, we follow the same framework used in the proof for Theorem 4.3.4.

The knowledge error in the IOP. By replacing $\mathcal{V}$ with $\mathcal{P}_0$, we claim that the second part of the proof in Appendix 4.8, which argues for $0 \le i < M$, provides evidence of the knowledge error in this implementation.

The knowledge soundness of PCS. We observe that the PCS scheme between $\mathcal{P}_i$ and $\mathcal{P}_0$ is a variant of the KZG protocol, with $\mathsf{pp} = \left(g^{R_i(\tau_Y)L_j(\tau_X)}\right) 0 \le j < T$ instead of $\left(g^{L_i(\tau_X)}\right)$ for the standard KZG protocol with pp in the Lagrange basis. To prove the knowledge soundness of the KZG variant, we proceed as follows:

assuming $\mathsf{pp} = \left(g^{L_j(\tau_X)}\right) 0 \leq j < T$, the adversary $\mathcal{A}$ for the KZG protocol generates $z^*, \mathbf{x}^*, \mathsf{com}^*, \pi^*$. Then, $\mathcal{A}$ creates $\mathsf{pp}' = \left(g^{R_i(\tau_Y)L_J(\tau_X)}\right)$ for the KZG variant using a randomly sampled τ_Y. If an extractor $\mathcal{E}^{\mathcal{A}'(\cdot)}(1^\lambda, \mathsf{pp}')$ can successfully compute f^* such that $f^*(\mathbf{x}^*) = z^*$, we can construct $\mathcal{E}^{\mathcal{A}(\cdot)}(1^\lambda, \mathsf{pp})$ by invoking $\mathcal{E}'$ and returning f^*. Therefore, we proved that the KZG variant also has knowledge soundness. □

Chapter 5

zkBridge: Trustless Cross-chain Bridges Made Practical

Blockchains have seen growing traction with cryptocurrencies reaching a market cap of over 1 trillion dollars, major institution investors taking interests, and global impacts on governments, businesses, and individuals. Also growing significantly is the heterogeneity of the ecosystem where a variety of blockchains co-exist. Cross-chain bridge is a necessary building block in this multi-chain ecosystem. Existing solutions, however, either suffer from performance issues or rely on trust assumptions of committees that significantly lower the security. Recurring attacks against bridges have cost users more than 1.5 billion USD. In this paper, we introduce zkBridge, an efficient cross-chain bridge that guarantees strong security without external trust assumptions. With succinct proofs, zkBridge not only guarantees correctness, but also significantly reduces on-chain verification cost. We propose novel succinct proof protocols that are orders-of-magnitude faster than existing solutions for workload in zkBridge. With a modular design, zkBridge enables a broad spectrum of use cases and capabilities, including message passing, token transferring, and other computational logic operating on state changes from different chains. To demonstrate the practicality of zkBridge, we implemented a prototype bridge from Cosmos to Ethereum, a particularly challenging direction that involves large proof circuits that existing systems cannot efficiently handle. Our evaluation shows that zkBridge achieves practical performance: proof generation takes less than 20 seconds, while verifying proofs on-chain costs less than 230K gas. For completeness, we also implemented and evaluated the direction from Ethereum to other EVM-compatible chains (such as BSC) which involve smaller circuits and incurs much less overhead.

This work was previously published in [Zha+21a].

5.1 Introduction

Since the debut of Bitcoin, blockchains have evolved to an expansive ecosystem of various applications and communities. Cryptocurrencies like Bitcoin and Ethereum are gaining rapid traction with the market cap reaching over a trillion USD [Coi] and institutional investors [Ham22; Win21] taking interests. Decentralized Finance (DeFi) demonstrates that blockchains can enable finance instruments that are otherwise impossible (e.g., flash loans [QZLG21]). More recently, digital artists [Bee] and content creators [You] resort to blockchains for transparent and accountable circulation of their works.

Also growing significantly is the heterogeneity of the ecosystem. A wide range of blockchains have been proposed and deployed, ranging from ones leveraging computation (e.g., in Proof-of-Work [Nak08]), to economic incentives (e.g., in Proof-of-Stake [GHMVZ17; BLMR14; KRDO17; DGKR17; BPS16]), and various other resources such as storage [RD16; Fil; DFKP15; ABFG14], and even time [Int]. While it is rather unclear that one blockchain dominates others in all aspects, these protocols employ different techniques and achieve different security guarantees and performance. It has thus been envisioned that (e.g., in [Amu; Mul; But]) the ecosystem will grow to a *multi-chain* future where various protocols co-exist, and developers and users can choose the best blockchain based on their preferences, the cost, and the offered amenities.

A central challenge in the multi-chain universe is how to enable secure *cross-chain bridges* through which applications on different blockchains can communicate. An ecosystem with efficient and inexpensive bridges will enable assets held on one chain to effortlessly participate in marketplaces hosted on other chains. In effect, an efficient system of bridges will do for blockchains what the Internet did for siloed communication networks.

The core functionality of a bridge between blockchains $\mathcal{C}_1$ and $\mathcal{C}_2$ is to prove to applications on $\mathcal{C}_2$ that a certain event took place on $\mathcal{C}_1$, and vice versa. We use a generic notion of a bridge, namely one that can perform multiple functions: message passing, asset transfers, etc. In our modular design, the bridge itself neither involves nor is restricted to any application-specific logic.

The problem. While cross-chain bridges have been built in practice [Rai; Polb; Lay; Axe], existing solutions either suffer from poor performance, or rely on central parties.

The operation of the bridge depends on the consensus protocols of both chains. If $\mathcal{C}_1$ runs Proof-of-Work, a natural idea is to use a light client protocol (e.g., SPV [Nak08]). Specifically, a smart contract on $\mathcal{C}_2$, denoted by $\mathcal{SC}_2$, will keep track of block headers of $\mathcal{C}_1$, based on which transaction inclusion (and other events) can be verified with Merkle proofs. This approach, however, incurs a significant computation and storage overhead, since $\mathcal{SC}_2$ needs to verify all block headers and keep a long and ever-growing list of them. For non-PoW chains, the verification can be even more expensive. For example, for a bridge between a Proof-of-Stake chain (like Cosmos) and Ethereum, verifying a single block header on Ethereum would cost about 64 million gas [Nea] (about \$6300 at time of writing), which is prohibitively high.

Currently, as an efficient alternative, many bridge protocols (PolyNetwork, Wormhole, Ronin, etc.) resort to a committee-based approach: a committee of validators are entrusted to sign off on state transfers. In these systems, the security boils down to, e.g., the honest majority assumption. This is problematic for two reasons. First, the extra trust assumption in the committee means the bridged asset is not as secure as native ones, complicating the security analysis of downstream applications. Second, relying on a small committee can lead to single point failures. Indeed, in a recent exploit of the Ronin bridge [Ron], the attackers were able to obtain five of the nine validator keys, through which they stole 624 million USD, making it the largest attack in the history of DeFi by Apr 2022[1]. Even the second and third largest attacks are also against bridges (\$611m

[1] see the ranking at `https://rekt.news/leaderboard`

was stolen from PolyNetwork [Pola] and $326m was stolen from Wormhole [Wora]), and key compromise was suspected in the PolyNetwork attack.

Our approach. We present zkBridge to enable an efficient cross-chain bridge without trusting a centralized committee. The main idea is to leverage zk-SNARK, which are succinct non-interactive proofs (arguments) of knowledge [WTSTW18; XZZPS19b; Zha+21a; BSBHR19; BSCTV14c; AHIV17; BSCRSVW19; COS20; CHMMVW20; ZGKPP17d; ZGKPP18; BBBPWM18; GWC19b; Set20]. A zk-SNARK enables a prover to efficiently convince $\mathcal{SC}_2$ that a certain state transition took place on $\mathcal{C}_1$. To do so, $\mathcal{SC}_2$ will keep track of a digest D of the latest tip of $\mathcal{C}_1$. To sync $\mathcal{SC}_2$ with new blocks in $\mathcal{C}_1$, anyone can generate and submit a zk-SNARK that proves to $\mathcal{SC}_2$ that the tip of $\mathcal{C}_1$ has advanced from D to D'.

This design offers three benefits. First, the soundness property of a zk-SNARK ensures the security of the bridge. Thus, we do not need additional security requirements beyond the security of the underlying blockchains. In particular zkBridge does not rely on a committee for security. Second, with a purpose-built zk-SNARK, $\mathcal{C}_2$ can verify a state transition of $\mathcal{C}_1$ far more efficiently than encoding the consensus logic of $\mathcal{C}_1$ in $\mathcal{SC}_2$. In this way, as an example for zkBridge from Cosmos to Ethereum, we reduce the proof verification cost from $\sim 80M$ gas to less than $230K$ gas on $\mathcal{C}_2$. The storage overhead of the bridge is reduced to constant. Third, by separating the bridge from application-specific logic, zkBridge makes it easy to enable additional applications on top of the bridge.

Technical challenges. To prove correctness of a given computation outcome using a zk-SNARK, one first needs to express the computation as an arithmetic circuit. While zk-SNARK verification is fast (logarithmic in the size of the circuit or even constant), proof generation time is at least linear, and in practice can be prohibitively expensive. Moreover, components used by real-world blockchains are not easily expressed as an arithmetic circuit. For example, the widely used EdDSA digital signature scheme is very efficient to verify on a CPU, but is expensive to express as an arithmetic circuit, requiring more than 2 million gates [Cirb]. In a cross-chain bridge, each state transition could require the verification of hundreds of signatures depending on the chains, making it prohibitively expensive to generate the required zk-SNARK proof. In order to make zkBridge practical, we must reduce proof generation time.

To this end, we propose two novel ideas. First, we observe that the circuits used by cross-chain bridges are *data-parallel*, in that they contain multiple identical copies of a smaller sub-circuit. Specifically, the circuit for verifying N digital signatures contains N copies of the signature verification sub-circuit. To leverage the data-parallelism, we propose deVirgo, a novel distributed zero-knowledge proofs protocol based on Virgo [ZXZS20]. deVirgo enjoys perfect linear scalability, namely, the proof generation time can be reduced by a factor of M if the generation is distributed over M machines. The protocol is of independent interest and might be useful in other scenarios. Other proof systems can be similarly parallelized [WZCPS18].

While deVirgo significantly reduces the proof generation time, verifying deVirgo proofs on chain, especially for the billion-gate circuits in zkBridge, can be expensive for smart contracts where computational resources are extremely limited. To compress the proof size and the verification cost, we recursively prove the correctness of a (potentially large) deVirgo proof using a classic zk-SNARK due to Groth [Gro16], hereafter denoted Groth16. The Groth16 prover outputs constant-size proofs that are fast to verify by a smart contract on an EVM blockchain. We stress that one cannot use Groth16 to generate the entire zkBridge proof because the circuits needed in zkBridge are too large for a Groth16 prover. Instead, our approach of compressing a deVirgo proof using Groth16 gives the best of both worlds: a fast deVirgo parallel prover for the bulk of the proof, where the resulting proof is compressed into a succinct Groth16 proof that is fast to verify. We elaborate on this technique in Section 5.5. This approach to compressing long proofs is also being adopted in commercial zk-SNARK systems such as [Pold; Polc; Ris].

Implementation and evaluation. To demonstrate the practicality of zkBridge, we implement an end-to-end prototype of zkBridge from Cosmos to Ethereum, given it is among the most challenging directions as it involves large circuits for correctness proofs. Our implementation includes the protocols of deVirgo and recursive proof with Groth16, and the transaction relay application. The experiments show that our system achieves practical performance. deVirgo can generate a block header relay proof within 20s, which is more than 100x faster than the original Virgo system with a single machine. Additionally, the on-chain verification cost decreases from $\sim$80M gas (direct signature verification) to less than 230K gas, due to the recursive proofs. In addition, as a prototype example, we also implement zkBridge from Ethereum to other EVM-compatible chains such as BSC, which involves smaller circuits for proof generation and incurs much less overhead.

5.1.1 Our contribution

In this paper, we make the following contributions:

- In this paper, we propose zkBridge, a trustless, efficient, and secure cross-chain bridge whose security relies on succinct proofs (cryptographic assumptions) rather than a committee (external trust assumptions). Compared with existing cross-chain bridge projects in the wild, zkBridge is the first solution that achieves the following properties at the same time.

  - **Trustless and Secure:** The correctness of block headers on remote blockchains is proven by zk-SNARKs, and thus no external trust assumptions are introduced. Indeed, as long as the connected blockchains and the underlying light-client protocols are secure, and there exists at least one honest node in the block header relay network, zkBridge is secure.
  - **Permissionless and Decentralized:** Any node can freely join the network to relay block headers, generate proofs, and claim the rewards. Due to the elimination of the commonly-used central or Proof-of-Stake style committee for block header validation, zkBridge also enjoys better decentralization.
  - **Extensible:** Smart contracts using zkBridge enjoy maximum flexibility because they can invoke the updater contract to retrieve verified block headers, and then perform their application-specific verification and functionality (e.g., verifying transaction inclusion through auxiliary Merkle proofs). By separating the bridge from application-specific logic, zkBridge makes it easy to develop applications on top of the bridge.
  - **Universal:** The block header relay network and the underlying proof scheme in zkBridge is universal
  - **Efficient:** With our highly optimized recursive proof scheme, block headers can be relayed within a short time (usually tens of seconds for proof generation), and the relayed information can be quickly finalized as soon as the proof is verified, thus supporting fast and flexible bridging of information.

  In summary, zkBridge is a huge leap towards building a secure, trustless foundation for blockchain interoperability.

- We propose a novel 2-layer recursive proof system, which is of independent interest, as the underlying zk-SNARK protocol to achieve both reasonable proof generation time and on-chain verification cost. Through the coordination of deVirgo and Groth16, we achieve a desirable balance between efficiency and cost.

  - For the first layer, aiming at prompt proof generation, we introduce deVirgo, a distributed version of Virgo proof system. deVirgo combines distributed sumcheck and distributed polynomial commitment to achieve optimal parallelism, through which the proof generation phase is much more accelerated by running on distributed machines. deVirgo is more than 100x faster than Virgo for the workload in zkBridge.
  - For the second layer, aiming at acceptable on-chain verification cost, we use Groth16 to recursively prove that the previously generated proof by deVirgo indeed proves the validity of the corresponding remote block headers. Through the second layer, the verification gas cost is reduced from an estimated $\sim 80M$ to less than $230K$, making on-chain verification practical.

- We implement an end-to-end prototype of zkBridge and evaluate its performance in two scenarios: from Cosmos to Ethereum (which is the main focus since it involves large proof circuits that existing systems cannot efficiently handle), and from Ethereum to other EVM-compatible chains (which in comparison involves much smaller circuits). The experiment results show that zkBridge achieves practical performance and is the first practical cross-chain bridge that achieves cryptographic assurance of correctness.

5.2 Background

In this section we cover the preliminaries, essential background on blockchains, and zero-knowledge proofs.

5.2.1 Notations

Let $\mathbb{F}$ be a finite field and λ be a security parameter. We use $f(), h()$ for polynomials, x, y for single variables, bold letters $\mathbf{x}, \mathbf{y}$ for vectors of variables. Both $\mathbf{x}[i]$ and x_i denote the i-th element in $\mathbf{x}$. For $\mathbf{x}$, we use notation $\mathbf{x}[i:k]$ to denote slices of vector $\mathbf{x}$, namely $\mathbf{x}[i:k] = (x_i, x_{i+1}, \cdots, x_k)$. We use $\mathbf{i}$ to denote the vector of the binary representation of some integer i.

Merkle Tree. Merkle tree [Mer87] is a data structure widely used to build commitments to vectors because of its simplicity and efficiency. The prover time is linear in the size of the vector while the verifier time and proof size are logarithmic in the size of the vector. Given a vector of $\mathbf{x} = (x_0, \cdots, x_{N-1})$, it consists of three algorithms:

- $\mathsf{rt} \leftarrow \mathsf{MT.Commit}(\mathbf{x})$
- $(\mathbf{x}[i], \pi_i) \leftarrow \mathsf{MT.Open}(\mathbf{x}, i)$
- $\{1, 0\} \leftarrow \mathsf{MT.Verify}(\pi_i, \mathbf{x}[i], \mathsf{rt})$.

5.2.2 Blockchains

A blockchain is a distributed protocol where a group of nodes collectively maintains a *ledger* which consists of an ordered list of *blocks*. A block blk is a data-structure that stores a header blkH and a list of transactions, denoted by $\mathsf{blk} = \{\mathsf{blkH}; \mathsf{trx}_1, \ldots, \mathsf{trx}_t\}$. A block header contains metadata about the block, including a pointer to the previous block, a compact representation of the transactions (typically a Merkle tree root), validity proofs such as solutions to cryptopuzzles in Proof-of-Work systems or validator signatures in Proof-of-Stake ones.

Security of blockchains. The security of blockchains has been studied extensively. Suppose the ledger in party i's local view is $\mathsf{LOG}_i^r = [\mathsf{blk}_1, \mathsf{blk}_2, \ldots, \mathsf{blk}_r]$ where r is the *height*. For any $2 \leq k \leq r$ and the k-th block blk_k , $\mathsf{blk}_k.\mathsf{ptr} = \mathsf{blkH}_{k-1}$, so every single block is linked to the previous one. For the purpose of this paper, we care about two (informal) properties:

1. **Consistency**: For any honest nodes i and j, and for any rounds of r_0 and r_1, it must be satisfied that either $\mathsf{LOG}_i^{r_0}$ is a prefix of $\mathsf{LOG}_j^{r_1}$ or vice versa.

2. **Liveness**: If an honest node receives some transaction trx at some round r, then trx will be included into the blockchain of all honest nodes eventually.

Smart contracts and gas. In addition to reaching consensus over the content of the ledger, many blockchains support expressive user-defined programs called *smart contracts*, which are stateful programs with state persisted on a blockchain. Without loss of generality, smart contract states can be viewed a key-value store (and often implemented as such.) Users send transactions to interact with a smart contract, and potentially alter its state.

A key limitation of existing smart contract platforms is that computation and storage are scarce resources and can be considerably expensive. Typically smart contract platforms such as Ethereum charge a fee (sometimes called gas) for every step of computation. For instance, EdDSA signatures are extremely cheap to verify (a performant CPU can verify 71000 of them in a second [BDLSY12]), but verifying a single EdDSA signature on Ethereum costs about 500K gas, which is about \$49 at the time of writing. Storage is also expensive on Ethereum. Storing 1KB of data costs about 0.032 ETH, which can be converted to approximately \$90 at the time of writing. This limitation is not unique to Ethereum but rather a reflection of the low capacity of permissionless blockchains in general. Therefore reducing on-chain computation and storage overhead is one of the key goals.

5.2.3 Light client protocol

In a blockchain network, there are full nodes as well as light ones. Full nodes store the entire history of the blockchain and verify all transactions in addition to verifying block headers. Light clients, on the other hand, only store the headers, and therefore can only verify a subset of correctness properties.

The workings of light clients depend on the underlying consensus protocol. The original Bitcoin paper contains a light client protocol (SPV [Nak08]) that uses Merkle proofs to enable a light client who only stores recent headers to verify transaction inclusion. A number of improvements have been proposed ever since. For instance, in Proof-of-Stake, typically a light client needs to verify account balances in the whole blockchain history (or up to a snapshot), and considers the risk of long range attacks. For BFT-based consensus, a light client needs to verify validator signatures and keeps track of validator rotation. We refer readers to [CBC21] for a survey.

To abstract consensus-specific details away, we use

$$\mathsf{LightCC}(\mathsf{LCS}_{r-1}, \mathsf{blkH}_{r-1}, \mathsf{blkH}_r) \to \{\mathsf{true}, \mathsf{false}\}$$

to denote the block validation rule of a light client: given a new block header blkH_r, $\mathsf{LightCC}$ determines if the header represents a valid next block after blkH_{r-1} given its current state LCS_{r-1}. We define the required properties of a light client protocol as follows:

Definition 5.2.1 (Light client protocol)**.** *A light client protocol enables a node to synchronize the block headers of the state of the blockchain. Suppose all block headers in party i's local view is $\mathsf{LOG}H_i^r = [\mathsf{blkH}_1, \mathsf{blkH}_2, ..., \mathsf{blkH}_r]$, the light client protocol satisfies following properties:*

1. ***Succinctness**: For each state update, the light client protocol only takes $O(1)$ time to synchronize the state.*
2. ***Liveness**: If an honest full node receives some transaction trx at some round r, then trx must be included into the blockchain eventually. A light client protocol will eventually include a block header blkH_i such that the corresponding block includes the transaction trx.*
3. ***Consistency**: For any honest nodes i and j, and for any rounds of r_0 and r_1, it must be satisfied that either $\mathsf{LOG}H_i^{r_0}$ is a prefix of $\mathsf{LOG}H_j^{r_1}$ or vice versa.*

5.2.4 Zero-knowledge proofs

An argument system for an NP relationship $\mathcal{R}$ is a protocol between a computationally-bounded prover $\mathcal{P}$ and a verifier $\mathcal{V}$. At the end of the protocol, $\mathcal{V}$ is convinced by $\mathcal{P}$ that there exists a witness $\mathbf{w}$ such that $(\mathbf{x}; \mathbf{w}) \in \mathcal{R}$ for some input $\mathbf{x}$. We use $\mathcal{G}$ to represent the generation phase of the public parameters pp. Formally, consider the definition below, where we assume $\mathcal{R}$ is known to $\mathcal{P}$ and $\mathcal{V}$.

Definition 5.2.2. *Let λ be a security parameter and $\mathcal{R}$ be an NP relation. A tuple of algorithm $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a zero-knowledge argument of knowledge for $\mathcal{R}$ if the following holds.*

- ***Completeness**. For every pp output by $\mathcal{G}(1^\lambda)$, $(\mathbf{x}; \mathbf{w}) \in \mathcal{R}$ and $\pi \leftarrow \mathcal{P}(\mathbf{x}, \mathbf{w}, \mathsf{pp})$,*

$$\Pr[\mathcal{V}(\mathbf{x}, \pi, \mathsf{pp}) = \mathsf{1}] = 1$$

- ***Knowledge Soundness**. For any PPT prover $\mathcal{P}^*$, there exists a PPT extractor $\mathcal{E}$ such that for any auxiliary string $\mathbf{z}$, $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$, $\pi^* \leftarrow \mathcal{P}^*(\mathbf{x}, \mathbf{z}, \mathsf{pp})$, $w \leftarrow \mathcal{E}^{\mathcal{P}^*(\cdot)}(\mathbf{x}, \mathbf{z}, \mathsf{pp})$, and*

$$\Pr[(\mathbf{x}; \mathbf{w}) \notin \mathcal{R} \wedge \mathcal{V}(\mathbf{x}, \pi^*, \mathsf{pp}) = \mathsf{1}] \leq \mathsf{negl}(\lambda),$$

  where $\mathcal{E}^{\mathcal{P}^*(\cdot)}$ represents that $\mathcal{E}$ can rewind $\mathcal{P}^*$,

- ***Zero knowledge**. There exists a PPT simulator $\mathcal{S}$ such that for any PPT algorithm $\mathcal{V}^*$, $(\mathbf{x}; \mathbf{w}) \in \mathcal{R}$, pp output by $\mathcal{G}(1^\lambda)$, it holds that*

$$\mathsf{View}(\mathcal{V}^*(\mathsf{pp}, \mathbf{x})) \approx \mathcal{S}^{\mathcal{V}^*}(\mathbf{x}),$$

  where $\mathsf{View}(\mathcal{V}^*(\mathsf{pp}, \mathbf{x}))$ denotes the view that the verifier sees during the execution of the interactive process with $\mathcal{P}$, $\mathcal{S}^{\mathcal{V}^*}(\mathbf{x})$ denotes the view generated by $\mathcal{S}$ given input $\mathbf{x}$ and transcript of $\mathcal{V}^*$, and $\approx$ denotes two perfectly indistinguishable distributions.

We say that $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a ***succinct*** *argument system*[2] *if the total communication (proof size) between $\mathcal{P}$ and $\mathcal{V}$, as well as $\mathcal{V}$'s running time, are* $\mathsf{poly}(\lambda, |\mathbf{x}|, \log|\mathcal{R}|)$, *where $|\mathcal{R}|$ is the size of the circuit that computes $\mathcal{R}$ as a function of λ.*

5.3 zkBridge Protocol

At a high level, a smart contract is a stateful program with states persisted on a blockchain. A bridge like zkBridge is a service that enables smart contracts on different blockchains to transfer states from one chain to another in a secure and verifiable fashion.

Below we first explain the design of zkBridge and its workflow through an example, then we specify the protocol in more detail. For ease of exposition, we focus on one direction of the bridge, but the operation of the opposite direction is symmetric.

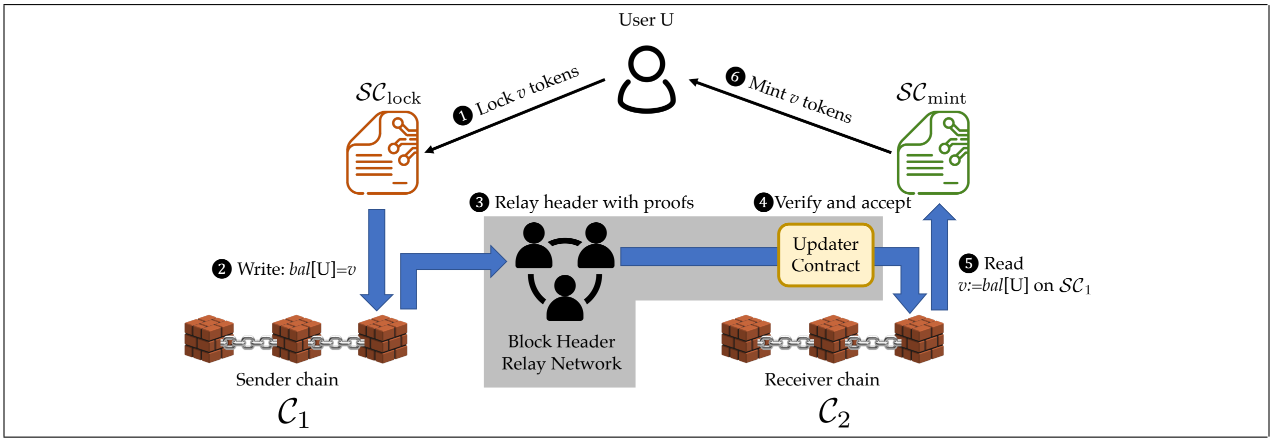


Figure 5.1: The design of zkBridge illustrated with the example of cross-chain token transfer. The components in shade belongs to zkBridge. For clarity we only show one direction of the bridge and the opposite direction is symmetric.

5.3.1 Overview of zkBridge design

To make it easy for different applications to integrate with zkBridge, we adopt a modular design where we separate application-specific logic (e.g., verifying smart contract states) from the core bridge functionality (i.e., relaying block headers).

Figure 5.1 shows the architecture and workflow of zkBridge. The core bridge functionality is provided by a **block header relay network** (trusted only for liveness) that relays block headers of $\mathcal{C}_1$ along with correctness proofs, and an **updater contract** on $\mathcal{C}_2$ that verifies and accepts proofs submitted by relay nodes. The updater contract maintains a list of recent block headers, and updates it properly after verifying proofs submitted by relay nodes; it exposes a simple and application-agnostic API, from which application smart

[2] In our construction, we only need a succinct non-interactive arguments of knowledge (SNARK) satisfying the first two properties and the succinctnes for validity. The zero knowledge property could be used to further achieve privacy.

contracts can obtain the latest block headers of the sender blockchain and build application-specific logic on top of it.

Applications relying on zkBridge will typically deploy a pair of contracts, a sender contract and a receiver contract on $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. We refer to them collectively as application contracts or relying contracts. The receiver contract can call the updater contract to obtain block headers of $\mathcal{C}_1$, based on which they can perform application specific tasks. Depending on the application, receiver contracts might also need a user or a third party to provide application-specific proofs, such as Merkle proofs for smart contract states.

As an example, Fig. 5.1 shows the workflow of *cross-chain token transfer*, a common use case of bridges, facilitated by zkBridge. Suppose a user $\mathcal{U}$ wants to trade assets (tokens) she owns on blockchain $\mathcal{C}_1$ in an exchange residing on another blockchain $\mathcal{C}_2$ (presumably because $\mathcal{C}_2$ charges lower fees or has better liquidity), she needs to move her funds from $\mathcal{C}_1$ to $\mathcal{C}_2$. A pair of smart contracts $\mathcal{SC}_{\text{lock}}$ and $\mathcal{SC}_{\text{mint}}$ are deployed on blockchains $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. To move the funds, the user locks $\$\texttt{v}$ tokens in $\mathcal{SC}_{\text{lock}}$ (step ① in Fig. 5.1) and then requests $\$\texttt{v}$ tokens to be issued by $\mathcal{SC}_{\text{mint}}$. To ensure solvency, $\mathcal{SC}_{\text{mint}}$ should only issue new tokens if and only if the user has locked tokens on $\mathcal{C}_1$. This requires $\mathcal{SC}_{\text{mint}}$ to read the states of $\mathcal{SC}_{\text{lock}}$ (the balance of $\mathcal{U}$, updated in step ②) from a different blockchain, which it cannot do directly. zkBridge enables this by relaying the block headers of $\mathcal{C}_1$ to $\mathcal{C}_2$ along with proofs (step ③ and ④). $\mathcal{SC}_{\text{mint}}$ can retrieve the block headers from the smart contract frontend (the updater contract), check that the balance of user $\mathcal{U}$ is indeed $\$\texttt{v}$ (step ⑤), and only then mint $\$\texttt{v}$ tokens (Step ⑥).

Besides cross-chain token transfer, zkBridge can also enable various other applications such as cross-chain collateralized loans, general message passing, etc. We present three use cases in Section 5.3.3.

5.3.2 Protocol detail

Having presented the overview, in this section, we specify the protocol in more detail.

Security and system model

For the purpose of modeling bridges, we model a blockchain $\mathcal{C}$ as a block-number-indexed key-value store, denoted as $\mathcal{C}[t] : \mathcal{K} \to \mathcal{V}$ where t is the block number, $\mathcal{K}$ and $\mathcal{V}$ are key and value spaces respectively. In Ethereum, for example, $\mathcal{V} = \{0,1\}^{256}$ and keys are the concatenation of a smart contract identifier $\mathcal{SC}$ and a per-smart-contract storage address K. For a given contract $\mathcal{SC}$, we denote the value stored at address K at block number t as $\mathcal{SC}[t, K]$, and we call $\mathcal{SC}[t, \cdot]$ the *state* of $\mathcal{SC}$ at block number t. Again, for ease of exposition, we focus on the direction from $\mathcal{SC}_1$ to $\mathcal{SC}_2$, denoted as $\mathcal{BR}[\mathcal{SC}_1 \to \mathcal{SC}_2]$.

Functional and security goals. We require the bridge $\mathcal{BR}[\mathcal{SC}_1 \to \mathcal{SC}_2]$ to reflect states of $\mathcal{SC}_1$ correctly and timely:

1. **Correctness**: For all t, K, $\mathcal{SC}_2$ accepts a wrong state $V \neq \mathcal{SC}_1[t, K]$ with negligible probability.

2. **Liveness**: Suppose $\mathcal{SC}_2$ needs to verify $\mathcal{SC}_1$'s state at (t, K), the bridge will provide necessary information eventually.

Security assumptions. For correctness, zkBridge does not introduce extra trust assumptions besides those made by the underlying blockchains. Namely, we assume both the sender blockchain and the receiver blockchain are consistent and live (Section 5.2), and the sender chain has a light client protocol to enable fast block header verification. For both properties, we assume there is at least one honest node in the relay network, and that the zk-SNARK used is sound.

Construction of zkBridge

As described in Section 5.3, a bridge $\mathcal{BR}[\mathcal{SC}_1 \to \mathcal{SC}_2]$ consists of three components: a block header relay network, a updater contract, and one or more application contracts. Below we specify the protocols for each component.

Block header relay network. We present the formal protocol of block header relay network in Protocol 14.

Protocol 14 Block header relay network

```
procedure RelayNextHeader($\mathsf{LCS}_{r-1}, \mathsf{blkH}_{r-1}$)
    Contact $k$ different full nodes to get the block headers following $\mathsf{blkH}_{r-1}$, namely $\mathsf{blkH}_r$.
    Generate a ZKP $\pi$ proving
        $$\mathsf{LightCC}(\mathsf{LCS}_{r-1}, \mathsf{blkH}_{r-1}, \mathsf{blkH}_r) \to \mathsf{true}.$$
    Send $(\pi, \mathsf{blkH}_r, \mathsf{blkH}_{r-1})$ to the updater contract.
```

Nodes in the block header relay network run RelayNextHeader with the current state of the updater contract $(\mathsf{LCS}_{r-1}, \mathsf{blkH}_{r-1})$ as input. The exact definition of LCS_{r-1} is specific to light client protocols (see [CBC21] for a survey). The relay node then connects to full nodes in $\mathcal{C}_1$ and gets the block header blkH_r following blkH_{r-1}. The relay node generates a ZKP π showing the correctness of blkH_r, by essentially proving that blkH_r is accepted by a light client of $\mathcal{C}_1$ after block blkH_{r-1}. It then sends (π, blkH_r) to the updater contract on $\mathcal{C}_2$. To avoid the wasted proof time due to collision (note that when multiple relay nodes send at the same time, only one proof can be accepted), relay nodes can coordinate using standard techniques (e.g., to send in a round robin fashion). While any zero-knowledge proofs protocol could be used, our highly optimized one will be presented later in Section 5.4.

To incentivize block header relay nodes, provers may be rewarded with fees after validating their proofs. We leave incentive design for future work. A prerequisite of any incentive scheme is unstealability [SCPTZ21], i.e., the guarantee that malicious nodes cannot steal others' proofs. To this end, provers will embed their identifiers (public keys) in proofs, e.g., as input to the hash function in the Fiat-Shamir heuristic [FS86].

We note that this design relies on the security of the light client verifier of the sender chain. For example, the light client verifier must reject a valid block header that may eventually become orphaned and not part of the sender chain.

The updater contract. The protocol for the updater contract is specified in Protocol 15.

The updater contract maintains the light client's internal state including a list of block headers of $\mathcal{C}_1$ in headerDAG. It has two publicly exposed functions. The HeaderUpdate function can be invoked by any block header relay node, providing supposedly the next block header and a proof as input. If the proof verifies against the current light client state LCS and blkH_{r-1}, the contract will do further light-client checks, and then the state will be updated accordingly. Since the caller of this function must pay a fee, DoS attacks are naturally prevented.

The GetHeader function can be called by receiver contracts to get the block header at height t. Receiver contracts can use the obtained block header to finish application-specific verification, potentially with the help of a user or some third party.

Application contracts. zkBridge has a modular design in that the updater contract is application-agnostic. Therefore in $\mathcal{BR}[\mathcal{SC}_1 \to \mathcal{SC}_2]$, it is up to the application contracts $\mathcal{SC}_1$ and $\mathcal{SC}_2$ to decide what the informa-

Protocol 15 The updater contract

```
headerDAG := $\emptyset$                                         ▷ DAG of headers
LCS := $\perp$                                                   ▷ light client state
procedure HeaderUpdate($\pi$, $\mathsf{blkH}_r$, $\mathsf{blkH}_{r-1}$)
    if $\mathsf{blkH}_{r-1} \notin$ headerDAG then
        return False                          ▷ skip if parent block is not in the DAG
    if $\pi$ verifies against LCS, $\mathsf{blkH}_{r-1}$, $\mathsf{blkH}_r$ then
        Update LCS according to the light client protocol.
        Insert $\mathsf{blkH}_r$ into headerDAG.
procedure GetHeader($t$)                   ▷ $t$ is a unique identifier to a block header
    if $t \notin$ headerDAG then
        return $\perp$                                               ▷ tell the caller to wait
    else
        return headerDAG[$t$], LCS    ▷ The LCS will help users to determine if $t$ is on a fork.
```

tion to bridge is. Generally, proving that $\mathcal{SC}_1[t, K] = V$ is straightforward: $\mathcal{SC}_2$ can request for a Merkle proof for the leaf of the state Trie Tree (at block number t) corresponding to address K. The receiver contract can obtain blkH_t from the updater contract by calling the function GetHeader(t). Then it can verify $\mathcal{SC}_1[t, K] = V$ against the Merkle root in blkH_t. Required Merkle proofs are application-specific, and are typically provided by the users of $\mathcal{SC}_2$, some third party, or the developer/maintainer of $\mathcal{SC}_2$.

Security arguments. The security of zkBridge is stated in the following theorem.

Theorem 5.3.1. *The bridge $\mathcal{BR}[\mathcal{SC}_1 \to \mathcal{SC}_2]$ implemented by protocols 14 and 15 satisfies both consistency and liveness, assuming the following holds:*

1. *there is at least one honest node in the block header relay network;*
2. *the sender chain is consistent and live;*
3. *the sender chain has a light-client verifier as in Def. 5.2.1; and*
4. *the succinct proof system is sound.*

Proof (sketch). To prove the consistency of DAG, we first need to convert the DAG into a list of blocks to match the definition of blockchain consistency. We define an algorithm *Longest* : DAG $\to$ List such that given a DAG, the algorithm will output a list MainChain representing the main chain. For example, if the sender chain is Ethereum, the algorithm *Longest* will first calculate the path with the maximum total difficulty in the DAG represented by L, and then output MainChain := L[: $-K$]. Here K is a security parameter. By assumption 1 and 2, there will be an honest node in our system running either a full node or a light node, which will be consistent with the sender chain. Also, according to assumption 1, at least one prover node is honestly proving the light client execution. By assumption 4 that the proof system is sound, the updater contract will correctly verify the light-client state. We argue that the updater contract is correctly running the light-client protocol. Therefore, by the consistency of the light-client protocol, MainChain will be consistent with any other honest node.

The liveness of our protocol directly follows from the liveness of $\mathcal{C}_1$ and its light client protocol. ∎

5.3.3 Application use cases

In this section, we present three examples of applications that zkBridge can support.

Transaction inclusion: a building block. A common building block of cross-chain applications is to verify transaction inclusion on another blockchain. Specifically, the goal is to enable a receiver contract $\mathcal{SC}_2$ on $\mathcal{C}_2$ to verify that a given transaction trx has been included in a block B_t on $\mathcal{C}_1$ at height t. To do so, the receiver contract $\mathcal{SC}_2$ needs a user or a third-party service to provide the Merkle proof for trx in B_t. Then, $\mathcal{SC}_2$ will call the updater contract to retrieve the block header of $\mathcal{C}_1$ at height t, and then verify the provided Merkle proof against the Merkle root contained in the header.

Next, we will present three use cases that extend the building block above.

1. Message passing and data sharing. Cross-chain message passing is another common building block useful for, e.g., sharing off-chain data cross blockchains.

Message passing can be realized as a simple extension of transaction inclusion, by embedding the message in a transaction. Specifically, to pass a message m from $\mathcal{C}_1$ to $\mathcal{C}_2$, a user can embed m in a transaction trx_m, send trx_m to $\mathcal{C}_1$, and then execute the above transaction inclusion proof.

2. Cross-chain assets transfer/swap. Bridging native assets is a common use case with growing demand. In this application, users can stake a certain amount of token T_A on the sender blockchain $\mathcal{C}_1$, and get the same amount of token T_A (for native assets transfer if eligible) or a certain amount of token T_B of approximately the same value (for native assets swap) on the receiver blockchain $\mathcal{C}_2$. With the help of the transaction inclusion proof, native assets transfer/swap can be achieved, as illustrated at a high level in Section 5.3.1. Here we specify the protocol in more detail.

To set up, the developers will deploy a lock contract $\mathcal{SC}_{\text{lock}}$ on $\mathcal{C}_1$ and a mint contract $\mathcal{SC}_{\text{mint}}$ on $\mathcal{C}_2$. For a user who wants to exchange n_A of token T_A for an equal value in token T_B, she will first send a transaction $\mathsf{trx}_{\text{lock}}$ that transfers n_A of token T_A to $\mathcal{SC}_{\text{lock}}$, along with an address $addr_{\mathcal{C}_2}$ to receive token T_B on $\mathcal{C}_2$. After $\mathsf{trx}_{\text{lock}}$ is confirmed in a block B, the user will send a transaction $\mathsf{trx}_{\text{mint}}$ to $\mathcal{SC}_{\text{mint}}$, including sufficient information to verify the inclusion of $\mathsf{trx}_{\text{lock}}$. Based on information in $\mathsf{trx}_{\text{mint}}$, $\mathcal{SC}_{\text{mint}}$ will verify that $\mathsf{trx}_{\text{lock}}$ has been included on $\mathcal{C}_1$, and transfer the corresponding T_B tokens to the address $addr_{\mathcal{C}_2}$ specified in $\mathsf{trx}_{\text{lock}}$. Finally, $\mathcal{SC}_{\text{mint}}$ will mark $\mathsf{trx}_{\text{lock}}$ as minted to conclude the transfer.

3. Interoperations for NFTs. In the application of Non-fungible Token (NFT) interoperations, users always lock/stake the NFT on the sender blockchain, and get minted NFT or NFT derivatives on the receiver blockchain. By designing the NFT derivatives, the cross-chain protocol can separate the ownership and utility of an NFT on two blockchain systems, thus supporting locking the ownership of the NFT on the sender blockchain and getting the utility on the receiver blockchain.

5.3.4 Efficient Proof Systems for zkBridge

The most computationally demanding part of zkBridge is the zero-knowledge proofs generation that relay nodes must do for every block. So far we have abstracted away the detail of proof generation, which we will address in Sections 5.4 and 5.5. Here, we present an overview of our solution.

For Proof-of-Stake chains, the proofs involve verifying hundreds of signatures. A major source of overhead is field transformation between different elliptic curves when the sender and receiver chains use different cryptography implementation, which is quite common in practice. For example, Cosmos uses EdDSA on Curve25519 whereas Ethereum natively supports a different curve BN254. The circuit for verifying a single Cosmos signature in the field supported by Ethereum involves around 2 million gates, thus verifying a block (typically containing 32 signatures) will involve over 64 million gates, which is too big for existing zero-knowledge proofs schemes.

To make zkBridge practical, we propose two ideas.

Reducing proof time with deVirgo We observe that the ZKP circuit for verifying multiple signatures is composed of multiple copies of one sub-circuit. Our first idea is to take advantage of this special structure and distribute proof generation across multiple servers. We propose a novel *distributed ZKP protocol* dubbed deVirgo, which carefully parallelizes the Virgo [ZXZS20] protocol, one of the fastest ZKP systems (in terms of prover time) without a trusted setup. With deVirgo, we can accelerate proof generation in zkBridge with perfect linear scalability. We will dive into the detail of deVirgo in Section 5.4.

Reducing on-chain cost by recursive verification. While verifying deVirgo proofs on ordinary CPUs is very efficient, on-chain verification is still costly. To further reduce the on-chain verification cost (computation and storage), we use *recursive verification*: the prover recursively proves the correctness of a (potentially large) Virgo proof using a smart-contract-friendly zero-knowledge protocol to get a small and verifier-efficient proof. At a high level, we trade slightly increased proof generation time for much reduced on-chain verification cost: the proof size reduces from $200+$KB to 131 bytes, and the required computation reduces from infeasible amount of gas to 210K gas. We will present more detail of recursive verification in Section 5.5.

5.4 Distributed proof generation

As observed previously, the opportunity for fast prover time stems from the fact that the circuit for verifying N signatures consists of N copies of identical sub-circuits. This type of circuits is called data-parallel [Tha15]. The advantage of data-parallel circuits is that there is no connection among different sub-copies. Therefore, each copy can be handled separately. We consider accelerating the proof generation on such huge circuits by dealing with each sub-circuit in parallel. In this section, we propose a distributed zk-SNARK protocol on data-parallel circuits.

There are many zero knowledge proofs protocols [ZXZS20; XZZPS19b; Set20; WTSTW18; BSCTV14c; BSCRSVW19; AHIV17; BSBHR19; Zha+21a; GWC19b; COS20] supporting our computation. We choose Virgo as the underlying ZKP protocols for two reasons: 1. Virgo does not need a trusted setup and is plausibly post-quantum secure. 2. Virgo is one of the fastest protocols with succinct verification time and succinct proof size for problems in large scale. We present a new distributed version of Virgo for data-parallel arithmetic circuits achieving optimal scalability without any overhead on the proof size. Specifically, our protocol of deVirgo on data-parallel circuits with N copies using N parallel machines is N times faster than the original Virgo while the proof size remains the same. Our scheme is of independent interest and is possible to be used in other Virgo-based systems to improve the efficiency.

We provide the overall description of deVirgo as follows. Suppose the prover has N machines in total, labeled from $\mathcal{P}_0$ to $\mathcal{P}_{N-1}$. Assume $\mathcal{P}_0$ is the master node while other machines are ordinary nodes. Assume $\mathcal{V}$ is the verifier. Given a data-parallel arithmetic circuit consisting of N identical structures, the naïve algorithm of the distributed Virgo is to assign each sub-circuit to a separate node. Then each node runs Virgo to generate the proof separately. The concatenation of N proofs is the final proof. Unfortunately, the proof size in this naive algorithm scales linearly in the number of sub-circuits, which can be prohibitively large for data-parallel circuits with many sub-copies. To address the problem, our approach removes the additional factor of N in the proof size by aggregating messages and proofs among distributed machines. Specifically, the original protocol of Virgo consists of two major building blocks. One is the GKR protocol [GKR15], which consists of d sumcheck protocols [LFKN92] for a circuit of depth d. The other is the polynomial commitment (PC) scheme. We design distributed schemes for each of the sumcheck and the polynomial commitment (PC). In our distributed sumcheck protocol, a master node $\mathcal{P}_0$ aggregates messages from all machines, then sends the aggregated message to $\mathcal{V}$ in every round, instead of sending messages from all machines directly to $\mathcal{V}$. Our protocol for distributed sumcheck has exactly the same proof size as the original sumcheck protocol, thus saving a factor N over the naïve distributed protocol. Additionally, in our distributed PC protocol, we optimize the commitment phase and make $\mathcal{P}_0$ aggregate N commitments into one instead of sending N commitments directly to $\mathcal{V}$. During the opening phase, the proof can also be aggregated, which improves the proof size by a logarithmic factor in the size of the polynomial.

We present preliminaries in Section 5.4.1, the detail of the distributed sumcheck protocol in Section 5.4.2 and the detail of the distributed PC protocol in Section 5.4.3. We combine them all together to build deVirgo in Section 5.4.4.

5.4.1 Preliminaries

Multi-linear extension/polynomial. Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. The *multi-linear extension/polynomial* of V is the unique polynomial $\tilde{V} : \mathbb{F}^\ell \to \mathbb{F}$ such that $\tilde{V}(\mathbf{x}) = V(\mathbf{x})$ for all $\mathbf{x} \in \{0,1\}^\ell$. $\tilde{V}$ can be expressed as:

$$\tilde{V}(\mathbf{x}) = \sum\nolimits_{\mathbf{b}\in\{0,1\}^\ell} \prod\nolimits_{i=1}^{\ell}((1-x_i)(1-b_i)+x_ib_i)) \cdot V(\mathbf{b}),$$

where b_i is i-th bit of b.

Identity function. Let $\beta : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}$ be the identity function such that $\beta(\mathbf{x},\mathbf{y}) = 1$ if $\mathbf{x} = \mathbf{y}$, and $\beta(\mathbf{x},\mathbf{y}) = 0$ otherwise. Suppose β is the multilinear extension of β. Then β can be expressed as: $\beta(\mathbf{x},\mathbf{y}) = \prod_{i=1}^{\ell}((1-x_i)(1-y_i)+x_iy_i)$.

5.4.2 Distributed sumcheck

Background: the sumcheck protocol. The sumcheck problem is to sum a multivariate polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ over all binary inputs: $\sum_{b_1,\cdots,b_\ell\in\{0,1\}} f(b_1,\cdots,b_\ell)$. The sumcheck protocol allows the prover $\mathcal{P}$ to convince the verifier $\mathcal{V}$ that the summation is H via a sequence of interactions, and the formal protocol is presented in Protocol 8.

The high-level idea of the sumcheck protocol is to divide the verification into ℓ rounds. In each round, the prover only sends a univariate polynomial to the verifier. The verifier checks the correctness of the polynomial by a single equation. Then this variable will be replaced by a random point sampled by the verifier. As there are totally ℓ variables in f, after ℓ rounds, the claim about the summation will be reduced to a claim about f on a random vector $\mathbf{r}$. Given the oracle access to f on a random vector, the verifier can check the last claim.

The GKR protocol. We follow the convention in prior works of GKR protocols [CMT12; Tha13b; ZGKPP17d; XZZPS19b; ZXZS20]. We denote the number of gates in the i-th layer as S_i and let $s_i =$

Protocol 8 (**Sumcheck**). The protocol proceeds in ℓ rounds.

- In the first round, $\mathcal{P}$ sends a univariate polynomial

$$f_1(x_1) \overset{def}{=} \sum_{b_2,\ldots,b_\ell \in \{0,1\}} f(x_1, b_2, \ldots, b_\ell),$$

  $\mathcal{V}$ checks $H = f_1(0) + f_1(1)$. Then $\mathcal{V}$ sends a random challenge $r_1 \in \mathbb{F}$ to $\mathcal{P}$.

- In the i-th round, where $2 \leq i \leq \ell - 1$, $\mathcal{P}$ sends a univariate polynomial

$$f_i(x_i) \overset{def}{=} \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} f(r_1, \ldots, r_{i-1}, x_i, b_{i+1}, \ldots, b_\ell),$$

  $\mathcal{V}$ checks $f_{i-1}(r_{i-1}) = f_i(0) + f_i(1)$, and sends a random challenge $r_i \in \mathbb{F}$ to $\mathcal{P}$.

- In the ℓ-th round, $\mathcal{P}$ sends a univariate polynomial

$$f_\ell(x_\ell) \overset{def}{=} f(r_1, r_2, \ldots, r_{l-1}, x_\ell),$$

  $\mathcal{V}$ checks $f_{\ell-1}(r_{\ell-1}) = f_\ell(0) + f_\ell(1)$. The verifier generates a random challenge $r_\ell \in \mathbb{F}$. Given oracle access to an evaluation $f(r_1, r_2, \ldots, r_\ell)$ of f, $\mathcal{V}$ will accept if and only if $f_\ell(r_\ell) = f(r_1, r_2, \ldots, r_\ell)$. The oracle access can be instantiated by PC.

$\lceil \log S_i \rceil$. (For simplicity, we assume S_i is a power of 2, and we can pad the layer with dummy gates otherwise.) We then define a function $V_i : \{0,1\}^{s_i} \rightarrow \mathbb{F}$ that takes a binary string $\mathbf{b} \in \{0,1\}^{s_i}$ and returns the output of gate $\mathbf{b}$ in layer i, where $\mathbf{b}$ is called the gate label. With this definition, V_0 corresponds to the output of the circuit, and V_d corresponds to the input layer. Finally, we define two additional functions $add_i, mult_i : \{0,1\}^{s_{i-1}+2s_i} \rightarrow \{0,1\}$, referred to as *wiring predicates* in the literature. add_i ($mult_i$) takes one gate label $\mathbf{z} \in \{0,1\}^{s_{i-1}}$ in layer $i-1$ and two gate labels $\mathbf{x}, \mathbf{y} \in \{0,1\}^{s_i}$ in layer i, and outputs 1 if and only if gate $\mathbf{z}$ is an addition (multiplication) gate that takes the output of gate $\mathbf{x}, \mathbf{y}$ as input. With these definitions, for any $\mathbf{z} \in \{0,1\}^{s_i}$, V_i can be written as:

$$\begin{aligned} V_i(\mathbf{z}) = \textstyle\sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} (&add_{i+1}(\mathbf{z},\mathbf{x},\mathbf{y})(V_{i+1}(\mathbf{x}) + V_{i+1}(\mathbf{y})) \\ &+ mult_{i+1}(\mathbf{z},\mathbf{x},\mathbf{y}) V_{i+1}(\mathbf{x}) V_{i+1}(\mathbf{y})) \end{aligned} \tag{5.1}$$

In the equation above, V_i is expressed as a summation, so $\mathcal{V}$ can use the sumcheck protocol to check that it is computed correctly. As the sumcheck protocol operates on polynomials defined on $\mathbb{F}$, we rewrite the equation with their multi-linear extensions:

$$\begin{aligned}\tilde{V}_i(\mathbf{g}) &= \sum\nolimits_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} h_i(\mathbf{g},\mathbf{x},\mathbf{y}) \\ &= \sum\nolimits_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(\mathbf{g},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &\quad + \tilde{mult}_{i+1}(\mathbf{g},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}))\,, \end{aligned} \tag{5.2}$$

where $\mathbf{g} \in \mathbb{F}^{s_i}$ is a random vector.

With Equation 5.2, the GKR protocol proceeds as following. The prover $\mathcal{P}$ first sends the claimed output of the circuit to $\mathcal{V}$. From the claimed output, $\mathcal{V}$ defines polynomial $\tilde{V}_0$ and computes $\tilde{V}_0(\mathbf{g})$ for a random $\mathbf{g} \in \mathbb{F}^{s_0}$. $\mathcal{V}$ and $\mathcal{P}$ then invoke a sumcheck protocol on Equation 5.2 with $i = 0$. As described in Protocol 8, at the end of the sumcheck, $\mathcal{V}$ needs an oracle access to $h_i(\mathbf{g},\mathbf{u},\mathbf{v})$, where $\mathbf{u},\mathbf{v}$ are randomly selected in $\mathbb{F}^{s_{i+1}}$. To compute $h_i(\mathbf{g},\mathbf{u},\mathbf{v})$, $\mathcal{V}$ computes $\tilde{add}_{i+1}(\mathbf{g},\mathbf{u},\mathbf{v})$ and $\tilde{mult}_{i+1}(\mathbf{g},\mathbf{u},\mathbf{v})$ locally (they only depend on the wiring pattern of the circuit, not on the values), asks $\mathcal{P}$ to send $\tilde{V}_1(\mathbf{u})$ and $\tilde{V}_1(\mathbf{v})$ and computes $h_i(\mathbf{g},\mathbf{u},\mathbf{v})$ to complete the sumcheck protocol. In this way, $\mathcal{V}$ and $\mathcal{P}$ reduce a claim about the output to two claims about values in layer 1. $\mathcal{V}$ and $\mathcal{P}$ could invoke two sumcheck protocols on $\tilde{V}_1(\mathbf{u})$ and $\tilde{V}_1(\mathbf{v})$ recursively to layers above, but the number of the sumcheck protocols would increase exponentially.

One way to combine two claims $\tilde{V}_i(\mathbf{u})$ and $\tilde{V}_i(\mathbf{v})$ is using random linear combinations, as proposed in [CFS17; WTSTW18]. Upon receiving the two claims $\tilde{V}_i(\mathbf{u})$ and $\tilde{V}_i(\mathbf{v})$, $\mathcal{V}$ selects $\alpha_{i,1}, \alpha_{i,2} \in \mathbb{F}$ randomly and computes $\alpha_{i,1}\tilde{V}_i(\mathbf{u}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v})$. Based on Equation 5.2, this random linear combination can be written as

$$\begin{aligned} &\alpha_{i,1}\tilde{V}_i(\mathbf{u}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v}) \\ =&\alpha_{i,1} \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ +&\tilde{mult}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}) \\ +&\alpha_{i,2} \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ +&\tilde{mult}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}) \\ =& \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} (\alpha_{i,1}\tilde{add}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &+ (\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}) \end{aligned} \tag{5.3}$$

$\mathcal{V}$ and $\mathcal{P}$ then execute the sumcheck protocol on Equation 5.3 instead of Equation 5.2. At the end of the sumcheck protocol, $\mathcal{V}$ still receives two claims about $\tilde{V}_{i+1}$, computes their random linear combination and proceeds to the layer above recursively until the input layer. The formal GKR protocol is presented in Protocol 9.

Complexity of the sumcheck protocol in GKR protocol. For simplicity in the complexity analysis, we define the sumcheck equation in GKR protocol as

$$\tilde{V}_i(\mathbf{g}) = \sum_{\mathbf{x}\in\{0,1\}^{\ell}} f(\mathbf{x}, \tilde{V}_{i+1}(\mathbf{x})), \tag{5.4}$$

Protocol 9 (**GKR**). Let $\mathbb{F}$ be a finite field. Let $C\colon \mathbb{F}^m \to \mathbb{F}^k$ be a d-depth layered arithmetic circuit. $\mathcal{P}$ wants to convince that $C(\mathbf{x}) = \mathbf{1}$ where $\mathbf{x}$ is the input from $\mathcal{V}$, and $\mathbf{1}$ is the output. Without loss of generality, assume m and k are both powers of 2 and we can pad them if not.

1. $\mathcal{V}$ chooses a random $\mathbf{g} \in \mathbb{F}^{s_0}$ and sends it to $\mathcal{P}$.
2. $\mathcal{P}$ and $\mathcal{V}$ run a sumcheck protocol on

$$1 = \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_1}} (\tilde{add}_1(\mathbf{g}^{(0)},\mathbf{x},\mathbf{y})(\tilde{V}_1(\mathbf{x}) + \tilde{V}_1(\mathbf{y})) + \tilde{mult}_1(\mathbf{g}^{(0)},\mathbf{x},\mathbf{y})\tilde{V}_1(\mathbf{x})\tilde{V}_1(\mathbf{y}))$$

   At the end of the protocol, $\mathcal{V}$ receives $\tilde{V}_1(\mathbf{u}^{(1)})$ and $\tilde{V}_1(\mathbf{v}^{(1)})$. $\mathcal{V}$ computes $\tilde{mult}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$, $\tilde{add}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$ and checks that $\tilde{add}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$ $(\tilde{V}_1(\mathbf{u}^{(1)}) + \tilde{V}_1(\mathbf{v}^{(1)})) + \tilde{mult}_1(\mathbf{g}^{(0)},\mathbf{u}^{(1)},\mathbf{v}^{(1)})$ $\tilde{V}_1(\mathbf{u}^{(1)})\tilde{V}_1(\mathbf{v}^{(1)})$ equals to the last message of the sumcheck.

3. For $i = 1, ..., d-1$:
   - $\mathcal{V}$ randomly selects $\alpha_{i,1}, \alpha_{i,2} \in \mathbb{F}$ and sends them to $\mathcal{P}$.
   - $\mathcal{P}$ and $\mathcal{V}$ run the sumcheck on the equation

$$\alpha_{i,1}\tilde{V}_i(\mathbf{u}^{(i)}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v}^{(i)}) =$$
$$\sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} ((\alpha_{i,1}\tilde{add}_{i+1}\mathbf{u}^{(i)},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v}^{(i)},\mathbf{x},\mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y}))$$
$$+(\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u}^{(i)},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v}^{(i)},\mathbf{x},\mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}))$$

   - At the end of the sumcheck protocol, $\mathcal{P}$ sends $\mathcal{V}$ $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$.
   - $\mathcal{V}$ computes the right-hand side of the above equation by replacing $\mathbf{x}$ and $\mathbf{y}$ by $\mathbf{u}^{(i+1)}$ and $\mathbf{v}^{(i+1)}$ respectively. checks if it equals to the last message of the sumcheck. If all checks in the sumcheck pass, V uses $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$ to proceed to the $(i+1)$-th layer. Otherwise, $\mathcal{V}$ outputs `0` and aborts.

4. At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$. $\mathcal{V}$ evaluates $\tilde{V}_d$ at $\mathbf{u}^{(d)}$ and $\mathbf{v}^{(d)}$ using the input and checks that they are the same as the two claims. If yes, output `1`; otherwise, output `0`.

where f is some polynomial from $\mathbb{F}^{\ell}$ to $\mathbb{F}$ and $\mathbf{g}$ is a random vector in $\mathbb{F}^{\ell}$. For the multivariate polynomial of f defined in Equation 5.4, the prover time in Protocol 8 is $O(2^{\ell})$. The proof size is $O(\ell)$ and the verifier time is $O(\ell)$.

In the setting of data-parallel circuits, we distribute the sumcheck polynomial f among parallel machines. Suppose the data-parallel circuit C consists of N identical sub-circuits of $\mathsf{C}_0, \cdots, \mathsf{C}_{N-1}$ and $N = 2^n$ for some integer n without loss of generality. The polynomial $f : \mathbb{F}^{\ell} \to \mathbb{F}$ is defined on C by Equation 5.4.

The idea of our distributed sumcheck protocol is to treat each sub-copy as a new circuit as there is no wiring connections across different sub-circuits. We define polynomials of $f^{(0)}, \cdots, f^{(N-1)}$ on $\mathsf{C}_0, \cdots, \mathsf{C}_{N-1} : \mathbb{F}^{\ell-n} \to \mathbb{F}$ respectively by Equation 5.4 in the GKR protocol, which have the same form as f defined on C. The naïve approach is running the sumcheck protocol on these polynomials separately. As there are N proofs in total and each size is $O(\ell - n)$, the total proof size will be $O(N(\ell - n))$. To reduce the proof size back to ℓ, the prover needs to aggregate N proofs to generate a single proof on f. We observe that the sumcheck protocol on data-parallel circuits satisfies $f^{(i)}(\mathbf{x}) = f(\mathbf{x}, \mathbf{i})$. As shown in Protocol 8, the protocol proceeds for ℓ variables round by round. We first run the sumcheck protocol on variables that are irrelevant to the index of sub-copies in the circuit. In the first $(\ell - n)$ rounds, each prover $\mathcal{P}_i$ generates the univariate polynomial of $f_j^{(i)}(x_j)$ for $f^{(i)}(\mathbf{x})$ and sends it to $\mathcal{P}_0$. $\mathcal{P}_0$ constructs the univariate polynomial for $f_j(x_j)$ by summing $f_j^{(i)}(x_j)$ altogether since $f_j(x_j) = \sum_{i=0}^{N} f_j^{(i)}(x_j)$, and sends $f_j(x_j)$ to $\mathcal{V}$ in the j-th round. The aggregation among parallel machines reduces the proof size to constant in each round. Hence the final proof size is only $O(\ell)$. A similar approach has appeared in [WHGSW16]. The main focus of [WHGSW16] was improving the prover time of the sumcheck protocol in the GKR protocol to $O(2^{\ell}(\ell-n))$ for data-parallel circuits, which was later subsumed by [XZZPS19b] with a prover running in $O(2^{\ell})$ time. Instead, our scheme is focused on improving the prover time by N times with distributed computing on N machines without any overhead on the proof size.

With this idea in mind, we rewrite the sumcheck equation on f as follows.

$$H = \sum_{\mathbf{b} \in \{0,1\}^{\ell}} f(\mathbf{b}) = \sum_{i=0}^{N-1} \sum_{\mathbf{b} \in \{0,1\}^{\ell-n}} f^{(i)}(\mathbf{b}).$$

Then we divide the original sumcheck protocol on f into 3 phases naturally in the setting of distributed computing. We present the formal protocol of distributed sumcheck in Protocol 10.

1. From round 1 to round $(\ell - n)$ (step 1 in Protocol 10), $\mathcal{P}_i$ runs the sumcheck protocol on $f^{(i)}$ and sends the univariate polynomial to $\mathcal{P}_0$. After receiving all univariate polynomials from other machines, $\mathcal{P}_0$ aggregates these univariate polynomials by summing them together and sends the aggregated univariate polynomial to the verifier. When $\mathcal{P}_0$ receives a random query from the verifier, $\mathcal{P}_0$ relays the random challenge to all nodes as the random query of the current round.

2. In round $(\ell - n)$ (step 2 in Protocol 10), the polynomials of $f^{(0)}, \cdots, f^{(N-1)}$ have been condensed to one evaluation on a random vector $\mathbf{r} \in \mathbb{F}^{\ell-n}$. $\mathcal{P}_0$ uses these N points as an array to construct the multi-linear polynomial $f' : \mathbb{F}^n \to \mathbb{F}$ such that $f'(\mathbf{x}) = f(\mathbf{r}, \mathbf{x}[1 : n])$.[3]

3. After round $(\ell - n)$ (step 3 in Protocol 10), $\mathcal{P}_0$ continues to run the sumcheck protocol on f' with $\mathcal{V}$ in last n rounds.

In this way, the computation of $\mathcal{P}_i$ is equivalent to running the sumcheck protocol in Virgo on C_i. It accelerates the sumcheck protocol in Virgo by N times without any overhead on the proof size using N distributed

[3] The approach can extend to the product of two multi-linear polynomials, which matches the case in Virgo.

Protocol 10 (**Distributed sumcheck**). Suppose the prover has N machines $\mathcal{P}_0, \cdots, \mathcal{P}_{N-1}$ and suppose $\mathcal{P}_0$ is the master node. Each P_i holds a polynomial $f^{(i)} : \mathbb{F}^{\ell-n} \to \mathbb{F}$ such that $f^{(i)}(\mathbf{x}) = f(\mathbf{x}[1 : \ell - n], \mathbf{i})$. Suppose $\mathcal{V}$ is the verifier. The protocol proceeds in 3 phases consisting of ℓ rounds.

1. In the j-th round, where $1 \le j \le \ell - n$, each $\mathcal{P}_i$ sends P_0 a univariate polynomial

   $$f_j^{(i)}(x_j) \overset{def}{=} \sum_{\mathbf{b} \in \{0,1\}^{\ell-n-j}} f^{(i)}(\mathbf{r}[1 : j-1], x_j, \mathbf{b}),$$

   After receiving all univariate polynomials from $P_1, \cdots, P_{N-1}$, P_0 computes

   $$f_j(x_j) = \sum_{i=0}^{N-1} f_j^{(i)}(x_j)$$

   then sends $f_j(x_j)$ to $\mathcal{V}$. $\mathcal{V}$ checks $f_{j-1}(r_{j-1}) = f_j(0) + f_j(1)$, and sends a random challenge $r_j \in \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ relays r_j to $P_1, \cdots, P_{N-1}$.

2. In the j-th round, where $j = \ell - n$, after receiving r_j from $\mathcal{P}_0$, each $\mathcal{P}_i$ computes $f^{(i)}(\mathbf{r}[1 : j])$ and sends $f^{(i)}(\mathbf{r}[1 : j])$ to $\mathcal{P}_0$. Then $\mathcal{P}_0$ constructs a multi-linear polynomial $f' : \mathbb{F}^n \to \mathbb{F}$ such that $f'(\mathbf{i}) = f^{(i)}(\mathbf{r}[1 : j])$ for $0 \le i < N$.

3. In the j-th round, where $\ell - n < j \le \ell$, $\mathcal{P}_0$ and $\mathcal{V}$ run Protocol 8 on the statement:

   $$H' = \sum_{\mathbf{b} \in \{0,1\}^n} f'(\mathbf{b}),$$

   where $H' = \sum_{i=0}^{N-1} f^{(i)}(\mathbf{r}[1 : \ell - n])$.

machines, which is optimal for distributed algorithms both in asymptotic complexity and in practice. We give the complexity of Protocol 10 in the following.

Complexity of the distributed sumcheck protocol. For the multivariate polynomial of f defined in Equation 5.4, The total prover work is $O(2^\ell)$ while the prover work for each machine is $O(\frac{2^\ell}{N})$. The communication between N machines is $O(N\ell)$. The proof size and the verifier time are both $O(\ell)$.

5.4.3 Distributed polynomial commitment

In the last step of the sumcheck phase, the prover needs to prove to the verifier $y = f(r_1, \cdots, r_\ell)$ for some value y. In Virgo, The prover convinces $\mathcal{V}$ of the evaluation by invoking the PC scheme. We present the PC scheme in Virgo and the complexity of the scheme in the following.

Background: the polynomial commitment in Virgo. Let $\mathcal{F}$ be a family of ℓ-variate multi-linear polynomial over $\mathbb{F}$. Let $\mathbb{H}$, $\mathbb{L}$ be two disjoint multiplicative subgroups of $\mathbb{F}$ such that $|\mathbb{H}| = 2^{\ell}$ and $|\mathbb{L}| = \rho|\mathbb{H}|$, where ρ is a power of 2. The polynomial commitment (PC) in Virgo for $f \in \mathcal{F}$ and $\mathbf{r} \in \mathbb{F}^{\ell}$ consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{PC.KeyGen}(1^{\lambda})$: Given the security parameter λ, the algorithm samples a collision resistant hash function from a hash family as pp.
- $\mathsf{com}_f \leftarrow \mathsf{PC.Commit}(f, \mathsf{pp})$: Given a multi-linear polynomial f, the prover treats 2^{ℓ} coefficients of f as evaluations of a univariate polynomial f_U on $\mathbb{H}$. The prover uses the inverse fast Fourier transform (IFFT) to compute f_U. Then the prover computes $\mathbf{f}_{\mathbb{L}}$ as evaluations of f_U on $\mathbb{L}$ via the fast Fourier transform (FFT). Let $\mathsf{com}_f = \mathsf{MT.Commit}(\mathbf{f}_{\mathbb{L}})$.
- $(y, \pi_f) \leftarrow \mathsf{PC.Open}(f, \mathbf{r}, \mathsf{pp})$: The prover computes $y = f(\mathbf{r})$. Given $c = O(\lambda)$ random indexes $(k_1, \cdots, k_c)$, the prover computes $(\mathbf{f}_{\mathbb{L}}[k_1], \pi_{k_1}) = \mathsf{MT.Open}(\mathbf{f}_{\mathbb{L}}, k_1)$, $\cdots$, $(\mathbf{f}_{\mathbb{L}}[k_c], \pi_{k_c}) = \mathsf{MT.Open}(\mathbf{f}_{\mathbb{L}}, k_c)$. Let π_f = $(\mathbf{f}_{\mathbb{L}}[k_1], \pi_{k_1}, \cdots, \mathbf{f}_{\mathbb{L}}[k_c], \pi_{k_c})$.[4]
- $\{1, 0\} \leftarrow \mathsf{PC.Verify}(\mathsf{com}_f, \mathbf{r}, y, \pi_f, \mathsf{pp})$: The verifier parses $\pi_f = (\mathbf{q}_{\mathbb{L}}[k_1], \pi_{k_1}, \cdots, \mathbf{q}_{\mathbb{L}}[k_c], \pi_{k_c})$, then checks that $\mathbf{q}_{\mathbb{L}}[k_1]$, $\cdots$, $\mathbf{q}_{\mathbb{L}}[k_c]$ are consistent with y by a certain equation $p(f_{\mathbb{L}}[k_1], \cdots, f_{\mathbb{L}}[k_c], y) = 0$, [5] and checks that $\mathbf{f}_{\mathbb{L}}[k_1]$, $\cdots$, $\mathbf{f}_{\mathbb{L}}[k_c]$ are consistent with com_f by $\mathsf{MT.Verify}$ $(\pi_{k_1}, \mathbf{f}_{\mathbb{L}}[k_1], \mathsf{com}_f)$, $\cdots$, $\mathsf{MT.Verify}$ $(\pi_{k_c}, \mathbf{f}_{\mathbb{L}}[k_c], \mathsf{com}_f)$. If all checks pass, the verifier outputs $\mathsf{1}$, otherwise the verifier outputs $\mathsf{0}$.

Complexity of PC in Virgo. The prover time is $O(\ell \cdot 2^{\ell})$. The proof size is $O(\lambda \ell^2)$ and the verifier time is $O(\lambda \ell^2)$.

In the setting of distributed PC, $\mathcal{P}_i$ knows $f^{(i)}$. With the help of β function, we have

$$f(\mathbf{r}) = \sum_{i=0}^{N-1} \beta(\mathbf{r}[\ell - n + 1 : \ell], \mathbf{i}) f^{(i)}(\mathbf{r}[1 : \ell - n]). \tag{5.5}$$

A straightforward way for distributed PC is that $\mathcal{P}_i$ runs the PC scheme on $f^{(i)}$ separately. In particular, $\mathcal{P}_i$ invokes $\mathsf{PC.Commit}$ to commit $f^{(i)}$ in the beginning of the sumcheck protocol. In the last round, $\mathcal{P}_i$ runs $\mathsf{PC.Open}$ to compute $f^{(i)}(\mathbf{r}[1 : \ell - n])$ and sends the proof to $\mathcal{V}$. After receiving all $f^{(i)}(\mathbf{r}[1 : \ell - n])$ from $\mathcal{P}_i$, $\mathcal{V}$ invokes $\mathsf{PC.Verify}$ to validate N polynomial commitments separately. Then $\mathcal{V}$ computes $\beta(\mathbf{r}[\ell - n + 1 : \ell], \mathbf{i})$ for each i. Finally, $\mathcal{V}$ checks $f(\mathbf{r}) = \sum_{i=0}^{N-1} \beta(\mathbf{r}[\ell - n + 1 : \ell], \mathbf{i}) f^{(i)}(\mathbf{r}[1 : \ell - n])$.

Although the aforementioned naïve distributed protocol achieves $O(2^{\ell}(\ell - n))$ in computation time for each machine, the total proof size is $O(\lambda N (\ell - n)^2)$ as the individual proof size for each $\mathcal{P}_i$ is $O(\lambda(\ell - n)^2)$. To reduce the proof size, we optimize the algorithm by aggregating N commitments and N proofs altogether. For simplicity, we assume $\rho = 1$ without loss of generality in the multi-linear polynomial commitment[6]. We present the formal protocol of distributed PC in Protocol 11.

[4] The prover also computes $\log |\mathbb{L}|$ polynomials of $f_1, \cdots, f_{\log |\mathbb{L}|}$ depending on f. But sizes of these polynomials are $\frac{|\mathbb{L}|}{2}, \cdots, 1$ respectively. The prover commits these polynomial and opens them on at most c locations correspondingly. Our techniques on distributed commitment and opening can apply to these smaller polynomials easily. We omit the process for simplicity. It brings a logarithmic factor in the size of the polynomial on the proof size and the verification time.

[5] p also takes all openings on polynomials of $f_1, \cdots, f_{\log |\mathbb{L}|}$ (at most c for each polynomial) as input, we omit them for simplicity.

[6] In Virgo, $\rho = 32$ for security requirements. Our scheme can extend to $\rho = 32$ easily.

Protocol 11 (**Distributed PC**). Suppose the prover has N machines of $\mathcal{P}_0, \cdots, \mathcal{P}_{N-1}$ and suppose $\mathcal{P}_0$ is the master node. Each P_i holds a polynomial $f^{(i)} : \mathbb{F}^{\ell-n} \to \mathbb{F}$ such that $f(\mathbf{x}) = \beta(\mathbf{x}[\ell-n+1 : \ell], \mathbf{i}) f^{(i)}(x[1 : \ell - n])$. Suppose $\mathcal{V}$ is the verifier. Let $\mathbb{H}$ and $\mathbb{L}$ be two disjoint multiplicative subgroups of $\mathbb{F}$ such that $|\mathbb{H}| = \frac{2^\ell}{N}$ and $|\mathbb{L}| = \rho|\mathbb{H}|$. For simplicity, We assume $\rho = 1$. Let $\mathsf{pp} = \mathsf{PC.KeyGen}(1^\lambda)$. The protocol proceeds in following steps.

1. Each $\mathcal{P}_i$ invokes $\mathsf{PC.Commit}(f^{(i)}, \mathsf{pp})$ to compute $\mathbf{f}_{\mathbb{L}}^{(i)}$ by IFFT and FFT.
2. Each $\mathcal{P}_i$ sends $\mathbf{f}_{\mathbb{L}}^{(i)}[1], \cdots, \mathbf{f}_{\mathbb{L}}^{(i)}[N]$ to $\mathcal{P}_0, \cdots, \mathcal{P}_{N-1}$ separately.
3. Each $\mathcal{P}_i$ receives $\mathbf{f}_{\mathbb{L}}^{(0)}[i+1], \cdots, \mathbf{f}_{\mathbb{L}}^{(N-1)}[i+1]$ from other machines. Assuming $\mathbf{h}_{\mathbb{L}}^{(i)}$ = ($\mathbf{f}_{\mathbb{L}}^{(0)}[i+1], \cdots, \mathbf{f}_{\mathbb{L}}^{(N-1)}[i+1]$), $\mathcal{P}_i$ computes $\mathsf{com}_{h^{(i)}}$ = $\mathsf{MT.Commit}(\mathbf{h}_{\mathbb{L}}^{(i)})$ and sends $\mathsf{com}_{h^{(i)}}$ to $\mathcal{P}_0$.
4. Suppose $\mathbf{h}$ = ($\mathsf{com}_{h^{(0)}}, \cdots, \mathsf{com}_{h^{(N-1)}}$), $\mathcal{P}_0$ computes com = $\mathsf{MT.Commit}(\mathbf{h})$ and sends com to $\mathcal{V}$.
5. After receiving the random vector $\mathbf{r}$ from $\mathcal{V}$, $\mathcal{P}_0$ relays $\mathbf{r}$ to each $\mathcal{P}_i$. Each $\mathcal{P}_i$ computes $f^{(i)}(\mathbf{r}[1 : \ell - n])$ and sends it to $\mathcal{V}$ via $\mathcal{P}_0$.
6. To prove the correctness of $f^{(i)}(\mathbf{r}[1 : \ell - n])$, given random index of $k_1, \cdots, k_c$ from $\mathcal{V}$, $\mathcal{P}_{k_1-1}, \cdots, \mathcal{P}_{k_c-1}$ send $\mathbf{h}_{\mathbb{L}}^{(k_1-1)}, \cdots, \mathbf{h}_{\mathbb{L}}^{(k_c-1)}$ to $\mathcal{V}$ via $\mathcal{P}_0$.$\mathcal{P}_0$ also generates $(\mathbf{h}[k_1], \pi_{k_1})$ = $\mathsf{MT.Open}(\mathbf{h}, k_1), \cdots, (\mathbf{h}[k_c], \pi_{k_c})$ = $\mathsf{MT.Open}(\mathbf{h}, k_c)$ and send them to $\mathcal{V}$.
7. $\mathcal{V}$ checks $f(\mathbf{r}) = \sum_{i=0}^{N-1} \beta(\mathbf{r}[\ell - n + 1 : \ell], \mathbf{i}) f^{(i)}(\mathbf{r}[1 : \ell - n])$. $\mathcal{V}$ checks $\mathbf{h}[k_1]$=$\mathsf{MT.Commit}(\mathbf{h}_{\mathbb{L}}^{(k_1-1)}), \cdots, \mathbf{h}[k_c]$=$\mathsf{MT.Commit}(\mathbf{h}_{\mathbb{L}}^{(k_c-1)})$. Then $\mathcal{V}$ checks $\pi_{k_1}, \cdots, \pi_{k_c}$ by $\mathsf{MT.Verify}(\pi_{k_1}, \mathbf{h}[k_1], \mathsf{com}), \cdots, \mathsf{MT.Verify}(\pi_{k_c}, \mathbf{h}[k_c], , \mathsf{com})$. $\mathcal{V}$ also checks $q(\mathbf{f}_{\mathbb{L}}^{(i)}[k_1], \cdots, \mathbf{f}_{\mathbb{L}}^{(i)}[k_c], \mathbf{f}^{(i)}(\mathbf{r}[1 : \ell - n])) = 0$ for each i as shown in $\mathsf{PC.Verify}$. If all checks pass, $\mathcal{V}$ outputs $\mathsf{1}$, otherwise $\mathcal{V}$ outputs $\mathsf{0}$.

The idea of our scheme is that each $\mathcal{P}_i$ exchanges data with other machines immediately after computing $\mathbf{f}_{\mathbb{L}}^{(i)}$ instead of invoking $\mathsf{MT.Commit}$ on $\mathbf{f}_{\mathbb{L}}^{(i)}$ directly. The advantage of such arrangement is that the prover aggregates evaluation on the same index into one branch and can open them together by a single Merkle tree proof for this branch. As described in the polynomial commitment of Virgo, the prover needs to open $f_{\mathbb{L}}$ on some random indexes depending on $\mathbf{r}$ in $\mathsf{PC.Open}$. As $\mathbf{r}$ is identical to each $f^{(i)}$, the prover would open each $f_{\mathbb{L}}^{(i)}$ at same indexes. If the prover aggregates $f_{\mathbb{L}}^{(i)}$ by the indexes, she can open N values in one shot by providing only one Merkle tree path instead of naïvely providing N Merkle tree paths, which helps her to save the total proof size by a logarithmic factor in the size of the polynomial.

Specifically, $\mathcal{P}_i$ collects evaluations of $\mathbf{f}_{\mathbb{L}}^{(0)}[i+1], \cdots, \mathbf{f}_{\mathbb{L}}^{(N-1)}[i+1]$ with identical index of $(i+1)$ in $\mathbb{L}$ from other machines (step 1 and step 2). Then $\mathcal{P}_i$ invokes $\mathsf{MT.Commit}$ to get a commitment, $com_{h^{(i)}}$, for these values, and submits $com_{h^{(i)}}$ to $\mathcal{P}_0$ (step 3). $\mathcal{P}_0$ invokes $\mathsf{MT.Commit}$ on $com_{h^{(0)}}, \cdots, com_{h^{(N-1)}}$ to compute the aggregated commitment, com, and $\mathcal{P}_0$ sends com to $\mathcal{V}$ (step 4). In the $\mathsf{PC.Open}$ phase, given a random index k_j from $\mathcal{V}$, $\mathcal{P}_0$ retrieves $\mathbf{f}_{\mathbb{L}}^{(N-1)}[k_j], \cdots, \mathbf{f}_{\mathbb{L}}^{(N-1)}[k_j]$ from $\mathcal{P}_{k_j-1}$, computes $(\mathsf{com}_{h^{(k_j-1)}}, \pi_{k_j})$ = $\mathsf{MT.Open}$ (com, k_j), and sends these messages to $\mathcal{V}$ (step 5 and step 6). $\mathcal{V}$ can validate N evaluations by invoking $\mathsf{MT.Verify}$ only once (step 7). With this approach, we reduce the proof size to $O(\lambda(N+\ell^2))$. And the complexity of Protocol 11 is shown in the following.

Complexity of distributed PC. Given that f is a multi-linear polynomial with ℓ variables, the total communication among N machines is $O(2^\ell)$. The total prover work is $O(2^\ell \cdot \ell)$ while the prover work for each device is $(\frac{2^\ell}{N} \cdot \ell)$. The proof size is $O(\lambda(N+\ell^2))$. The verification cost is $O(\lambda(N+\ell^2))$.

5.4.4 Combining everything together

In this section, we combine the distributed sumcheck and the distributed PC altogether to build $\mathsf{deVirgo}$.

Background: The Virgo protocol. By combining the GKR protocol and the polynomial commitment in Section 5.4.3 We present the formal protocol of Virgo in Protocol 12 and the the complexity of Protocol 12 in the following[7].

Protocol 12 (**Virgo**). Let $\mathbb{F}$ be a finite field. Let C: $\mathbb{F}^m \to \mathbb{F}^k$ be a d-depth layered arithmetic circuit. $\mathcal{P}$ wants to convince that $\mathbf{1} = C(\mathbf{x}, \mathbf{w})$ where $\mathbf{x}$ and $\mathbf{w}$ are input and $\mathbf{1}$ is the output. Without loss of generality, assume m and k are both powers of 2 and we can pad them if not.

1. Set $\mathsf{pp} \leftarrow \mathsf{PC.KeyGen}(1^\lambda)$. $\mathcal{P}$ invokes $\mathsf{PC.Commit}(\tilde{V}_d, \mathsf{pp})$ to generate $\mathsf{com}_{\tilde{V}_d}$ and sends $\mathsf{com}_{\tilde{V}_d}$ to $\mathcal{V}$.
2. $\mathcal{P}$ and $\mathcal{V}$ run step 1-3 in Protocol 9.
3. At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$. $\mathcal{P}$ and $\mathcal{V}$ invoke $\mathsf{PC.Open}$ and $\mathsf{PC.Verify}$ on $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ with $\mathsf{com}_{\tilde{V}_d}$ and pp. If they are equal to $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ sent by $\mathcal{P}$, $\mathcal{V}$ outputs $\mathsf{1}$, otherwise $\mathcal{V}$ outputs $\mathsf{0}$.

Complexity of Virgo [ZXZS20]. Given a layered arithmetic circuit C with d layers and m inputs, Protocol 12 is a zero-knowledge proof protocol as defined in Definition 5.2.2 for the function computed by C. The prover time is $O(|C| + m \log m)$. The proof size is $O(d \log |C| + \lambda \log^2 m)$ and The verification time is also $O(d \log |C| + \lambda \log^2 m)$.

$\mathsf{deVirgo}$. For a data-parallel layered arithmetic circuit C with N copies and d layers, following the workflow of Virgo in Protocol 12, our distributed prover replaces d sumcheck schemes in Virgo by d distributed sumcheck schemes, and replaces the PC scheme in Virgo by our distributed PC scheme to generate the proof. We present the formal protocol of deVirgo in Protocol 13. And we have the theorem as follows.

Theorem 5.4.1. *Protocol 13 is an argument of knowledge satisfying the completeness and knowledge soundness in Definition 5.2.2 for the relation $C(\mathbf{x}, \mathbf{w}) = \mathbf{1}$, where C consists of N identical copies of $C_0, \cdots, C_{N-1}$.*

[7] Protocol 12 is a knowledge argument system rather than a zero-knowledge proof protocol as we actually use the knowledge argument system in our construction.

Protocol 13 (**Distributed Virgo**). Let $\mathbb{F}$ be a finite field. Let C: $\mathbb{F}^{mN} \to \mathbb{F}^k$ be a d-depth layered arithmetic circuit. Suppose C is also a data-parallel circuit with N identical copies. $\mathcal{P}$ is a prover with N distributed machines and wants to convince $\mathcal{V}$ that $\mathbf{1} = C(\mathbf{x}, \mathbf{w})$ where $\mathbf{x}$ and $\mathbf{w}$ are input, and $\mathbf{1}$ is the output. Without loss of generality, assume m, N, and k are powers of 2 and we can pad them if not.

1. Set $\mathsf{pp} \leftarrow \mathsf{PC.KeyGen}(1^\lambda)$. Define the multi-linear extension of array $(\mathbf{x}, \mathbf{w})$ as $\tilde{V}_d$. $\mathcal{P}$ invokes step 1-4 in Protocol 11 on $\tilde{V}_d$ to get $\mathsf{com}_{\tilde{V}_d}$ and sends $\mathsf{com}_{\tilde{V}_d}$ to $\mathcal{V}$.

2. Define the multi-linear extension of array $\mathbf{1}$ as $\tilde{V}_0$. $\mathcal{V}$ chooses a random $g \in \mathbb{F}^{s_0}$ and sends it to $\mathcal{P}$.

3. $\mathcal{P}$ and $\mathcal{V}$ run Protocol 10, the distributed sumcheck protocol, on

$$1 = \sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_1}} (\tilde{add}_1(\mathbf{g}^{(0)}, \mathbf{x}, \mathbf{y})(\tilde{V}_1(\mathbf{x}) + \tilde{V}_1(\mathbf{y})) + \tilde{mult}_1(\mathbf{g}^{(0)}, \mathbf{x}, \mathbf{y})\tilde{V}_1(\mathbf{x})\tilde{V}_1(\mathbf{y}))$$

   At the end of the protocol, $\mathcal{V}$ receives $\tilde{V}_1(\mathbf{u}^{(1)})$ and $\tilde{V}_1(\mathbf{v}^{(1)})$. $\mathcal{V}$ computes $\tilde{mult}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})$, $\tilde{add}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})$ and checks that $\tilde{add}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})$ $(\tilde{V}_1(\mathbf{u}^{(1)}) + \tilde{V}_1(\mathbf{v}^{(1)})) + \tilde{mult}_1(\mathbf{g}^{(0)}, \mathbf{u}^{(1)}, \mathbf{v}^{(1)})$ $\tilde{V}_1(\mathbf{u}^{(1)})\tilde{V}_1(\mathbf{v}^{(1)})$ equals to the last message of the sumcheck.

4. For $i = 1, ..., d-1$:

   - $\mathcal{V}$ randomly selects $\alpha_{i,1}, \alpha_{i,2} \in \mathbb{F}$ and sends them to $\mathcal{P}$.
   - $\mathcal{P}$ and $\mathcal{V}$ run Protocol 10, the distributed sumcheck protocol, on

$$\begin{aligned} &\alpha_{i,1}\tilde{V}_i(\mathbf{u}^{(i)}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v}^{(i)}) = \\ &\quad \sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} ((\alpha_{i,1}\tilde{add}_{i+1}\mathbf{u}^{(i)}, \mathbf{x}, \mathbf{y}) + \alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v}^{(i)}, \mathbf{x}, \mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &\quad +(\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u}^{(i)}, \mathbf{x}, \mathbf{y}) + \alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v}^{(i)}, \mathbf{x}, \mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y})) \end{aligned}$$

   - At the end of the sumcheck protocol, $\mathcal{P}$ sends $\mathcal{V}$ $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$.
   - $\mathcal{V}$ computes the right-hand side of the above equation by replacing $\mathbf{x}$ and $\mathbf{y}$ by $\mathbf{u}^{(i+1)}$ and $\mathbf{v}^{(i+1)}$ respectively. checks if it equals to the last message of the sumcheck. If all checks in the sumcheck pass, V uses $\tilde{V}_{i+1}(\mathbf{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\mathbf{v}^{(i+1)})$ to proceed to the $(i+1)$-th layer. Otherwise, $\mathcal{V}$ outputs 0 and aborts.

5. At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$. $\mathcal{P}$ invokes step 5-6 in Protocol 11 to open $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ while $\mathcal{V}$ invokes step 7 in Protocol 11 to validate $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$. If they are equal to $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ sent by $\mathcal{P}$, $\mathcal{V}$ outputs 1, otherwise $\mathcal{V}$ outputs 0.

Proof (sketch). **Completeness.** The completeness is straightforward.

Knowledge soundness. deVirgo generates the same proof as Virgo for d sumcheck protocols. So we only need to consider the knowledge soundness of distributed PC scheme. If the commitment of f is inconsistent with the opening of $f(\mathbf{r})$ in the distributed PC scheme, there must exist at least one $f^{(i)}(\mathbf{r}[1 : \ell - n])$ being inconsistent with the commitment f by Equation 5.5. Otherwise, when all $f^{(i)}(\mathbf{r}[1 : \ell - n])$ are consistent with the commitment of f, $f(\mathbf{r})$ must be consistent with the commitment of f. As shown in Protocol 11, com_f is equivalent to $com_{f^{(i)}}$ with additional dummy messages in each element of the vector in the Merkle tree commitment. It does not affect the soundness of the PC in Virgo in the random oracle model [ZXZS20; ZXHSZ22]. The verifier outputs 0 in the $\mathsf{PC.Verify}$ phase with the probability of $(1 - \mathsf{negl}(\lambda))$. Therefore, deVirgo still satisfies knowledge soundness.

The zero-knowledge property is not necessary as there is no private witness in the setting of zkbridge. However, we can achieve zero-knowledge for deVirgo by adding some hiding polynomials. Virgo uses the same method to achieve zero-knowledge. ▮

Additionally, Fiore and Nitulescu [FN16] introduced the notion of O-SNARK for SNARK over authenticated data such as cryptographic signatures. Protocol 13 is an O-SNARK for any oracle family, albeit in the random oracle model. To see this, Virgo relies on the construction of computationally sound proofs of Micali [Mic00] to achieve non-interactive proof and knowledge soundness in the random oracle model, which has been proven to be O-SNARK in [FN16]. Hence Virgo is an O-SNARK, and so is deVirgo because deVirgo also relies on the same model.

Protocol 13 achieves optimal linear scalability on data-parallel circuits without significant overhead on the proof size. In particular, our protocol accelerates Virgo by N times given N distributed machines. Additionally, the proof size in our scheme is reduced by a factor of N compared to the naïve solution of running each sub-copy of data-parallel circuits separately and generating N proofs. The complexity of Protocol 13 is shown in the following.

Complexity of distributed Virgo. Given a data-parallel layered arithmetic circuit C with N sub-copies, each having d layers and m inputs, the total prover work of Protocol 13 is $O(|C| + Nm \log m)$. The prover work for a single machine is $O(|C|/N + m \log m)$, and the total communication among machines is $O(Nm + Nd \log |C|)$. The proof size is $O(d \log |C| + \lambda(N + \log^2 m))$. The verification cost is $O(d \log |C| + \lambda(N + \log^2 m))$.

5.5 Reducing proof size and verifier time

Although $\mathsf{deVirgo}$ improves the prover time by orders of magnitude, we want to further reduce the cost of the verification time and the proof size. As mentioned in the above section, the circuit which validates over 100 signatures is giant due to non-compatible instructions on different curves across different blockchains. Additionally, Virgo's proof size, which is around 210KB for a circuit with 10 million gates, is large in practice. Thus we cannot post $\mathsf{deVirgo}$'s proof on-chain and validate the proof directly. Aiming at smaller proof size and simpler verification on-chain, we propose to further compress the proof by recursive proofs with two layers. Intuitively, for a large-scale statement $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$ in Definition 5.2.2, the prover generates the proof π_1 by a protocol with fast prover time in the first layer. If the length of π_1 is not as short as desired, then the prover can produce a shorter proof π_2 by invoking another protocol for $(\mathbf{x}, \pi_1) \in \mathcal{R}'$ in the second layer, where $\mathcal{R}'$ represents that π_1 is a valid proof for $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$. To shrink the proof size and simplify the verification as much as possible, we choose Groth16 as the second layer ZKP protocol since Groth16 has constant proof size and fast verification time. Moreover, the curve in Groth16 is natively supported by Ethereum, which is beneficial for saving on-chain cost on Ethereum. In our approach, the prover invokes deVirgo to generate π_1 on the initial circuit in the first layer. In the second layer, the prover invokes Groth16 to generate π_2 on the circuit implementing the verification algorithm of deVirgo where $|\pi_2| \ll |\pi_1|$. The prover only needs to submit π_2 on-chain for verification. The recursion helps cross-chain bridges to reduce gas cost on blockchains because of simple verification on the compatible curve. The security of recursive proofs relies on random oracle assumption, which can be instantiated by a cryptographic hash function in practice [COS20].

# of sigs	Total circuit size	Circuit size for GKR part	Circuit size for PC part
1	1.2×10^7 gates	8.4×10^6 gates	3.3×10^6 gates
4	1.2×10^7 gates	8.4×10^6 gates	4.0×10^6 gates
32	1.3×10^7 gates	8.4×10^6 gates	4.7×10^6 gates
128	1.4×10^7 gates	8.4×10^6 gates	5.4×10^6 gates

Table 5.1: The verification circuit size of deVirgo

Performance gains. We use the signature validation circuit for Cosmos [Cos] as an example to show concrete numbers of the verification circuit of deVirgo in Table 5.1. We record the size of the whole verification circuit in the 2^{nd} column, the size for the GKR part in the 3^{rd} column, and the size for the PC part in the 4^{th} column, as the number of signatures in data-parallel circuits increases from 1 to 128 in the 1^{st} column. The number of gates in the 2^{nd} column equals the sum of numbers of gates in the 3^{rd} column and the 4^{th} column. As shown in Table 5.1, although the data-parallel circuit size expands, the size for the sumcheck part in deVirgo's verification circuit does not change. That is because the verification for the GKR part is only based on the structure of the sub-circuit, which is identical among different copies. However, the size for the PC part in deVirgo's verification circuit up-scales sub-linearly in the number of copies due to the growth of the polynomial size. Even given 128 copies of the signature validation circuit, the bottleneck of deVirgo's verification circuit is the sumcheck part. Therefore, the recursive proof size and the recursive verification cost are independent of the number of signatures to validate in our instance. In addition, the prover time of Groth16 on the verification circuit of deVirgo is only 25% of the prover time of deVirgo in practice. Therefore, our recursive proof scheme reduces the on-chain proof verification cost from $\sim 8 \times 10^7$ gas (an estimation) to less than 2.3×10^5 gas.

5.6 Implementation and Evaluation

To demonstrate the practicality of zkBridge, we implement a prototype from Cosmos [Cos] (a PoS blockchain built on top of the Tendermint [Kwo14] protocol) to Ethereum, and from Ethereum to other EVM-compatible chains such as BSC. Supports for other blockchains can be similarly implemented with additional engineering effort, as long as they support light client protocols defined in Definition 5.2.1. In this section, we discuss implementation detail, its performance, as well as operational cost.

The bridge from Cosmos to Ethereum is realized with the full blown zkBridge protocol presented so far to achieve practical performance. In comparison, the direction from Ethereum to other EVM-compatible chains incurs much less overhead for proof generation and does not require deVirgo. Therefore, in what follows, we mainly focus on the direction from Cosmos to Ethereum.

5.6.1 Implementation details

The bridge from Cosmos to Ethereum consists of four components: a relayer that fetches Cosmos block headers and sends them to Ethereum (implemented in 300+ lines of Python), deVirgo (implemented in 10000+ lines of C++) for distributed proof generation, a handcrafted recursive verification circuit, and an updater contract on Ethereum (implemented in 600+ lines of Solidity). Our signature verification circuit is based on the optimized signature verification circuit [Edd]. However, we use Gnark instead of Circom as in [Edd] for better efficiency for proof generation.

Generating correctness proofs. Relay nodes submit Cosmos block headers to the updater contract on Ethereum along with correctness proofs, which proves that the block is properly signed by the Cosmos validator committee appointed by the previous block. (In Cosmos a hash of the validator committee members is included in the previous block.)

In Cosmos, each block header contains about 128 EdDSA signatures (on Curve25519), Merkle roots for transactions and states, along with other metadata, where 32 top signatures are required to achieve supermajority stakes. However, the most efficient curve supported by the Ethereum Virtual Machine (EVM) is BN254. To verify Cosmos digital signatures in EVM, one must simulate Curve25519 on curve BN254, which will lead to large circuits. Concretely, to verify a Cosmos block header (mainly, to verify about 32 signatures), we need about 64 million gates. We implement deVirgo (Section 5.4) and recursive verification (Section 5.5) to accelerate proof generation and verification.

Moreover, in practical deployment, multiple relayers can form a pipeline to increase the throughput. Looking ahead, based on the evaluation results, our implementation can handle 1 second block time in Cosmos with $120+$ capable relayers in the network.

For proof verification, we build an outer circuit that verifies Virgo proofs and use Gnark [Gna] to generate the final Groth16 proof that can be efficiently verified by the updater contract on Ethereum.

The updater contract. We implement the updater contract on Ethereum in Solidity that verifies Groth16 proofs and keeps a list of the Cosmos block headers in its persistent storage. The cost of verifying a Groth16 proof on-chain is less than $230K$ gas.

The updater contract exposes a simple API which takes block height as its input, and returns the corresponding block header. The receiver contracts can then use the block header to complete application-specific verification.

Batching. Instead of calling the updater contract on every new block header, we implemented *batching* where the updater contract stores Merkle roots of batches of B consecutive block headers. The prover will first collect B consecutive blocks, and then makes a unified proof for all B blocks. The updater contract will only need to verify one proof for the batch of B blocks. After the verification, the updater contract checks the difficulty, stores the block headers, and updates the light-client state. Storing one Merkle root every B blocks also reduces storage cost. Thus B can be set to balance user experience and cost: With a larger B, users need to wait longer, but the cost of running the system is lower.

We implement the aforementioned batched proof verification and show the experimental results in Section 5.6.2. With batching, the cost for storing block headers and maintaining light-client states is amortized across B blocks. The bulk of the cost incurred by the updater contract is SNARK proof verification, which is the focus of our evaluation below.

In addition, we propose a more complex batching optimization presented in the following for further optimization.

On-chain Gas Cost Optimization To further optimize the on-chain gas cost of block header verification and storage for a universal zkBridge, we propose the following approach, in which the prover will not bother to pay for on-chain proof verification or block header storage, and users are encouraged to submit the proof they need by our incentive design.

In our optimization, the same as the aforementioned batched proof, the prover generates one single proof for every 2^d blocks where d is a system configuration, and each proof checks and shows the validity of all signatures in the corresponding 2^d blocks. However, instead of submitting the Merkle root of the batch along with the proof on-chain immediately, provers simply post the proof to the users (e.g., through a website), and it's up to the users to retrieve and post the proof on-chain. Thus there's no more on-chain gas cost for provers through the approach.

For users who want to verify a transaction tx in a block blk, the workflow is as follows.

1. If blk has already been submitted on-chain, go to the next step. Otherwise, retrieve the proofs for the sequence of blocks from the first unsubmitted one to blk, and then invoke the updater contract to verify all the proofs on-chain and store the information of the corresponding sequence of blocks. The process can be expensive. However, once the proofs are verified and the blocks are confirmed by the updater contract, the user becomes the owner of all these proofs on-chain, and can benefit from the proofs by charging later users who rely on these proofs to verify their transactions on-chain.

2. Thanks to the previously submitted proofs, the validity of the corresponding block is already proved at this step. And the work can never be accomplished without the efforts of proving all the blocks prior to blk (including blk). Suppose blk is the i^{th} block, then for each block with index in the range $[i - t + 1, i]$, the user should pay a certain amount of fee to the block proof owner in compensation, where t is a system configuration and the definition of block proof owner is defined in the previous step.

In this case, provers don't bother to pay for on-chain verification any more, and the proofs are only submitted and verified on demand, which is more cost-efficient and can reduce possible waste. Moreover, through carefully-designed incentive, we can actually encourage users to submit the proofs as a possible investment, and it can also help with the popularity of our bridge.

Through the optimization, the cost performance of our bridge can be summarized as follows. If there is high demand, then each proof will be submitted immediately upon generation, and in this case each user needs to pay for at most one time of on-chain proof verification. It then degenerates into our original batched proof, but users are responsible of paying for the on-chain verification instead. If the sender chain is so unpopular that there is little bridging demand from the chain, then we successfully avoid unnecessarily submitting the proofs on-chain for meaningless but costly verification. And even if a user suddenly exists and requires bridging in this case, the request can also be fulfilled by retrieving the proofs from provers and sending them for on-chain verification one by one.

And thus we can see that, the new design can actually benefit both the provers and the users.

5.6.2 Evaluation

We evaluate the performance of zkBridge (from Cosmos to Ethereum) from four aspects: proof generation time, proof generation communication cost, proof size, and on-chain verification cost.

Experiment setup. We envision that a relayer node in zkBridge will be deployed as a service in a managed network, therefore we evaluate zkBridge in a data-center-like environment. Specifically, we run all the experiments on 128 AWS EC2 c5.24xlarge instances with the Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz and 192GB of RAM. Our implementation for the proof generation is parallelized with at most 128 machines. We report the average running time of 10 executions. Whenever applicable, we report costs both in terms of running time and monetary expenses.

	Proof Gen. Time (seconds)			**Proof Gen. Comm. (GB)**		**Proof Size (Bytes)**		**On-chain Ver. Cost (gas)**	
# of sigs	deVirgo	RV	total	total	per-machine	w/o RV	w/ RV	w/o RV	w/ RV
8	12.52	4.90	17.42	7.34	0.92	1946476	131	78M	227K
32	12.80	5.41	18.21	32.24	1.01	1952492	131	78M	227K
128	13.28	5.49	18.77	131.89	1.03	1958508	131	79M	227K

Table 5.2: Evaluation results. RV is the shorthand for recursive verification.

Proof generation time of deVirgo. We first evaluate the main cryptographic building block—deVirgo—and compare its performance with the original Virgo [ZXZS20]. The source code of the original Virgo is obtained at `https://github.com/sunblaze-ucb/Virgo`. We run both protocols on the same circuit for correctness proofs, which mainly consists of N invocation of EdDSA signature verification.

Figure 5.2 shows the prover time (in seconds) against different N. For deVirgo, we repeat the experiment with 8, 32, 128 distributed machines. According to Fig. 5.2, the prover time of the original Virgo increases linearly in the number of signatures N, while the prover time of deVirgo is almost independent of N until N is greater than the number of servers when computation becomes an bottleneck. The linear scalability suggests that the workload of each machine only depends on its own sub-circuit and the communication overhead is small. Table 5.2 reports the communication cost among parallel machines. The total communication cost is linear in the number of machines, consistent with the analysis in Section 5.4.4, with each machine sending and receiving around 1 GB of data. Since we envision a relayer node in zkBridge to be deployed in a data-center-like environment, the amount of traffic is reasonable.

In practice, the Cosmos block headers typically have $N = 128$ signatures while 32 top signatures are sufficient to achieve super-majority. Therefore, generating a correctness proof for a Cosmos block header would take more than 400 seconds with the original Virgo, but it decreases to 13.28 seconds with deVirgo, implying a 30x speedup. In general, as is consistent with the analysis in Section 5.4, deVirgo accelerates the proof generation on data-parallel circuits with N copies by a factor of almost N, which is optimal for distributed algorithms.

Proof size and verification time.

To reduce on-chain verification cost, we use the recursive verification technique presented in Section 5.5. Now we report on its efficacy.

Recursive proof generation time. We implement recursive verification by invoking Groth16 (constructed using gnark [Gna]) on the verification circuit. We report the proof time in deVirgo, the generation time of

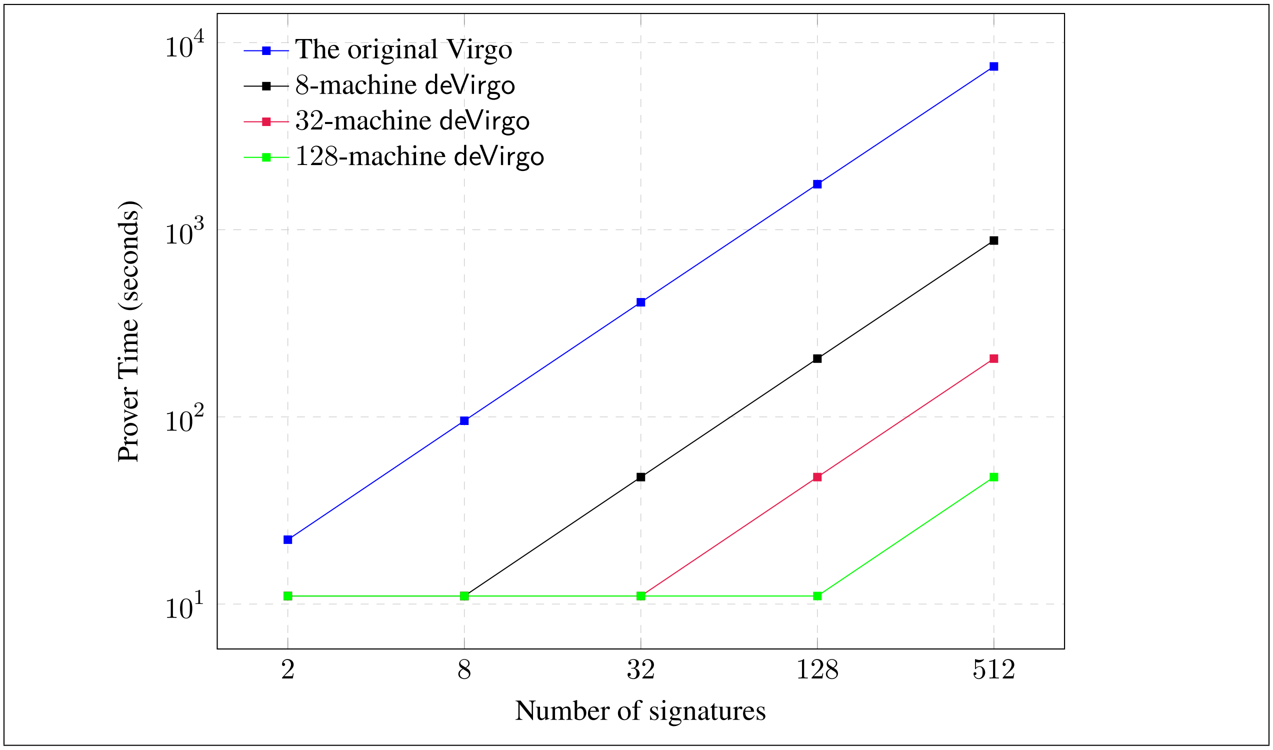


Figure 5.2: Prover time of deVirgo and the original Virgo for Cosmos block header verification.

recursive proofs (the column marked RV), and the sum, in Table 5.2, for various numbers of signatures. The RV time almost remains constant in the number of signatures verified by the deVirgo proofs. That is because of the data-parallel structure of the state transition proof circuit: the size of Groth16 verification circuit is only a function of the size of a sub-circuit.

The main benefit of recursive verification is a reduction in both proof size and verification cost.

Reduced proof size. Table 5.2 shows the proof size both with and without recursive verification. For the practical scenario where $N = 32$, the proof size is reduced from 1.9 MB to 131 Bytes. Overall, for $N = 32$, with an increase of about 25% in prover time, we get a reduction of around 14000x in proof size.

Reduced on-chain verification cost. The final proof is 131 Bytes while the final verification only costs 3 pairings. As shown in Table 5.2, the on-chain verification cost is constant (227K). In comparison, without recursive verification, directly verifying Virgo proofs on-chain would be infeasible. (Our estimation of the gas cost is 78M, which far exceeds the single block gas limit 30M).

Comparison with optimistic bridges. With batching, the confirmation latency of zkBridge is under 2 minutes, including 3×32 seconds for waiting for all blocks in the batch and another 20 seconds for proof generation. While this is not blazing fast, in comparison, optimistic bridges have much longer confirmation time. E.g., NEAR's Rainbow bridge has a challenge window of 4 hours [Nea] before which the transfer cannot be confirmed.

5.6.3 Cost analysis

In this section, we analyze the operational cost of zkBridge, which consists of off-chain cost (generating proofs) and on-chain cost (storing headers and verifying proofs).

Off-chain cost. Off-chain cost can vary significantly based on the deployment. While we use AWS in our performance benchmark, it may not be the best option for practical deployment. AWS service is expensive due to its high margin, elastic scaling capability, and high reliability, which isn't necessary for our proof generation process. To show a representative range, we consider two deployment options: cloud-based and self-hosted. For cloud-based deployment, we search for reputable and economical dedicated server rental services and choose Hetzner[Het] as an example. For self-hosted options, we calculate the cost to purchase the hardware and the on-going cost (mainly the electricity).

On AWS c5.24xlarge, it takes 18 seconds to generate a proof with 32 machines. Renting a server with a similar spec as AWS c5.24xlarge from Hetzner costs $253.12 per month, thus the cost of cloud-based deployment with Hetzner will be around $8100 per month for all 32 machines. It translates to $0.02 per block.

To estimate the cost for self-hosted deployment, we use online tools to configure a machine with a comparable spec to that in AWS. Table 5.3 reports the configuration and each machine costs around $4.5k. The total setup cost is thus around $4.5k $\times 32 =$ $144k. For self-hosted servers, the main on-going cost is electricity. With each machine consuming 657W power, a 32-machine cluster consumes 0.105 kWh per block. Assuming US average electricity rate $0.12/KWh [Use], the electricity cost is $0.012 per block, or $5184 per month.

Hardware type	Hardware name	Power	Price	Quantity
CPU	AMD Ryzen Threadripper 3970X	435W	$2325.99	1
Memory	CMK256GX4M8D3600C18	96W	$1129.99	1
Motherboard	MSI TRX40 PRO WIFI	80W	$565.57	1
Power Supply	EVGA 220-T2-1000-X1	94% efficiency	$332.88	1
SSD	MZ-V8P1T0B/AM	6.2W	$129.99	1
Total		657W	$4484.42	

Table 5.3: Prover hardware configuration.

On-chain cost. On-chain cost refers to the total gas used for on-chain operation, and we report the equivalent USD cost based on the gas price (about 20 gwei) and ETH price (about 1600 USD) at the time of writing (August 2022). If we use efficient batched proofs, for a batch of N headers, the bulk of the verification cost is that of verifying one Groth16 proof, which costs less than $230K$ gas, roughly $7.36. If we choose $N = 32$ for example, the on-chain cost will be $0.23 per block. Moreover, if we adopt the optimization mentioned in Section 5.6.1, we can further reduce the on-chain cost and offload the cost to users if the number of users is large.

5.6.4 Ethereum to other EVM-compatible chains

So far we have focused on the bridge from Cosmos to Ethereum because generating and verifying correctness proofs for that direction is challenging. We also implement a prototype of a bridge from Ethereum to other EVM-compatible blockchains.

The high level idea is simple: upon receiving a block header, the updater contract on the receiver chain verifies the PoW and appends it to the list of headers if the verification is passed. However, a wrinkle to the implementation is that Ethereum uses a memory hard hash function, EthHash [Woo+14], which is prohibitively inefficient to run on-chain. Basically, EthHash involves randomly accessing elements in a 1 gigabyte dataset (called a DAG) derived from a public seed and the block height. Generating the DAGs on-chain is prohibitively expensive.

Our idea is to pre-compute many DAGs off-chain and store their hashes on-chain. Specifically, as part of zkBridge setup, we pre-compute 2,048 DAGs , build a Merkle tree for each DAG using MiMC [AGRRT16], and store the Merkle roots on-chain. Per EthHash specification, a new DAG is generated every 30,000 blocks, so 2,048 of them can last for 10 years; the off-chain pre-computation process takes no more than 4 days. Then, the correctness proofs will show that a given EthHash PoW is correct with respect to the Merkle root of the DAG corresponding to the block in question. We emphasize that the setup process is verifiable and anyone can verify the published Merkle roots on their own before using the service. The circuit for verifying EthHash PoW has around 2 million gates.

The rest of the protocol is the same as a regular light client, which involves storing the headers, following the longest chain by computing accumulated difficulty, resolving forks, etc.

Cost analysis. Since EthHash PoW verification circuit has only around 2 million constraints, a single machine with the configuration in Table 5.3 can generate a proof within 10 seconds. As long as the receiver chain is EVM-compatible, the on-chain cost will be close to that presented in Section 5.6.3, since the updater contract only verifies Groth16 proofs in all cases.

5.7 Related work

In this section, we compare zkBridge to existing cross-chain bridge systems and the line of work on zk-rollups which also uses ZKPs for scalability and security.

Cross-chain bridges in the wild and security issues. Cross-chain systems are widely deployed and used. Below we briefly survey the representative ones. The list is not meant to be exhaustive. PolyNetwork [Polb] is an interoperability protocol using a side-chain as the relay with a two-phase commitment protocol. Wormhole [Worb] is a generic message-passing protocol secured by a network of guardian nodes, and its security relies on $\frac{2}{3}$ of the committee being honest. Ronin operates in a similar model. While relying on decentralized committees for security, practical deployment usually opts for relatively small ones for efficiency (e.g., 9 in case of Ronin). Committee breaches are far from being rare in practice. In a recent exploit against Ronin [Ron], the attacker obtained five of the nine validator keys, stealing 624 million USD. PolyNetwork and Wormhole were also recently attacked, losing \$611m [Pola] and \$326m [Wora] respectively. Key compromise was suspected in the PolyNetwork attack.

An alternative design is to leverage economic incentives. Nomad [Nomb] (which recently lost more than \$190m to hackers due to an implementation bug [Noma]) and Near's Rainbow Bridge [Rai] are such examples. These systems require participants to deposit a collateral, and rely on a watchdog service to continuously monitor the blockchain and confiscate offenders' collateral upon detecting invalid updates. Optimistic protocols fundamentally require a long confirmation latency in order to ensure invalid updates can be detected with high probability (e.g., Near [Rai] requires 4 hours). Moreover, participants must deposit significantly collateral (e.g., 20 ETH in Near [Rai]). Both issues can be avoided by zkBridge.

In summary, compared to existing protocols, zkBridge achieve both efficiency and cryptographic assurance. zkBridge is "trustless" in that it does not require extra assumptions other than those of blockchains and underlying cryptographic protocols. It also avoids the long confirmation of optimistic protocols.

zk-rollups. Rollups are protocols that batch transaction execution using ZKPs to scale up the layer-1 blockchains. Starkware [Sta], ZkSync [Zks], and Polygon Zero [Pole] are a few examples.

These zk-rollup solutions have not been applied to the bridge setting, where our work is the first to use ZKP to enable a decentralized trustless bridge. In addition, the current zk-rollup work in general has not dealt with such large circuits as in zkBridge, whereas in our work, we need to design and develop a number of techniques including deVirgo and proof recursion to make building a ZKP-based bridge practical for the first time. In particular, we leverage the data parallelism of the circuits to obtain a ZKP protocol that is more than 100x faster than existing protocols for the workload in zkBridge and combine it with proof recursion for efficient on-chain verification. The idea behind deVirgo protocol may be applicable to zk-rollups too.

Bibliography

[ABFG14] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. “Proofs of space: When space is of the essence”. In: *International Conference on Security and Cryptography for Networks*. Springer. 2014, pp. 538–557.

[ABST22] M. Ambrona, M. Beunardeau, A.-L. Schmitt, and R. R. Toledo. *aPlonK : Aggregated PlonK from Multi-Polynomial Commitment Schemes*. Cryptology ePrint Archive, Paper 2022/1352. `https://eprint.iacr.org/2022/1352`. 2022. URL: `https://eprint.iacr.org/2022/1352`.

[AGRRT16] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. “MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 191–219.

[AHIV17] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. “Ligero: Lightweight sublinear arguments without a trusted setup”. In: *CCS*. 2017.

[Amu] *A multichain approach is the future of the blockchain industry*. 2022. URL: `https://cointelegraph.com/news/a-multichain-approach-is-the-future-of-the-blockchain-industry` (visited on 04/24/2022).

[arm] armfazh. *flo-shani-aesni*. `https://github.com/armfazh/flo-shani-aesni`.

[Ate] *Ate-pairing*. `https://github.com/herumi/ate-pairing`.

[Aur] *libIOP*. `https://github.com/scipr-lab/libiop`.

[Axe] *Axelar*. `https://axelar.network/`. 2022.

[Azt] *Aztec*. `https://aztec.network/`. 2022.

[BBBPWM18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. “Bulletproofs: Short Proofs for Confidential Transactions and More”. In: *IEEE S&P*. Vol. 00. 2018, pp. 319–338.

[BBCDPGL18] C. Baum, J. Bootle, A. Cerulli, R. Del Pino, J. Groth, and V. Lyubashevsky. “Sub-linear Lattice-Based Zero-Knowledge Arguments for Arithmetic Circuits”. In: *CRYPTO*. Springer. 2018, pp. 669–699.

[BCCGP16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. “Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. 2016.

[BCCT13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. “Recursive Composition and Bootstrapping for SNARKS and Proof-Carrying Data”. In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*. STOC ’13. Palo Alto, California, USA: Association for Computing Machinery, 2013, pp. 111–120. ISBN: 9781450320290. DOI: `10.1145/2488608.2488623`. URL: `https://doi.org/10.1145/2488608.2488623`.

[BCG20] J. Bootle, A. Chiesa, and J. Groth. “Linear-Time Arguments with Sublinear Verification from Tensor Codes”. In: *TCC* (2020).

[BCGGHJ17] J. Bootle, A. Cerulli, E. Ghadafi, J. Groth, M. Hajiabadi, and S. K. Jakobsen. “Linear-time zero-knowledge proofs for arithmetic circuit satisfiability”. In: *ASIACRYPT*. Springer. 2017, pp. 336–365.

[BCL22] J. Bootle, A. Chiesa, and S. Liu. *Zero-Knowledge IOPs with Linear-Time Prover and Polylogarithmic-Time Verifier*. EUROCRYPT. 2022.

[BCLMS20] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. *Proof-Carrying Data without Succinct Arguments*. Cryptology ePrint Archive, Paper 2020/1618. `https://eprint.iacr.org/2020/1618`. 2020. URL: `https://eprint.iacr.org/2020/1618`.

[BCMS20] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. *Proof-Carrying Data from Accumulation Schemes*. Cryptology ePrint Archive, Paper 2020/499. `https://eprint.iacr.org/2020/499`. 2020. URL: `https://eprint.iacr.org/2020/499`.

[BDFG20] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. *Halo Infinite: Recursive zk-SNARKs from any Additive Polynomial Commitment Scheme*. Cryptology ePrint Archive, 2020/1536. 2020.

[BDLSY12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. “High-speed high-security signatures”. In: *Journal of cryptographic engineering* 2.2 (2012), pp. 77–89.

[Bee] *Beeple sold an NFT for $69 million - The Verge*. 2022-04-24. URL: `https://www.theverge.com/2021/3/11/22325054/beeple-christies-nft-sale-cost-everydays-69-million`.

[BEGKN94] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. “Checking the correctness of memories”. In: *Algorithmica* 12.2-3 (1994), pp. 225–244.

[Ben+14] E. Ben-Sasson et al. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *IEEE S&P*. 2014.

[BFHVXZ20] R. Bhadauria, Z. Fang, C. Hazay, M. Venkitasubramaniam, T. Xie, and Y. Zhang. “Ligero++: A New Optimized Sublinear IOP”. In: *CCS*. 2020.

[BFRSBW13] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. “Verifying computations with state”. In: *SOSP*. 2013.

[BFS20] B. Bünz, B. Fisch, and A. Szepieniec. “Transparent SNARKs from DARK compilers”. In: *Eurocrypt*. `https://eprint.iacr.org/2019/1229`. 2020.

[BG12] S. Bayer and J. Groth. “Efficient zero-knowledge argument for correctness of a shuffle”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2012, pp. 263–280.

[BLMR14] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld. “Proof of activity: Extending bitcoin’s proof of work via proof of stake [extended abstract] y”. In: *ACM SIGMETRICS Performance Evaluation Review* 42.3 (2014), pp. 34–37.

[BLNS20] J. Bootle, V. Lyubashevsky, N. K. Nguyen, and G. Seiler. “A non-PCP approach to succinct quantum-safe zero-knowledge”. In: *CRYPTO*. 2020.

[BMRS21] C. Baum, A. J. Malozemoff, M. Rosen, and P. Scholl. “Mac’n’Cheese: Zero-Knowledge Proofs for Arithmetic Circuits with Nested Disjunctions”. In: *CRYPTO*. 2021.

[BMV19] B. Bünz, M. Maller, and N. Vesely. “Efficient Proofs for Pairing-Based Languages”. In: 2019.

[BPS16] I. Bentov, R. Pass, and E. Shi. “Snow White: Provably Secure Proofs of Stake.” In: *IACR Cryptol. ePrint Arch.* 2016.919 (2016).

[BSBHR18] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. “Fast reed-solomon interactive oracle proofs of proximity”. In: *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[BSBHR19] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. “Scalable zero knowledge with no trusted setup”. In: *CRYPTO*. Springer. 2019, pp. 701–732.

[BSCGTV13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. “SNARKs for C: Verifying program executions succinctly and in zero knowledge”. In: *CRYPTO*. 2013.

[BSCRSVW19] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. “Aurora: Transparent succinct arguments for R1CS”. In: *Eurocrypt*. Springer. 2019, pp. 103–128.

[BSCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. “Interactive oracle proofs”. In: *Theory of Cryptography Conference*. Springer. 2016, pp. 31–60.

[BSCTV14a] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Scalable Zero Knowledge via Cycles of Elliptic Curves”. In: *CRYPTO*. 2014, pp. 276–294.

[BSCTV14b] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. *Scalable Zero Knowledge via Cycles of Elliptic Curves*. Cryptology ePrint Archive, Paper 2014/595. https://eprint.iacr.org/2014/595. 2014. URL: https://eprint.iacr.org/2014/595.

[BSCTV14c] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture”. In: *Proceedings of the USENIX Security Symposium*. 2014.

[But] *Vbuterin comments on [AMA] We are the EF’s Research Team (Pt. 7: 07 January, 2022)*. 2022. URL: https://old.reddit.com/r/ethereum/comments/rwojtk/ama_we_are_the_efs_research_team_pt_7_07_january/hrngyk8/ (visited on 04/24/2022).

[CBC21] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias. “SoK: Blockchain Light Clients”. In: *Cryptology ePrint Archive* (2021).

[CCHLRR18] R. Canetti, Y. Chen, J. Holmgren, A. Lombardi, G. N. Rothblum, and R. D. Rothblum. *Fiat-Shamir From Simpler Assumptions*. Cryptology ePrint Archive, Report 2018/1004. 2018.

[CD98] R. Cramer and I. Damgård. “Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free?” In: *Annual International Cryptology Conference (CRYPTO)*. 1998.

[CFS17] A. Chiesa, M. A. Forbes, and N. Spooner. “A Zero Knowledge Sumcheck and its Applications”. In: *CoRR* abs/1704.02086 (2017). arXiv: `1704.02086`. URL: `http://arxiv.org/abs/1704.02086`.

[Cha+17] M. Chase et al. “Post-quantum zero-knowledge and signatures from symmetric-key primitives”. In: *CCS*. ACM. 2017, pp. 1825–1842.

[CHMMVW20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. “Marlin: Preprocessing zksnarks with universal and updatable srs”. In: *Eurocrypt*. 2020, pp. 738–768.

[Cira] *Circom*. `https://github.com/iden3/circom`. 2022.

[Cirb] *ed25519-circom*. `https://github.com/Electron-Labs/ed25519-circom`. 2022.

[CMT12] G. Cormode, M. Mitzenmacher, and J. Thaler. “Practical Verified Computation with Streaming Interactive Proofs”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS ’12. 2012.

[Coi] *Cryptocurrency prices, charts and market capitalizations*. 2022. URL: `https://coinmarketcap.com/`.

[Cos] *Cosmos*. `https://cosmos.network/`. 2022.

[Cos+15] C. Costello et al. “Geppetto: Versatile verifiable computation”. In: *IEEE S&P*. 2015.

[COS20] A. Chiesa, D. Ojha, and N. Spooner. “Fractal: Post-quantum and transparent recursive proofs from holography”. In: *Eurocrypt*. `https://eprint.iacr.org/2019/1076`. 2020, pp. 769–793.

[CRVW02] M. Capalbo, O. Reingold, S. Vadhan, and A. Wigderson. “Randomness Conductors and Constant-Degree Lossless Expanders”. In: STOC. 2002.

[CS07] A. Czumaj and C. Sohler. “Testing Expansion in Bounded-Degree Graphs”. In: *IEEE FOCS*. 2007.

[CT10] A. Chiesa and E. Tromer. “Proof-Carrying Data and Hearsay Arguments from Signature Cards”. In: *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 5-7, 2010. Proceedings*. Ed. by A. C. Yao. Tsinghua University Press, 2010, pp. 310–331. URL: `http://conference.iiis.tsinghua.edu.cn/ICS2010/content/papers/25.html`.

[DFKP15] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. “Proofs of space”. In: *Annual Cryptology Conference*. Springer. 2015, pp. 585–605.

[DGKR17] B. David, P. Ga, A. Kiayias, and A. Russell. “Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol”. In: *Cryptology ePrint Archive* (2017).

[DI14] E. Druk and Y. Ishai. “Linear-Time Encodable Codes Meeting the Gilbert-Varshamov Bound and Their Cryptographic Applications”. In: *ITCS*. 2014.

[Din70] E. A. Dinic. “Algorithm for solution of a problem of maximum flow in networks with power estimation”. In: *Soviet Math. Doklady*. 1970.

[DIO21] S. Dittmer, Y. Ishai, and R. Ostrovsky. “Line-point zero knowledge and its applications”. In: *ITC*. 2021.

[Edd] *ed25519-circom*. `https://github.com/Electron-Labs/ed25519-circom`. 2022.

[EFG22] L. Eagen, D. Fiore, and A. Gabizon. *cq: Cached quotients for fast lookups*. Cryptology ePrint Archive, Paper 2022/1763. `https://eprint.iacr.org/2022/1763`. 2022. URL: `https://eprint.iacr.org/2022/1763`.

[ESLL19] M. F. Esgin, R. Steinfeld, J. K. Liu, and D. Liu. “Lattice-based zero-knowledge proofs: new techniques for shorter and faster constructions and applications”. In: *CRYPTO*. 2019.

[FDNZ21] Z. Fang, D. Darais, J. Near, and Y. Zhang. “Zero Knowledge Static Program Analysis”. In: *CCS*. 2021.

[FFGKOP16] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. “Hash first, argue later: Adaptive verifiable computations on outsourced data”. In: *CCS*. 2016.

[Fil] *Filecoin: A Decentralized Storage Network*. 2014. URL: `https://filecoin.io/filecoin.pdf`.

[FN16] D. Fiore and A. Nitulescu. “On the (in) security of SNARKs in the presence of oracles”. In: *Theory of Cryptography Conference*. Springer. 2016, pp. 108–138.

[FQZDC21] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu. *ZEN: Efficient Zero-Knowledge Proofs for Neural Networks*. Cryptology ePrint Archive, Report 2021/087. 2021.

[FS86] A. Fiat and A. Shamir. “How to Prove Yourself: Practical Solutions to Identification and Signature Problems”. In: *CRYPTO*. 1986.

[GAZBW22] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish. “Zero-Knowledge Middleboxes”. In: *USENIX Security*. 2022.

[GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. “Quadratic Span Programs and Succinct NIZKs without PCPs”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*. 2013, pp. 626–645.

[GHMVZ17] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. “Algorand: Scaling byzantine agreements for cryptocurrencies”. In: *Proceedings of the 26th symposium on operating systems principles*. 2017, pp. 51–68.

[Gil52] E. N. Gilbert. “A comparison of signalling alphabets”. In: *The Bell system technical journal* 31.3 (1952), pp. 504–522.

[GKMMM18] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. “Updatable and universal common reference strings with applications to zk-SNARKS”. In: *CRYPTO*. Springer. 2018, pp. 698–728.

[GKR08] S. Goldwasser, Y. T. Kalai, and G. Rothblum. “Delegating computation: interactive proofs for muggles”. In: *STOC*. 2008, pp. 113–122.

[GKR15] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. “Delegating Computation: Interactive Proofs for Muggles”. In: *J. ACM* 62.4 (Sept. 2015), 27:1–27:64. ISSN: 0004-5411.

[GLSTW] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby. *Brakedown: Linear-time and post-quantum SNARKs for R1CS*. Cryptology ePrint Archive. https://ia.cr/2021/1043.

[GMO16] I. Giacomelli, J. Madsen, and C. Orlandi. "ZKBoo: Faster Zero-Knowledge for Boolean Circuits." In: *USENIX Security*. 2016, pp. 1069–1083.

[GMR] S Goldwasser, S Micali, and C Rackoff. "The Knowledge Complexity of Interactive Proof-systems". In: *STOC 1985*, pp. 291–304.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. "The knowledge complexity of interactive proof systems". In: *SIAM Journal on computing* 18.1 (1989), pp. 186–208.

[Gna] *gnark*. https://docs.gnark.consensys.net/en/latest/. 2022.

[Gnu] *The GNU multiple precision arithmetic library*. https://gmplib.org/.

[Gol84] A. V. Goldberg. *Finding a maximum density subgraph*. University of California Berkeley, 1984.

[GR11] O. Goldreich and D. Ron. "On Testing Expansion in Bounded-Degree Graphs". In: *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation: In Collaboration with Lidor Avigad, Mihir Bellare, Zvika Brakerski, Shafi Goldwasser, Shai Halevi, Tali Kaufman, Leonid Levin, Noam Nisan, Dana Ron, Madhu Sudan, Luca Trevisan, Salil Vadhan, Avi Wigderson, David Zuckerman*. Ed. by O. Goldreich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 68–75. ISBN: 978-3-642-22670-0. DOI: 10.1007/978-3-642-22670-0_9. URL: https://doi.org/10.1007/978-3-642-22670-0_9.

[Gro09] J. Groth. "Linear algebra with sub-linear zero-knowledge arguments". In: *Annual International Cryptology Conference (CRYPTO)*. Springer, 2009, pp. 192–208.

[Gro10] J. Groth. "Short pairing-based non-interactive zero-knowledge arguments". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2010, pp. 321–340.

[Gro16] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *EUROCRYPT 2016*. 2016, pp. 305–326.

[GWC19a] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. 2019.

[GWC19b] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge". In: *Cryptology ePrint Archive* (2019).

[Hab22] U. Haböck. *Multivariate lookups based on logarithmic derivatives*. Cryptology ePrint Archive, Paper 2022/1530. https://eprint.iacr.org/2022/1530. 2022. URL: https://eprint.iacr.org/2022/1530.

[Hal] *The halo2 book*. URL: https://zcash.github.io/halo2/.

[Ham22] J. Hamlin. *Big investors are finally serious about crypto. but experienced talent is still scarce.* 2022. URL: https://www.institutionalinvestor.com/article/b1x0gr2y3dzzp3/Big-Investors-Are-Finally-Serious-About-Crypto-But-Experienced-Talent-Is-Still-Scarce.

[Her] *Hermez*. https://polygon.technology/solutions/polygon-hermez/. 2022.

[Het] *Hetzner*. https://www.hetzner.com/. 2022.

[HLW06] S. Hoory, N. Linial, and A. Wigderson. "Expander graphs and their applications". In: *BULL. AMER. MATH. SOC.* 43.4 (2006), pp. 439–561.

[HR22] J. Holmgren and R. Rothblum. "Faster Sounder Succinct Arguments and IOPs". In: 2022.

[Hyr] *Hyrax reference implementation*. https://github.com/hyraxZK/hyraxZK.

[IKO] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. "Efficient Arguments without Short PCPs". In: *22nd Annual IEEE Conference on Computational Complexity (CCC 2007).*

[IKOS07] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. "Zero-knowledge from secure multiparty computation". In: *Proceedings of the annual ACM symposium on Theory of computing*. ACM. 2007, pp. 21–30.

[Int] *Hyperledger Sawtooth*. 2017. URL: https://sawtooth.hyperledger.org/ (visited on 2017).

[ISW21] Y. Ishai, H. Su, and D. J. Wu. "Shorter and faster post-quantum designated-verifier zksnarks from lattices". In: *CCS*. 2021.

[Jsn] "jSNARK". In: 2015.

[Kil92] J. Kilian. "A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract)". In: *STOC*. 1992.

[KKW18] J. Katz, V. Kolesnikov, and X. Wang. "Improved non-interactive zero knowledge with applications to post-quantum signatures". In: *CCS*. 2018.

[KMSWP] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts". In: *Proceedings of Symposium on security and privacy (SP), 2016.*

[KPPS20] A. E. Kosba, D. Papadopoulos, C. Papamanthou, and D. Song. "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs". In: *USENIX Security*. 2020.

[KRDO17] A. Kiayias, A. Russell, B. David, and R. Oliynykov. "Ouroboros: A provably secure proof-of-stake blockchain protocol". In: *Annual international cryptology conference*. Springer. 2017, pp. 357–388.

[KS16] S. Khot and R. Saket. "Hardness of Bipartite Expansion". In: *ESA*. 2016.

[KS22] A. Kothapalli and S. Setty. *SuperNova: Proving universal machine executions without universal circuits*. Cryptology ePrint Archive, Paper 2022/1758. https://eprint.iacr.org/2022/1758. 2022. URL: https://eprint.iacr.org/2022/1758.

[KST22] A. Kothapalli, S. Setty, and I. Tzialla. “Nova: Recursive Zero-Knowledge Arguments from Folding Schemes”. In: *Advances in Cryptology – CRYPTO 2022*. Ed. by Y. Dodis and T. Shrimpton. Cham: Springer Nature Switzerland, 2022, pp. 359–388. ISBN: 978-3-031-15985-5.

[Kwo14] J. Kwon. “Tendermint: Consensus without mining”. In: *Draft v. 0.6, fall* 1.11 (2014).

[KZG] A. Kate, G. M. Zaverucha, and I. Goldberg. “Constant-Size Commitments to Polynomials and Their Applications”. In: *ASIACRYPT 2010*, pp. 177–194.

[Lay] *LayerZero*. `https://layerzero.network/`. 2022.

[LFKN92] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. “Algebraic Methods for Interactive Proof Systems”. In: *J. ACM* 39.4 (Oct. 1992), pp. 859–868. ISSN: 0004-5411.

[Liba] “libsnark”. In: 2014.

[Libb] *Reference implementation of Libra*. `https://github.com/sunblaze-ucb/Libra`.

[Lin01] Y. Lindell. *Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation*. Cryptology ePrint Archive, Paper 2001/107. `https://eprint.iacr.org/2001/107`. 2001. URL: `https://eprint.iacr.org/2001/107`.

[Lip12] H. Lipmaa. “Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments”. In: *Theory of Cryptography Conference*. 2012.

[LKKO20] S. Lee, H. Ko, J. Kim, and H. Oh. *vCNN: Verifiable Convolutional Neural Network based on zk-SNARKs*. Cryptology ePrint Archive, Report 2020/584. 2020.

[LXZ21] T. Liu, X. Xie, and Y. Zhang. “zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy”. In: *CCS*. 2021.

[MBKM19] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. “Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings”. In: *CCS*. `https://eprint.iacr.org/2019/099`. 2019.

[Mer87] R. C. Merkle. “A digital signature based on a conventional encryption function”. In: *Conference on the theory and application of cryptographic techniques*. 1987.

[Mic00] S. Micali. “Computationally Sound Proofs”. In: *SIAM J. Comput.* (2000).

[Mie09] T. Mie. “Short PCPPs verifiable in polylogarithmic time with O (1) queries”. In: *Annals of Mathematics and Artificial Intelligence* (2009).

[Mul] *Multi-chain future likely as Ethereum’s DeFi dominance declines | Bloomberg Professional Services*. 2022. URL: `https://www.bloomberg.com/professional/blog/multi-chain-future-likely-as-ethereums-defi-dominance-declines/` (visited on 04/24/2022).

[Nak08] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Decentralized Business Review* (2008), p. 21260.

[Nea] *ETH-NEAR Rainbow Bridge – NEAR Protocol*. 2022. URL: `https://near.org/blog/eth-near-rainbow-bridge/` (visited on 05/02/2022).

[Noma] *Nomad crypto bridge loses $200 million in “chaotic” hack*. https://www.theverge.com/2022/8/2/23288785/nomad-bridge-200-million-chaotic-hack-smart-contract-cryptocurrency. 2022.

[Nomb] *Nomad Protocol*. https://docs.nomad.xyz/the-nomad-protocol/overview. 2021.

[NS07] A. Nachmias and A. Shapira. “Testing the Expansion of a Graph.” In: *Electronic Colloquium on Computational Complexity (ECCC)* 14 (Jan. 2007). DOI: 10.1016/j.ic.2009.09.002.

[PHGR13] B. Parno, J. Howell, C. Gentry, and M. Raykova. “Pinocchio: Nearly practical verifiable computation”. In: *IEEE S&P*. 2013, pp. 238–252.

[Pip76] N. Pippenger. “On the evaluation of powers and related problems”. In: *SFCS 1976*. IEEE Computer Society. 1976.

[Pola] *At least $611 million stolen in massive cross-chain hack*. 2021. URL: https://www.theblockcrypto.com/post/114045/at-least-611-million-stolen-in-massive-cross-chain-hack.

[Polb] *Poly Network*. https://poly.network/. 2020.

[Polc] *Polygon Hermez*. https://polygon.technology/solutions/polygon-hermez/. 2022.

[Pold] *Polygon Miden*. https://polygon.technology/solutions/polygon-miden/. 2022.

[Pole] *Polygon Zero*. https://polygon.technology/solutions/polygon-zero/. 2022.

[PST13] C. Papamanthou, E. Shi, and R. Tamassia. “Signatures of Correct Computation”. In: *TCC 2013*. 2013, pp. 222–242.

[QZLG21] K. Qin, L. Zhou, B. Livshits, and A. Gervais. “Attacking the defi ecosystem with flash loans for fun and profit”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2021, pp. 3–32.

[Rai] *Rainbow Bridge*. https://near.org/bridge/. 2020.

[RD16] L. Ren and S. Devadas. “Proof of space from stacked expanders”. In: *Theory of Cryptography Conference*. Springer. 2016, pp. 262–285.

[Ris] *Risc Zero*. https://www.risczero.com/. 2022.

[Ron] *Ronin Attack Shows Cross-Chain Crypto Is a ‘Bridge’ Too Far*. 2022. URL: https://www.coindesk.com/layer2/2022/04/05/ronin-attack-shows-cross-chain-crypto-is-a-bridge-too-far/ (visited on 04/24/2022).

[RR20] N. Ron-Zewi and R. D. Rothblum. “Local proofs approaching the witness length”. In: *FOCS*. 2020.

[RR22] N. Ron-Zewi and R. D. Rothblum. “Proving as fast as computing: Succinct arguments with constant prover overhead”. In: *STOC*. 2022.

[Sch80] J. T. Schwartz. “Fast probabilistic algorithms for verification of polynomial identities”. In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 701–717.

[SCPTZ21] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang. "Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments". In: *IACR Cryptol. ePrint Arch.* (2021), p. 599.

[Scr] *Scroll*. `https://scroll.io/`. 2022.

[Set20] S. Setty. "Spartan: Efficient and general-purpose zkSNARKs without trusted setup". In: *CRYPTO*. Springer International Publishing, 2020, pp. 704–737.

[SL20] S. Setty and J. Lee. *Quarks: Quadruple-efficient transparent zkSNARKs*. Cryptology ePrint Archive, Report 2020/1275. 2020.

[Spi96] D. A. Spielman. "Linear-time encodable and decodable error-correcting codes". In: *IEEE Transactions on Information Theory* 42.6 (1996), pp. 1723–1731.

[Sta] *Starkware*. `https://starkware.co/`. 2022.

[SZT02] D. Song, D. Zuckerman, and J. Tygar. "Expander graphs for digital stream authentication and robust overlay networks". In: *S&P*. IEEE. 2002.

[Tha13a] J. Thaler. "Time-Optimal Interactive Proofs for Circuit Evaluation". In: *Advances in Cryptology – CRYPTO 2013*. Ed. by R. Canetti and J. A. Garay. 2013. ISBN: 978-3-642-40084-1.

[Tha13b] J. Thaler. "Time-Optimal Interactive Proofs for Circuit Evaluation". In: *Annual International Cryptology Conference (CRYPTO)*. Ed. by R. Canetti and J. A. Garay. 2013. ISBN: 978-3-642-40084-1.

[Tha15] J. Thaler. *A Note on the GKR Protocol*. Available at `http://people.cs.georgetown.edu/jthaler/GKRNote.pdf`. 2015.

[Tur90] K. Turkowski. "Filters for common resampling tasks". In: *Graphics gems*. Academic Press Professional, Inc. 1990, pp. 147–165.

[Use] *Average Price of Electricity*. `https://www.eia.gov/electricity/monthly/epm_table_grapher.php?t=epmt_5_6_a`. 2022.

[Var57] R. R. Varshamov. "Estimate of the number of signals in error correcting codes". In: *Dock-lady Akad. Nauk, SSSR* 117 (1957), pp. 739–741.

[Vira] *Reference implementation of Virgo*. `https://github.com/sunblaze-ucb/Virgo`. 2020.

[Virb] *Virgo implementation*. `https://github.com/TAMUCrypto/virgo-plus`. 2021.

[Vit] *An Incomplete Guide to Rollups*. `https://vitalik.ca/general/2021/01/05/rollup.html`. 2022.

[VSBW13] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. "A Hybrid Architecture for Interactive Verifiable Computation". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. 2013.

[Wa] R. S. Wahby and lcpc authors. *lcpc*. `https://github.com/conroi/lcpc`.

[Wah+17] R. S. Wahby et al. "Full accounting for verifiable outsourcing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017.

[WHGSW16] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish. "Verifiable asics". In: *IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 759–778.

[Win21] L. Wintermeyer. *Institutional money is pouring into the crypto market and its only going to grow*. 2021. URL: `https://www.forbes.com/sites/lawrencewintermeyer/2021/08/12/institutional-money-is-pouring-into-the-crypto-market-and-its-only-going-to-grow/?sh=2660a69d1459`.

[Woo+14] G. Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[Wora] *Blockchain Bridge Wormhole Suffers Possible Exploit Worth Over $326M*. 2022. URL: `https://www.coindesk.com/tech/2022/02/02/blockchain-bridge-wormhole-suffers-possible-exploit-worth-over-250m/` (visited on 2022).

[Worb] *Wormhole Solana*. `https://solana.com/wormhole`. 2020.

[WSRBW15] R. S. Wahby, S. T. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. "Efficient RAM and control flow in verifiable outsourced computation". In: *NDSS*. 2015.

[WTSTW18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. "Doubly-efficient zkSNARKs without trusted setup". In: *S&P*. IEEE. 2018, pp. 926–943.

[WYKW20] C. Weng, K. Yang, J. Katz, and X. Wang. "Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits". In: *S&P*. 2020.

[WYXKW21] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. *Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning*. USENIX Security. 2021.

[WZCPS18] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. "DIZK: A Distributed Zero-Knowledge Proof System". In: (2018).

[Xie+22] T. Xie et al. "zkBridge: Trustless Cross-chain Bridges Made Practical". In: *arXiv preprint arXiv:2210.00264* (2022).

[XZS22] T. Xie, Y. Zhang, and D. Song. "Orion: Zero Knowledge Proof with Linear Prover Time". In: Springer-Verlag, 2022.

[XZZPS19a] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *CRYPTO*. 2019.

[XZZPS19b] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *CRYPTO*. 2019.

[XZZPS19c] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *CRYPTO*. 2019.

[You] *YouTube includes NFTs in new creator tools*. 2022. URL: `https://www.nbcnews.com/pop-culture/viral/youtube-includes-nfts-new-creator-tools-rcna15813`.

[YSWW21] K. Yang, P. Sarkar, C. Weng, and X. Wang. "QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field". In: *CCS*. 2021.

[ZBKMNS22] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin. "Caulk: Lookup Arguments in Sublinear Time". In: *Cryptology ePrint Archive* (2022).

[Zca] *Zcash*. `https://z.cash/`.

[ZFZS20] J. Zhang, Z. Fang, Y. Zhang, and D. Song. "Zero Knowledge Proofs for Decision Tree Predictions and Accuracy". In: *CCS*. 2020.

[ZGKPP17a] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. *A Zero-Knowledge Version of vSQL*. Cryptology ePrint. 2017.

[ZGKPP17b] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. *A Zero-Knowledge Version of vSQL*. Cryptology ePrint Archive: Report 2017/1146. 2017.

[ZGKPP17c] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases". In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 863–880.

[ZGKPP17d] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases". In: *S&P*. 2017.

[ZGKPP18] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vRAM: Faster verifiable RAM with program-independent preprocessing". In: *S&P*. 2018.

[Zha+21a] J. Zhang et al. "Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time". In: *CCS*. 2021.

[Zha+21b] J. Zhang et al. "Doubly efficient interactive proofs for general arithmetic circuits with linear prover time". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 159–177.

[Zip79] R. Zippel. "Probabilistic algorithms for sparse polynomials". In: *International Symposium on Symbolic and Algebraic Manipulation*. Springer. 1979, pp. 216–226.

[Zkr] *An Incomplete Guide to Rollups*. `https://vitalik.ca/general/2021/01/05/rollup.html`.

[Zks] *zkSync*. `https://zksync.io/`. 2022.

[ZXHSZ22] J. Zhang, T. Xie, T. Hoang, E. Shi, and Y. Zhang. "Polynomial Commitment with a {One-to-Many} Prover and Applications". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 2965–2982.

[ZXZS20] J. Zhang, T. Xie, Y. Zhang, and D. Song. "Transparent polynomial delegation and its applications to zero knowledge proof". In: *S&P*. IEEE. 2020, pp. 859–876.