Efficient Zero-Knowledge Proofs: Theory and Practice

Jiaheng Zhang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-20
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-20.html

May 1, 2025

Efficient Zero-Knowledge Proofs: Theory and Practice

by

Jiaheng Zhang

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 Sanjam Garg
Associate Professor Nikhil Srivastava

Spring 2023

Efficient Zero-Knowledge Proofs: Theory and Practice

Abstract

Efficient Zero-Knowledge Proofs: Theory and Practice

by

Jiaheng Zhang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

Zero-knowledge proof is a cryptographic protocol enabling provers to convince verifiers of a statement's validity without disclosing any additional information beyond its truthfulness. It can be used to address security and privacy challenges in diverse fields, such as artificial intelligence, data analytics, and blockchain.

In this dissertation, we delve into the zero-knowledge proof, examining both its theoretical foundations and real-world applications. In theory, we introduce Libra, Virgo and Virgo++, a series of pioneering zero-knowledge protocols that boast optimal prover time, rapid verifier time, and succinct proof size while maintaining exceptional efficiency in practical settings. In practice, we explore the groundbreaking implementation of zero-knowledge proof in creating secure, trustless, and permissionless cross-chain bridges for blockchain networks. Furthermore, we also investigate the application of zero-knowledge proof in the realm of machine learning, showcasing its potential to ensure the integrity of machine learning models, as demonstrated with the decision tree model. The applied zero-knowledge proof protocols presented in this dissertation offer robust security assurances coupled with pragmatic efficiency.

To my family, friends, mentors, and colleagues.

Contents

List of Figures

List of Tables

Acknowledgments

In the course of completing my Ph.D. and dissertation, I have been fortunate to receive the unwavering support of numerous individuals. First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Dawn Song, whose invaluable guidance, steadfast encouragement, and genuine kindness have been instrumental throughout my Ph.D. journey. As an esteemed professor in computer security and privacy, Prof. Song consistently offered me inspiring insights, generously shared her extensive knowledge, and enriched our research discussions with her vast experience. Her diverse background in the industry has allowed her to introduce me to real-world challenges, fostering my ability to create impactful work and tackle the most complex problems.

I am profoundly thankful to Prof. Yupeng Zhang from Texas A&M University, whom I had the pleasure of meeting during his postdoctoral tenure at Berkeley while I began my Ph.D. in 2018. Our collaboration has grown from strength to strength since that initial encounter, and Prof. Zhang has proven to be an indispensable mentor and a trusted friend throughout my Ph.D. journey. He has guided me in every aspect of research, ranging from pinpointing valuable problems and employing cryptographic tools and elegant protocols, to uncovering potential applications, articulating motivations and contributions with clarity, and presenting our findings to the wider research community. Beyond research, his valuable life advice has been of immense support.

I also wish to convey my heartfelt appreciation to Prof. Elaine Shi at CMU, who has been both a mentor and a friend since my undergraduate days. Prof. Shi's outstanding achievements as a computer scientist have been a constant source of inspiration, while her genuine concern for my well-being has provided unwavering encouragement. She has offered invaluable advice regarding both my research and career aspirations. The knowledge and wisdom I have gained from Professors Song, Zhang, and Shi serve as my primary motivation for pursuing a faculty position, as I aspire to pass on their invaluable teachings to the next generation of students.

I would like to express my heartfelt appreciation to Prof. Sanjam Garg and Prof. Nikhil Srivastava for their invaluable contributions as members of both my dissertation and qualifying committees. I am equally grateful to Prof. Alessandro Chiesa for serving on my qualifying committee and for introducing me to cutting-edge zero-knowledge proof constructions during his course. My sincere thanks go to Prof. Raluca Popa for her role as my hooder at the Ph.D. commencement, as well as her invaluable support within the security group. Furthermore, I am profoundly grateful to Prof. Sanjam Garg and Prof. Raluca Popa for their unwavering assistance, guidance, and encouragement throughout my faculty job search. Their generosity was mirrored by Prof. Dan Boneh, Prof. Justin Thaler, Prof. Dawn Song, Prof. Elaine Shi, and Prof. Yupeng Zhang, to whom I extend my heartfelt thanks for their invaluable support.

Throughout my Ph.D. journey, I have had the privilege of collaborating with an exceptional group of individuals, whose contributions have been instrumental in shaping my research experience. I would like to extend my gratitude to Zerui Cheng, Arka Rai Choudhuri, Zhiyong Fang, Thang Hoang, Abhishek Jain, Zhengzhong Jin, Yongzheng Jia, Tianyi Liu, Charalampos Papamanthou, Weijie Wang, Tiancheng Xie, Xiang Xie, Fan Zhang, and Yinuo Zhang. Their collective expertise, creativity, and dedication have been invaluable to our collaborative efforts.

I would also like to express my appreciation to my former mentors, Prof. Xiaotie Deng, Prof. Yong Yu, Prof. Yu Yu, Prof. John Hopcroft, and others from Shanghai Jiao Tong University. As an undergraduate student in the ACM Class at Shanghai Jiao Tong University, I was fortunate to be guided by extraordinary teachers and advisors who introduced me to the captivating realm of computer science.

Throughout my academic journey, I have been incredibly fortunate to be surrounded by a close-knit circle of friends who have consistently provided unwavering support and encouragement during the highs and lows of life. Their presence has been indispensable in completing my Ph.D. I am truly grateful to have the company of friends from across the globe, spanning from China to the U.S.A., and covering diverse regions from the West Coast to the East Coast, the Bay Area to Boston, Los Angeles to New York, Seattle to Pittsburgh, Shanghai to Beijing, Singapore to Hong Kong, and beyond. While it is impossible to enumerate all their names, I would like to extend my deepest gratitude to all the colleagues and friends who have been by my side throughout this journey.

I am especially grateful to Tiancheng Xie, who has been a friend, co-author, roommate, and even family member to me. His support throughout the entire process has been invaluable, particularly when overcoming numerous challenges while I was in China during the COVID-19 pandemic.

I also wish to thank the members of the Association of Chinese Entrepreneurs (ACE) club at Berkeley. Since joining the club in my first year, I have relished discussing, playing, and socializing with its members, many of whom are undergraduate, master, and Ph.D. students from various departments. Their enthusiasm, passion, and vitality remind me to cherish my own formative years.

Additionally, I am grateful for the numerous friends who have shared conversations and leisure activities with me during ordinary days. We exchange news, stories, and emotions, and one such friend is Lianke Qin, who keeps me informed with daily messages and essential information.

I would also like to acknowledge the individuals in two WeChat groups: The Sun Never Sets Communication Group and the Chinese Graduate Communication Group of 2021. These groups have connected me with friends from around the world, expanding my horizons and bringing immense joy.

Lastly, my profound gratitude goes to my family for their unwavering support. I would like to express my heartfelt appreciation to my mother, Hongmei Hao, and my father, Jun Zhang, who have always provided a haven for me to recharge and gather strength. Their steadfast encouragement and unconditional love have been the driving force behind my journey.

Chapter 1

Introduction

The rapid growth of artificial intelligence, data analytics, and blockchains has resulted in an explosion of sensitive data, including personal information, that requires secure and simple privacy-preserving solutions. Unfortunately, data breaches have become increasingly frequent, resulting in significant economic losses and serious concerns about data privacy and security. Cryptographic algorithms and tools have been developed to address these challenges, and Zero-Knowledge Proofs [GMR] (ZKP) has emerged as a promising solution. ZKP is a cryptographic protocol that enables a prover to convince a verifier of the validity of a statement without disclosing any additional information beyond its truthfulness. For example, the prover can convince the verifier that there exists a x satisfying $f(x) = y$ for a public function f without revealing x to the verifier. However, applying ZKP to real-world use cases has been challenging due to the gap between its theoretical feasibility and practical efficiency, especially for large-scale statements. Therefore, this thesis aims to advance ZKP from both theoretical and application perspectives, bridging the gap. On the theory side, we propose novel ZKP protocols with optimal prover time, succinct proof size, and succinct verifier time, resolving a long-standing open problem. On the application side, we are the first to employ ZKP in machine learning, providing practical guarantees for the integrity and reproducibility of machine learning models. We also propose the first solution for building truly trustless and permissionless cross-chain bridges in blockchains using ZKP, called zkBridge. Our applied ZKP protocols provide rigorous security guarantees along with practical efficiency. The zkBridge technology has generated excitement in the industry, with numerous entities and researchers joining the zkBridge collective to bring the technology to wide industry adoption, creating a secure, universal foundation for multichain interoperability.

1.1 ZKP protocols with optimal prover computation

The area of ZKP has a rich history and has transitioned from a purely theoretical question to solutions deployed at sophisticated cryptocurrency constructions, e.g., ZCash [Ben+14]. However, the existing protocols cannot scale to complicated statements in practice because of heavy prover computations. In particular, the ZKP protocol used in ZCash needs a prover running quasi-linear

time in the statement size and each statement needs a separate trusted setup. They are both time-consuming. To address the bottleneck of the prover time in ZKP, we introduce three protocols, namely Libra [XZZPS19a], Virgo [ZXZS], and Virgo++ [Zha+20]. All these protocols focus on accelerating proof generation in ZKP. We also develop libraries of these protocols and released codes online [Liba; Vira; Virb] for developers.

1.1.1 Libra

Libra presents a groundbreaking solution to the long-standing open problem of creating a ZKP system based on layered arithmetic circuits that achieves optimal prover time, succinct proof size, and succinct verifier time. Specifically, for a circuit of size C, Libra's performance is as follows: (i) the prover time is $O(C)$, regardless of the circuit type; (ii) the proof size and verification time are both $O(d \log C)$ for d-depth log-space uniform circuits (e.g., RAM programs). Moreover, Libra features a one-time trusted setup that depends solely on the size of the input to the circuit and not on the circuit logic. The foundation of Libra is a new linear-time algorithm for the prover of the interactive proof protocol by Goldwasser, Kalai, and Rothblum (also known as the GKR protocol), as well as an efficient method for transforming the GKR protocol into zero-knowledge using small masking polynomials. Libra not only boasts impressive asymptotic performance but also demonstrates practical efficiency. For instance, our implementation reveals that it takes just 200 seconds to generate a proof for constructing a SHA2-based Merkle tree root with 256 leaves, outperforming all existing ZKP systems. Libra's proof size and verification time are equally competitive.

1.1.2 Virgo

Libra requires a one-time trusted setup, which we aim to eliminate by designing a transparent polynomial commitment (PC) scheme with succinct proof size and fast verification cost. Leveraging this PC scheme, we develop Virgo, a transparent ZKP protocol that offers significantly improved performance compared to previous transparent ZKP systems. Specifically, Virgo's prover time is at least an order of magnitude faster, while its verification time is a mere tens of milliseconds. Our new succinct zero-knowledge argument scheme for layered arithmetic circuits does not rely on a trusted setup and exhibits a prover time of $O(C + n \log n)$ and a proof size of $O(D \log C + \log^2 n)$ for a D-depth circuit with n inputs and C gates. Additionally, the verification time is succinct at $O(D \log C + \log^2 n)$ for structured circuits. Utilizing lightweight cryptographic primitives such as collision-resistant hash functions, our scheme is plausibly post-quantum secure. We have implemented Virgo based on our new scheme and compared its performance to existing systems. Experiments reveal that it takes only 53 seconds to generate a proof for a circuit computing a Merkle tree with 256 leaves—significantly faster than all other succinct zero-knowledge argument schemes. The verification time is 50ms, and the proof size is 253KB, both of which are competitive with existing systems. At the core of Virgo is a novel transparent zero-knowledge verifiable polynomial delegation scheme with logarithmic proof size and verification time. This interactive oracle proof model-based scheme may hold independent interest.

1.1.3 Virgo++

Both Libra and Virgo operate on layered arithmetic circuits, which are incompatible with other ZKP frameworks that utilize arbitrary arithmetic circuits. To address this limitation, we introduce Virgo++, which generalizes the optimal prover for arbitrary arithmetic circuits. In theory, Virgo++ shares the same asymptotic complexity as Virgo, but it directly supports arbitrary arithmetic circuits. In practice, Virgo++ boasts a 10x faster prover time and only a 1.2x larger proof size compared to Virgo for general arithmetic circuits. Owing to its exceptional efficiency, Virgo++ takes just 0.3 seconds to generate a proof for a circuit with over 600,000 gates, making it 13 times faster than the original interactive proof protocol for the corresponding layered circuit. The proof size is 208 kilobytes, and the verifier time is 66 milliseconds. Our key technique involves a new sumcheck equation that reduces a claim about the output of one layer to claims about its input only, rather than claims about all the layers above, which would inevitably incur an overhead proportional to the circuit's depth. We have developed efficient algorithms for the prover to run this sumcheck protocol and to combine multiple claims back into one in linear time based on the circuit's size.

1.2 Applications of ZKP on machine learning and blockchains

The above work has demonstrated significant improvements in the prover time of ZKP for general computations in theoretical and practical performances. On the practical side, we find more real-world applications of ZKP and design tailored ZKP schemes with concrete optimizations for these applications, particularly in machine learning and blockchain.

1.2.1 Decision tree predictions and accuracy tests

Machine learning has emerged as a powerful tool in numerous applications across various fields. Despite its impressive success, concerns about the integrity of machine learning predictions and accuracy have been on the rise. For instance, a notable case reported in [Bot] reveals a company claiming to use machine learning techniques for building food delivery robots, when in reality, the robots were operated by remote workers. Additionally, the reproducibility of high-accuracy machine learning models is often challenging, and there is a lack of security guarantees for the correctness and consistency of machine learning predictions in production environments. To address these issues, we initiate the study of zero-knowledge machine learning and propose protocols for zero-knowledge decision tree predictions and accuracy tests in [ZFZS20]. These protocols enable the owner of a decision tree model to prove the model's predictions on a data sample or its accuracy on a public dataset without revealing any information about the model itself. We also develop efficient approaches for transforming decision tree predictions and accuracy tests into statements of ZKP. The protocols are implemented using a backend-efficient ZKP scheme, demonstrating their practical efficiency. This pioneering work lays the foundation for building fair and secure trading platforms for machine learning models on blockchains. Instead of posting the model directly on the blockchain, which exposes it to all users, the model provider posts a short zero-knowledge proof about the model's quality. Smart contracts are then used to enforce payment and model delivery

simultaneously. This groundbreaking research has inspired the burgeoning field of zero-knowledge machine learning, with subsequent work following the established framework and extending it to convolutional neural networks [LXZ21; FQZDC21].

1.2.2 Cross-chain bridges for blockchains

Since the advent of Bitcoin, the cryptocurrency market has experienced rapid growth, reaching a valuation of over 1 trillion USD in just 13 years. Moreover, with the rise of decentralized finance, blockchain technology has evolved to offer a variety of financial services that traditional finance cannot accommodate through automated smart contracts, such as flash loans. However, several critical issues still impede the further advancement of blockchains, one of which is the interoperability between different blockchains. To tackle this issue, we develop zkBridge in [Xie+22], an efficient cross-chain bridge that ensures robust security without relying on external trust assumptions. We utilize zero-knowledge proofs based on cryptographic assumptions, rather than a committee of validators often employed in existing constructions, to guarantee the accuracy of relayed block headers from one chain to another. With these relayed block headers, the bridges can facilitate message passing, token transfers, and other computational logic operations on state changes across different chains. As a result, zkBridge is viewed as the infrastructure of the multi-chain universe. At the core of this innovation is deVirgo, a distributed ZKP protocol derived from Virgo. deVirgo accelerates proof generation in Virgo by using distributed machines, achieving optimal linear scalability and minimal communication costs. Furthermore, we apply recursive proofs to deVirgo to further reduce the proof size and on-chain verification costs. zkBridge has made a significant impact on the industry, with dozens of entities and researchers joining the zkBridge collective to promote widespread adoption of the technology.

1.2.3 Polynomial commitment with a one-to-many prover and applications on blockchains.

Additionally, to enhance the functionality and scalability of verifiable secret sharing (VSS) and distributed key generation (DKG) for blockchains, we develop a one-to-many fashion of ZKP protocols and applied it to VSS and DKG in [ZXHSZ22]. Specifically, our protocol features one prover and N verifiers who share common computation, but each verifier possesses distinct outputs. We constructed an efficient ZKP protocol in this new setting with optimal complexity, capable of batching N proofs in a single step. A direct application of this novel ZKP protocol fashion is Shamir's VSS scheme. Consequently, we apply the protocol to create efficient VSS and DKG schemes, particularly suitable for a large number of parties. VSS can be employed to securely store secret keys of cryptocurrencies, while DKG can be used to generate random beacons on blockchains. Other applications of this scheme are of independent interest and hold the potential for significant advancements in various areas.

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 [XZZPS19a].

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. [BBBPWM] 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 [ZGKPP17b]. 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 3.2 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 [BBBPWM] 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 [BSCTV] and Bulletproofs [BBBPWM], and our verification is slower than libSNARK [BSCTV] 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 [BSCTV] 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 [BSCTV; 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 [BSCTV]	Ligero [AHIV17]	Bulletproofs [BBBPWM]	Hyrax [WTSTW18]	libSTARK [BSBHR19]	Aurora [BSCRSVW19]	Libra
$\mathcal{G}$	$O(C)$ per-statement trusted setup	no trusted setup					$O(n)$ one-time trusted setup
$\mathcal{P}$	$O(C \log C)$	$O(C \log C)$	$O(C)$	$O(C \log C)$	$O(C \log^2 C)$	$O(C \log C)$	$O(C)$
$\mathcal{V}$	$O(1)$	$O(C)$	$O(C)$	$O(\sqrt{n} + d \log C)$	$O(\log^2 C)$	$O(C)$	$O(d \log C)$
$\|\pi\|$	$O(1)$	$O(\sqrt{C})$	$O(\log C)$	$O(\sqrt{n} + d \log C)$	$O(\log^2 C)$	$O(\log^2 C)$	$O(d \log C)$
$\mathcal{G}$	1027s	NA					210s
$\mathcal{P}$	360s	400s	13,000s	1,041s	2,022s	3199s	201s
$\mathcal{V}$	0.002s	4s	900s	9.9s	0.044s	15.2s	0.71s
$\|\pi\|$	0.13KB	1,500KB	5.5KB	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 3.4 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 7.2.2.1), 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

[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.

programming 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 4.4.2, 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; BSCGTV; BFRSBW; BSCTV14; Cos+; WSRBW15; FFGKOP16] including libSNARK [BSCTV]. 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], Ames et al. [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 [BBBPWM], 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 [ZGKPP17b], 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 [CD]. 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 $\mathbf{A}$ 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 [ZGKPP17b; 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:*

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

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

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

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

  $\mathcal{V}$ 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

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

  $\mathcal{V}$ 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

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

  $\mathcal{V}$ 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

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

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 24. 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 $x_1, x_2, \ldots, x_l \in \{0,1\}^l$.*

$\tilde{V}$ 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 $\mathbf{A} = (a_0, a_1, \ldots, a_{n-1})$ 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:

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

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:

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

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

Protocol. With Equation 7.1, 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 7.1 with $i = 0$. As described in Section 7.2.2.1, 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 7.1, $\mathcal{V}$ 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 7.1, 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 6.3 instead of Equation 7.1. 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 9.

Theorem 2.2.4. *[VSBW13][Tha13a][CMT12][GKR15]. Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d layered arithmetic circuit. Protocol 9 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 $T = \mathsf{polylog}\,|C|$.*

Protocol 2. Let $\mathbb{F}$ be a prime field. Let C: $\mathbb{F}^n \to \mathbb{F}^k$ 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

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

  - 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.*

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

- ***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

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

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

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

and

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

Algorithm 1 $\mathcal{F} \leftarrow$ 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 6
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 6 of Algorithm 8 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 20 SumCheck[3].

Complexity analysis. Both Algorithms 8 and 20 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 20 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 $\{a_1, \ldots, a_\ell\} \leftarrow \mathsf{SumCheck}(f, \mathbf{A}, r_1, \ldots, r_\ell)$

Input: Multilinear f on ℓ variables, initial bookkeeping table $\mathbf{A}$, random $r_1, \ldots, r_\ell$;

Output: ℓ 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});

1: $\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}, r_1, \ldots, r_\ell)$
2: **for** $i = 1, \ldots, \ell$ **do**
3: **for** $t \in \{0, 1, 2\}$ **do**
4: $a_{it} = \sum_{b \in \{0,1\}^{\ell - i}} f(r_1, \ldots, r_{i-1}, t, b)$ ▷ All evaluations needed are in $\mathcal{F}$.
5: **end for**
6: **end for**
7: **return** $\{a_1, \ldots, a_\ell\}$;

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

Input: Multilinear f and g, initial bookkeeping tables $\mathbf{A}_f$ and $\mathbf{A}_g$, random $r_1, \ldots, r_\ell$;

Output: ℓ 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});

1: $\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}_f, r_1, \ldots, r_\ell)$
2: $\mathcal{G} \leftarrow \mathsf{FunctionEvaluations}(g, \mathbf{A}_g, r_1, \ldots, r_\ell)$
3: **for** $i = 1, \ldots, \ell$ **do**
4: **for** $t \in \{0, 1, 2\}$ **do**
5: $a_{it} = \sum_{b \in \{0,1\}^{\ell - i}} f(r_1, \ldots, r_{i-1}, t, b) \cdot g(r_1, \ldots, r_{i-1}, t, b)$ ▷ All evaluations needed are in $\mathcal{F}$ and $\mathcal{G}$.
6: **end for**
7: **end for**
8: **return** $\{a_1, \ldots, a_\ell\}$;

The above can be easily computed by computing evaluations for functions f and g *separately* using Algorithm 8 and the combining the results using our new Algorithm 9 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

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

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 7.1 and 6.3 satisfy these properties.

We note here that applying Algorithm 8 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 5 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 7 of Algorithm 8 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 4.8 as follows

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

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 9. 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 $I(g,z) = \prod_{i=1}^{\ell}((1-g_i)(1-z_i)+g_iz_i))$.*

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, $I(w,z) = \prod_{i=1}^{\ell}((1-w_i)(1-z_i)+w_iz_i))$ 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 $I(g,z) = \prod_{i=1}^{\ell}((1-g_i)(1-z_i)+g_iz_i))$. □

Lemma 2.3.3. *The bookkeeping table $\boldsymbol{A}_{h_g}$ 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 4.3.2, 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 10.

Procedure $\mathsf{Precompute}(g)$ is to evaluate $\mathbf{G}[z] = I(g,z) = \prod_{i=1}^{\ell}((1-g_i)(1-z_i)+g_iz_i))$ 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 4.3.2. 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 9 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 $\mathbf{A}_{h_g} \leftarrow$ 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 $g = g_1, \ldots, g_\ell$;
Output: Bookkeeping table $\mathbf{A}_{h_g}$;

procedure $\mathbf{G} \leftarrow$ Precompute(g) ▷ $\mathbf{G}$ is an array of size 2^ℓ.
    Set $\mathbf{G}[0] = 1$
    for $i = 0, \ldots, \ell - 1$ **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 4.3.2. □

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

Proof. Similar to Algorithm 10, 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 $\mathbf{A}_{f_1} \leftarrow$ 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 $\mathbf{A}_{f_1}$;

1: $\mathbf{G} \leftarrow$ Precompute(g)
2: $\mathbf{U} \leftarrow$ Precompute(u)
3: $\forall y \in \{0,1\}^\ell$, set $\mathbf{A}_{f_1}[y] = 0$
4: **for** every (z, x, y) such that $f_1(z, x, y)$ is non-zero **do**
5: $\quad \mathbf{A}_{f_1}[y] = \mathbf{A}_{f_1}[y] + \mathbf{G}[z] \cdot \mathbf{U}[x] \cdot f_1(z, x, y)$
6: **end for**
7: **return** $\mathbf{A}_{f_1}$;

Algorithm 6 $\{a_1, \ldots, a_{2\ell}\} \leftarrow$ 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: 2ℓ sumcheck messages for $\sum_{x,y \in \{0,1\}^\ell} f_1(g, x, y) f_2(x) f_3(y)$;

1: $\mathbf{A}_{h_g} \leftarrow$ Initialize_PhaseOne$(f_1, f_3, \mathbf{A}_{f_3}, g)$
2: $\{a_1, \ldots, a_\ell\} \leftarrow$ SumCheckProduct$(\sum_{y \in \{0,1\}^\ell} f_1(g, x, y) f_3(y), \mathbf{A}_{h_g}, f_2, \mathbf{A}_{f_2}, u_1, \ldots, u_\ell)$
3: $\mathbf{A}_{f_1} \leftarrow$ Initialize_PhaseTwo(f_1, g, u)
4: $\{a_{\ell+1}, \ldots, a_{2\ell}\} \leftarrow$ SumCheckProduct$(f_1(g, u, y), \mathbf{A}_{f_1}, f_3(y) \cdot f_2(u), \mathbf{A}_{f_3} \cdot f_2(u), v_1, \ldots, v_\ell)$
5: **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

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

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 7.1 can be decomposed into several instances that have the form of Equation 4.8 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 4.8. 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 7.2.2, 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 6.3 instead of Equations 7.1, which still satisfies the properties of Equation 4.8. One could view it as 6 instances of Equation 4.8 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 10, 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 4.8.

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 7.2.2, 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 [ZGKPP17b], 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 7.1, 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 12.

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 12.*

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

Construction 1. *We assume the existence of a zkVPD protocol defined in Section 3.2.2. For simplicity, we omit the randomness r_f and public parameters* pp, vp *without any ambiguity. To prove the claim $H = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$:*

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 $H = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$, $G = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} g(x_1, x_2, \ldots, x_\ell)$ 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 12. 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 7.1, 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

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

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 7.1 becomes

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

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 3.4, 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 zero-knowledge 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 4.12 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 $R_1(z_1,w),\ldots,R_d(z_1,w) : \mathbb{F}^2 \to \mathbb{F}$ 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 12 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 $i = 1,\ldots,d-1$:*

   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.*

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

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 7.2.1, 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 4.4.2 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 $R_1^*(z_1, w), \ldots, R_d^*(z_1, w) : \mathbb{F}^2 \to \mathbb{F}$ 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

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

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

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

$a_{i+1} = \alpha^{(i)} \tilde{mult}_{i+1}(u^{(i)}, u^{(i+1)}, v^{(i+1)}) + \beta^{(i)} \tilde{mult}_{i+1}(v^{(i)}, u^{(i+1)}, v^{(i+1)})$ 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 $a_0, \ldots, a_6$ 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 $a_0, \ldots a_6$ 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 4.4.2. 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 4.4.2, 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 [ZGKPP17b; 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 6.2.2, 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 3.3.2.

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 [FS]. 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 3.4, 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 7.1. 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 10 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
	[ZGKPP17b]	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: $k(x) = \text{sinc}(x)/\text{sinc}(ax), \text{if} -a < x < a; k(x) = 0, \text{otherwise}$, 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-$8\times$, 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 [BSCTV], Ligero[AHIV17], libSTARK[BSBHR19], Hyrax[WTSTW18], Bulletproofs[BBBPWM] and Aurora [BSCRSVW19]. See Section 6.1 for more explanations of these systems and their asymptotic.

- libSNARK: We use jsnark [Jsna] to write the circuits (rank one constraint system (R1CS)), which compiles them to zero knowledge proofs using the libSNARK backend [Libb].
- 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 [Jsna] 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 3.5.

Prover time. As shown in Figure 3.5(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

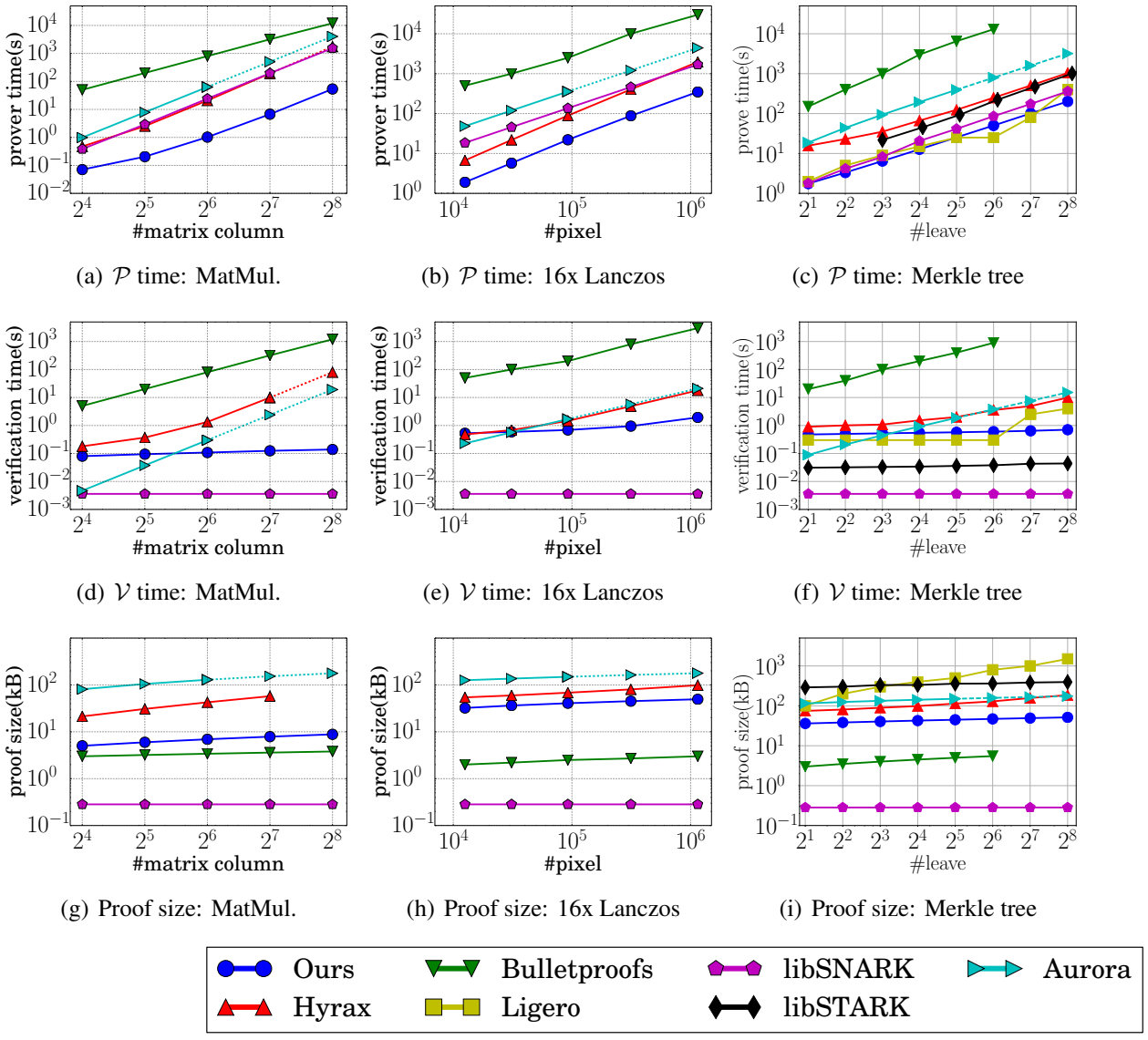


Figure 2.1: Comparisons of prover time, proof size and verification time between Libra and existing zero knowledge proof systems.

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 3.5(a) and 3.5(b). The performance of Libra is comparable to libSNARK on Merkle trees in Figure 3.5(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 [Jsna] 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 $2\times$ 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 3.5(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 3.5(f), the verification time of Libra is $8\times$ slower than Aurora when $M = 2$, and $15\times$ faster when $M = 256$. Libra is $2.5\times$ slower than Ligero with $M = 2$ and $4\times$ 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 3.5(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

Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof

We present a new succinct zero knowledge argument scheme for layered arithmetic circuits without trusted setup. The prover time is $O(C + n \log n)$ and the proof size is $O(D \log C + \log^2 n)$ for a D-depth circuit with n inputs and C gates. The verification time is also succinct, $O(D \log C + \log^2 n)$, if the circuit is structured. Our scheme only uses lightweight cryptographic primitives such as collision-resistant hash functions and is plausibly post-quantum secure. We implement a zero knowledge argument system, Virgo, based on our new scheme and compare its performance to existing schemes. Experiments show that it only takes 53 seconds to generate a proof for a circuit computing a Merkle tree with 256 leaves, at least an order of magnitude faster than all other succinct zero knowledge argument schemes. The verification time is 50ms, and the proof size is 253KB, both competitive to existing systems.

Underlying Virgo is a new transparent zero knowledge verifiable polynomial delegation scheme with logarithmic proof size and verification time. The scheme is in the interactive oracle proof model and may be of independent interest.

This work was previously published in [ZXZS].

3.1 Introduction

Zero knowledge proof (ZKP) allows a powerful prover to convince a weak verifier that a statement is true, without leaking any extra information about the statement beyond its validity. In recent years, significant progress has been made to bring ZKP protocols from purely theoretical interest to practical implementations, leading to its numerous applications in delegation of computations, anonymous credentials, privacy-preserving cryptocurrencies and smart contracts.

Despite of these great success, there are still some limitations of existing ZKP systems. In SNARK [PHGR13], the most commonly adopted ZKP protocol in practice, though the proof sizes are of just hundreds of bytes and the verification times are of several milliseconds regardless of the size of the statements, it requires a trusted setup phase to generate structured reference string (SRS) and the security will be broken if the trapdoor is leaked.

To address this problem, many ZKP protocols based on different techniques have been proposed recently to remove the trusted setup, which are referred as *transparent* ZKP protocols. Among these techniques, ZKP schemes based on the doubly efficient interactive proof proposed by Goldwasser et al. in [GKR15] (referred as *GKR* protocol in this paper) are particularly interesting due to their efficient prover time and sublinear verification time for statements represented as structured arithmetic circuits, making it promising to scale to large statements. Unfortunately, as of today we are yet to construct an efficient transparent ZKP system based on the GKR protocol with succinct[1] proof size and verification time. The transparent scheme in [WTSTW18] has square-root proof size and verification time, while the succinct scheme in [XZZPS19a] requires a one-time trusted setup. See Section 3.1.2 for more details.

Our contributions. In this paper, we advance this line of research by proposing a transparent ZKP protocol based on GKR with succinct proof size and verification time, when the arithmetic circuit representing the statement is structured. The prover time of our scheme is particularly efficient, at least an order of magnitude faster than existing ZKP systems, and the verification time is merely tens of milliseconds. Our concrete contributions are:

- **Transparent zero knowledge verifiable polynomial delegation.** We propose a new zero knowledge verifiable polynomial delegation (zkVPD) scheme without trusted setup. Compared to existing pairing-based zkVPD schemes [PST13; ZGKPP17c; ZGKPP17a], our new scheme does not require a trapdoor and linear-size public keys, and eliminates heavy cryptographic operations such as modular exponentiation and bilinear pairing. Our scheme may be of independent interest, as polynomial delegation/commitment has various applications in areas such as verifiable secret sharing [BKP11], proof of retrievability [YY13] and other constructions of ZKP [MBKM19].

- **Transparent zero knowledge argument.** Following the framework proposed in [ZGKPP17c], we combine our new zkVPD protocol with the GKR protocol efficiently to get a transparent ZKP scheme. Our scheme only uses light-weight cryptographic primitives such as collision-resistant hash functions and is plausibly post-quantum secure.

- **Implementation and evaluation.** We implement a ZKP system, Virgo, based on our new scheme. We develop optimizations such that our system can take arithmetic circuits on the field generated by Mersenne primes, the operations on which can be implemented efficiently using integer additions, multiplications and bit operations in C++. We plan to open source our system.

[1] "succinct" denotes poly-logarithmic in the size of the statement C.

3.1.1 Our Techniques

Our main technical contribution in this paper is a new transparent zkVPD scheme with $O(N \log N)$ prover time, $O(\log^2 N)$ proof size and verification time, where N is the size of the polynomial. We summarize the key ideas behind our construction. We first model the polynomial evaluation as the inner product between two vectors of size N: one defined by the coefficients of the polynomial and the other defined by the evaluation point computed on each monomial of the polynomial. The former is committed by the prover (or delegated to the prover after preprocessing in the case of delegation of computation), and the later is publicly known to both the verifier and the prover. We then develop a protocol that allows the prover to convince the verifier the correctness of the inner product between a committed vector and a public vector with proof size $O(\log^2 N)$, based on the univariate sumcheck protocol recently proposed by Ben-Sasson et al. in [BSCRSVW19] (See Section 3.2.4). To ensure security, the verifier needs to access the two vectors at some locations randomly chosen by the verifier during the protocol. For the first vector, the prover opens it at these locations using standard commitment schemes such as Merkle hash tree. For the second vector, however, it takes $O(N)$ time for the verifier to compute its values at these locations locally. In order to improve the verification time, we observe that the second vector is defined by the evaluation point of size only ℓ for a ℓ-variate polynomial, which is $O(\log N)$ if the polynomial is dense. Therefore, this computation can be viewed as a function that takes ℓ inputs, expands them to a vector of N monomials and outputs some locations of the vector. It is a perfect case for the verifier to use the GKR protocol to delegate the computation to the prover and validate the output, instead of computing locally. With proper design of the GKR protocol, the verification time is reduced to $O(\log^2 N)$ and the total prover time is $O(N \log N)$. We then turn the basic protocol into zero knowledge using similar techniques proposed in [AHIV17; BSCRSVW19]. The detailed protocols are presented in Section 3.3.

3.1.2 Related Work

Zero knowledge proof. Zero knowledge proof was introduced by Goldwasser et al. in [GMR89] and generic constructions based on probabilistically checkable proofs (PCPs) were proposed in the seminal work of Kilian [Kil92] and Micali [Mic00] in the early days. In recent years there has been significant progress in efficient ZKP protocols and systems. Following earlier work of Ishai [IKO], Groth [Gro10] and Lipmaa [Lip12], Gennaro et al. [GGPR13] introduced quadratic arithmetic programs (QAPs), which leads to efficient implementations of SNARKs [PHGR13; BSCGTV; BFRSBW; BSCTV14; Cos+; WSRBW15; FFGKOP16]. The proof size and verification time of SNARK are constant, which is particularly useful for real-world applications such as cryptocurrencies [Ben+14] and smart contract [KMSWP; BCGMMW18]. However, SNARKs require a per-statement trusted setup, and incurs a high overhead in the prover running time and memory consumption, making it hard to scale to large statements. There has been great research for generating the SRS through multi-parity secure computations [BSCGTV15] and making the SRS universal and updatable [GKMMM18; MBKM19].

Many recent works attempt to remove the trusted setup and construct transparent ZKP schemes. Based on "(MPC)-in-the-head" introduced in [IKOS07; GMO16; Cha+17], Ames et al. [AHIV17] proposed a ZKP scheme called Ligero. It only uses symmetric key operations and the prover time is fast in practice, but the proof size is $O(\sqrt{C})$ and the verification time is quasi-linear to the size of the circuit. Later, it is categorized as interactive oracle proofs (IOPs), and in the same model Ben-Sasson et al. built Stark [BSBHR19], transparent ZKP 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 $O(\log^2 C)$. Our new zkVPD and ZKP schemes fall in the IOP model.

In the seminal work of [GKR15], Goldwasser et al. proposed an efficient interactive proof for layered arithmetic circuits, which was extended to an arugment system by Zhang et al. in [ZGKPP17b] using a protocol for verifiable polynomial delegation. Later, Zhang et al. [ZGKPP18], Wahby et al. [WTSTW18] and Xie et al. [XZZPS19a] made the argument system zero knowledge by Cramer and Damgard transformation [CD] and random masking polynomials [CFS17]. The scheme of [WTSTW18], Hyrax, is transparent, yet the proof size and verification time are $O(\sqrt{n})$ where n is the input size of the circuit; the schemes of [ZGKPP17a] and [XZZPS19a] are succinct for structured circuits, but require one-time trusted setup. The prover time of the GKR protocol is substantially improved in [CMT12; Tha13b; Wah+17; WTSTW18; ZGKPP18], and recently Xie et al. [XZZPS19a] proposed a variant with $O(C)$ prover time for arbitrary circuits.

Other transparent ZKP schemes based on different techniques include discrete-log-based schemes [Gro09; BG12; BCCGP16; BBBPWM], hash-based schemes [BCGGHJ17] and lattice-based schemes [BBCDPGL18]. See Section 3.5.3 for detailed asymptotic complexity and practical performance of state-of-the-art systems with implementations.

Verifiable polynomial delegation. Verifiable polynomial delegation (VPD) allows a verifier to delegate the computation of polynomial evaluations to a powerful prover, and validates the result in time that is constant or logarithmic to the size of the polynomial. Earlier works in the literature include [KZG; BGV; FG]. Based on [KZG], Papamanthou et al. [PST13] propose a protocol for multivariate polynomials. Later in [ZGKPP17c], Zhang et al. extend the scheme to an argument of knowledge using powers of exponent assumptions, allowing a prover to commit to a multivariate polynomial, and open to evaluations at points queried by the verifier. In [ZGKPP17a], Zhang et al. further make the scheme zero knowledge. These schemes are based on bilinear maps and require a trusted setup phase that generates linear-size public keys with a trapdoor.

In a concurrent work, Bünz et al. [BFS19] propose another transparent polynomial commitment scheme without trusted setup. The scheme utilizes groups of unknown order and the techniques are different from our construction. The prover and verifier time are $O(N)$ and $O(\log N)$ modulo exponentiation in the group and the proof size is $O(\log N)$ group elements. Concretely, the proof size is 10-20KB for a circuit with 2^{20} gates when compiled to different ZKP systems [BFS19, Section 6], and the prover time and the verification time are not reported. Comparing to our scheme, we expect the prover and verifier time in our scheme are faster, while our proof size is larger, which gives an interesting trade-off.

3.2 Preliminaries

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. For a multivariate polynomial f, its "variable-degree" is the maximum degree of f in any of its variables. We often rely on polynomial arithmetic, which can be efficiently performed via fast Fourier tranforms and their inverses. In particular, polynomial evaluation and interpolation over a multiplicative coset of size n of a finite field can be performed in $O(n \log n)$ field operations via the standard FFT protocol, which is based on the divide-and-conquer algorthim.

3.2.1 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 through several rounds of interaction. We say that an interactive proof is public coin if $\mathcal{V}$'s challenge in each round is independent of $\mathcal{P}$'s messages in previous rounds. The proof system is interesting 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 3.2.1. *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 $\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x) = 1] = 1$.*
- ***ϵ-Soundness.*** *For any x with $f(x) \neq 1$ and any $\mathcal{P}^*$ it holds that $\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = 1] \leq \epsilon$*

Zero-knowledge arguments. 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 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 parameters pp. Formally, consider the definition below, where we assume R is known to $\mathcal{P}$ and $\mathcal{V}$.

Definition 3.2.2. *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 $(x, w) \in R$,*

$$\langle \mathcal{P}(\mathsf{pp}, w), \mathcal{V}(\mathsf{pp}) \rangle(x) = 1$$

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

$$\Pr[\langle \mathcal{P}(\mathsf{pp}), \mathcal{V}(\mathsf{pp}) \rangle(x) = 1 \wedge (x, w) \notin \mathcal{R} | w \leftarrow \varepsilon(\mathsf{pp}, x)]$$

- ***Zero knowledge***. *There exists a PPT simulator $\mathcal{S}$ such that for any PPT algorithm $\mathcal{V}^*$, auxiliary input $z \in \{0,1\}^*$, $(x; w) \in \mathcal{R}$, pp output by $\mathcal{G}(1^\lambda)$, it holds that*

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

*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 $\mathsf{poly}(\lambda, |x|, \log |w|)$.*

In the definition of zero knowledge, $\mathcal{S}^{\mathcal{V}^*}$ denotes that the simulator $\mathcal{S}$ is given the randomness of $\mathcal{V}^*$ sampled from polynomial-size space. This definition is commonly used in existing transparent zero knowledge proof schemes [AHIV17; BBBPWM; WTSTW18; BSCRSVW19].

3.2.2 Zero-Knowledge Verifiable Polynomial Delegation

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. We use $\mathcal{W}_{\ell,d}$ to denote the collection of all monomials in $\mathcal{F}$ and $N = |\mathcal{W}_{\ell,d}| = (d+1)^\ell$. A zero-knowledge verifiable polynomial delegation scheme (zkVPD) for $f \in \mathcal{F}$ and $t \in \mathbb{F}^\ell$ consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$,
- $\mathsf{com} \leftarrow \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp})$,
- $((y, \pi); \{0,1\}) \leftarrow \langle \mathsf{zkVPD.Open}(f, r_f), \mathsf{zkVPD.Verify}(\mathsf{com})\rangle(t, \mathsf{pp})$

Note that unlike the zkVPD in [PST13; ZGKPP17c; ZGKPP17a], our definition is transparent and does not have a trapdoor in zkVPD.KeyGen. π denotes the transcript seen by the verifier during the interaction with zkVPD.Open, which is similar to the proof in non-interactive schemes in [PST13; ZGKPP17c; ZGKPP17a].

Definition 3.2.3. *A zkVPD scheme satisfies the following properties:*

- ***Completeness.*** *For any polynomial $f \in \mathcal{F}$ and value $t \in \mathbb{F}^\ell$, $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$, $\mathsf{com} \leftarrow \mathsf{zkVPD.Commit}(f, r_f \mathsf{pp})$, it holds that*

$$\Pr\left[\langle \mathsf{zkVPD.Open}(f, r_f), \mathsf{zkVPD.Verify}(\mathsf{com})\rangle(t, \mathsf{pp}) = 1\right] = 1$$

- ***Soundness.*** *For any PPT adversary $\mathcal{A}$, $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$, the following probability is negligible of λ:*

$$\Pr\begin{bmatrix} (f^*, \mathsf{com}^*, t) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) \\ ((y^*, \pi^*); 1) \leftarrow \langle \mathcal{A}(), \mathsf{zkVPD.Verify}(\mathsf{com}^*)\rangle(t, \mathsf{pp}) \\ \mathsf{com}^* = \mathsf{zkVPD.Commit}(f^*, \mathsf{pp}) \\ f^*(t) \neq y^* \end{bmatrix}$$

- ***Zero Knowledge.*** *For security parameter λ, polynomial $f \in \mathcal{F}$, $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$, PPT algorithm $\mathcal{A}$, and simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$, consider the following two experiments:*

$\mathsf{Real}_{\mathcal{A},f}(\mathsf{pp})$:		$\mathsf{Ideal}_{\mathcal{A},\mathcal{S}^{\mathcal{A}}}(\mathsf{pp})$:
1. $\mathsf{com} \leftarrow \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp})$		1. $\mathsf{com} \leftarrow \mathcal{S}_1(1^\lambda, \mathsf{pp})$
2. $t \leftarrow \mathcal{A}(\mathsf{com}, \mathsf{pp})$		2. $t \leftarrow \mathcal{A}(\mathsf{com}, \mathsf{pp})$
3. (y, π) $\langle \mathsf{zkVPD.Open}(f, r_f), \mathcal{A}\rangle(t, \mathsf{pp})$	$\leftarrow$	3. $(y, \pi) \leftarrow \langle \mathcal{S}_2, \mathcal{A}\rangle(t_i, \mathsf{pp})$, *given oracle access to* $y = f(t)$.
4. $b \leftarrow \mathcal{A}(\mathsf{com}, y, \pi, \mathsf{pp})$		4. $b \leftarrow \mathcal{A}(\mathsf{com}, y, \pi, \mathsf{pp})$
5. *Output b*		5. *Output b*

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

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

3.2.3 Zero Knowledge Argument Based on GKR

In [XZZPS19a], Xie et al. proposed an efficient zero knowledge argument scheme named Libra. The scheme extends the interactive proof protocol for layered arithmetic circuits proposed by Goldwasser et al. [GKR15] (referred as the *GKR* protocol) to a zero knowledge argument using multiple instances of zkVPD schemes. Our scheme follows this framework and we review the detailed protocols here.

Sumcheck protocol. The sumcheck protocol is a fundamental protocol in the literature of interactive proof that has various applications. The problem is to sum a polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ on the binary hypercube $\sum_{b_1,b_2,...,b_\ell \in \{0,1\}} f(b_1, b_2, ..., b_\ell)$. 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 $\mathcal{V}$ the correctness of the sum. At the end of the sumcheck protocol, $\mathcal{V}$ needs an oracle access to the evaluation of f at a random point $r \in \mathbb{F}^\ell$ chosen by $\mathcal{V}$. The proof size of the sumcheck protocol is $O(d\ell)$, where d is the variable-degree of f, and the verification time of the protocol is $O(d\ell)$. The sumcheck protocol is complete and sound with $\epsilon = \frac{d\ell}{|\mathbb{F}|}$.

GKR protocol. 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 GKR protocol proceeds layer by layer. Upon receiving the claimed output from $\mathcal{P}$, in the first round, $\mathcal{V}$ and $\mathcal{P}$ run a 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 sumcheck. Finally, the protocol terminates with a claim about the input layer D, which can be checked directly by $\mathcal{V}$. If the check passes, $\mathcal{V}$ accepts the claimed output.

Formally speaking, we denote the number of gates in the i-th layer as S_i and let $s_i = \lceil \log S_i \rceil$. 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. As the sumcheck protocol works on $\mathbb{F}$, we then extend V_i to its *multilinear extension*, the unique polynomial $\tilde{V}_i : \mathbb{F}^{s_i} \to \mathbb{F}$ such that $\tilde{V}_i(x_1, x_2, ..., x_{s_i}) = V_i(x_1, x_2, ..., x_{s_i})$ for all $x_1, x_2, \ldots, x_{s_i} \in \{0,1\}^{s_i}$. As shown in prior work [CMT12], the closed form of $\tilde{V}_i$ can be computed as:

$$\tilde{V}_i(x_1, x_2, ..., x_{s_i}) = \sum_{b \in \{0,1\}^{s_i}} \prod_{i=1}^{s_i} [((1-x_i)(1-b_i) + x_i b_i) \cdot V_i(b)], \tag{3.1}$$

where b_i is i-th bit of b.

With these definitions, we can express the evaluations of $\tilde{V}_i$ as a summation of evaluations of $\tilde{V}_{i+1}$:

$$\alpha_i \tilde{V}_i(u^{(i)}) + \beta_i \tilde{V}_i(v^{(i)}) = \sum\nolimits_{x,y \in \{0,1\}^{s_{i+1}}} f_i(\tilde{V}_{i+1}(x), \tilde{V}_{i+1}(y)), \tag{3.2}$$

where $u^{(i)}, v^{(i)} \in \mathbb{F}^{s_i}$ are random vectors and $\alpha_i, \beta_i \in \mathbb{F}$ are random values. Note here that f_i depends on $\alpha_i, \beta_i, u^{(i)}, v^{(i)}$ and we omit the subscripts for easy interpretation.

With Equation 7.1, 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 $\tilde{V}_0(u^{(0)})$ and $\tilde{V}_0(v^{(0)})$ for random $u^{(0)}, v^{(0)} \in \mathbb{F}^{s_0}$. $\mathcal{V}$ then picks two random values α_0, β_0 and invokes a sumcheck protocol on Equation 7.1 with $\mathcal{P}$ for $i = 0$. As described before, at the end of the sumcheck, $\mathcal{V}$ needs an oracle access to

the evaluation of f_0 at $u^{(1)}, v^{(1)}$ randomly selected in $\mathbb{F}^{s_1}$. To compute this value, $\mathcal{V}$ asks $\mathcal{P}$ to send $\tilde{V}_1(u^{(1)})$ and $\tilde{V}_1(v^{(1)})$. Other than these two values, f_0 only depends on $\alpha_0, \beta_0, u^{(0)}, v^{(0)}$ and the gates and wiring in layer 0, which are all known to $\mathcal{V}$ and can be computed by $\mathcal{V}$ directly. In this way, $\mathcal{V}$ and $\mathcal{P}$ reduces two evaluations of $\tilde{V}_0$ to two evaluations of $\tilde{V}_1$ in layer 1. $\mathcal{V}$ and $\mathcal{P}$ then repeat the protocol recursively layer by layer. Eventually, $\mathcal{V}$ receives two claimed evaluations $\tilde{V}_D(u^{(D)})$ and $\tilde{V}_D(v^{(D)})$. $\mathcal{V}$ then checks the correctness of these two claims directly by evaluating $\tilde{V}_D$, which is defined by the input of the circuit. Let $\mathsf{GKR}.\mathcal{P}$ and $\mathsf{GKR}.\mathcal{V}$ be the algorithms for the GKR prover and verifier, we have the following theorem:

Lemma 3.2.4. *[GKR15; CMT12; Tha13b; XZZPS19a]. Let $C : \mathbb{F}^n \to \mathbb{F}$ be a layered arithmetic circuit with depth of D. $\langle \mathsf{GKR}.\mathcal{P}, \mathsf{GKR}.\mathcal{V} \rangle(C, x)$ is an interactive proof per Definition 7.2.1 for the function computed by C on input x with soundness $O(D \log |C|/|\mathbb{F}|)$. The total communication is $O(D \log |C|)$ and the running time of the prover $\mathcal{P}$ is $O(|C|)$. When C has regular wiring pattern[2], the running time of the verifier $\mathcal{V}$ is $O(n + D \log |C|)$.*

Extending GKR to Zero Knowledge Argument. There are two limitations of the GKR protocol: (1) It is not an argument system supporting witness from $\mathcal{P}$, as $\mathcal{V}$ needs to evaluate $\tilde{V}_D$ locally in the last round; (2) It is not zero knowledge, as in each round, both the sumcheck protocol and the two evaluations of $\tilde{V}_i$ leak information about the values in layer i.

To extend the GKR protocol to a zero knowledge argument, Xie et al. [XZZPS19a] address both of the problems using zero knowledge polynomial delegation. Following the approach of [ZGKPP17c; ZGKPP17a; WTSTW18], to support witness w as the input to the circuit, $\mathcal{P}$ commits to $\tilde{V}_D$ using zkVPD before running the GKR protocol. In the last round of GKR, instead of evaluating $\tilde{V}_D$ locally, $\mathcal{V}$ asks $\mathcal{P}$ to open $\tilde{V}_D$ at two random points $u^{(D)}, v^{(D)}$ selected by $\mathcal{V}$ and validates them using $\mathsf{zkVPD.Verify}$. In this way, $\mathcal{V}$ does not need to access w directly and the soundness still holds because of the soundness guarantee of zkVPD.

To ensure zero knowledge, using the techniques proposed by Chiesa et al. in [CFS17], the prover $\mathcal{P}$ masks the polynomial $\tilde{V}_i$ and the sumcheck protocol by random polynomials so that the proof does not leak information. For correctness and soundness purposes, these random polynomials are committed using the zkVPD protocol and opened at random points chosen by $\mathcal{V}$. In particular, for layer i, the prover selects a random bivariate polynomial $R_i(x_1, z)$ and defines

$$\dot{V}_i(x_1, \ldots, x_{s_i}) \stackrel{def}{=} \tilde{V}_i(x_1, \ldots, x_{s_i}) + Z_i(x_1, \ldots, x_{s_i}) \sum_{z \in \{0,1\}} R_i(x_1, z), \tag{3.3}$$

where $Z_i(x) = \prod_{i=1}^{s_i} x_i(1 - x_i)$, i.e., $Z_i(x) = 0$ for all $x \in \{0,1\}^{s_i}$. $\dot{V}_i$ is known as the *low degree extension* of V_i, as $\dot{V}_i(x) = \tilde{V}_i(x) = V_i(x)$ for all $x \in \{0,1\}^{s_i}$. As R_i is randomly selected by $\mathcal{P}$, revealing evaluations of $\dot{V}_i$ does not leak information about V_i, thus the values in the circuit. Additionally, $\mathcal{P}$ selects another random polynomial $\delta_i(x_1, \ldots, x_{s_{i+1}}, y_1, \ldots, y_{s_{i+1}}, z)$ to mask the sumcheck protocol. Let $H_i = \sum_{x,y \in \{0,1\}^{s_{i+1}}, z \in \{0,1\}} \delta_i(x_1, \ldots, x_{s_{i+1}}, y_1, \ldots, y_{s_{i+1}}, z)$, Equation 7.1 to run sumcheck on becomes

$$\begin{aligned} &\alpha_i \dot{V}_i(u^{(i)}) + \beta_i \dot{V}_i(v^{(i)}) + \gamma_i H_i \\ =&\sum\nolimits_{x,y \in \{0,1\}^{s_{i+1}}, z \in \{0,1\}} f_i'(\dot{V}_{i+1}(x), \dot{V}_{i+1}(y), R_i(u_1^{(i)}, z), R_i(v_1^{(i)}, z), \delta_i(x, y, z)), \end{aligned} \tag{3.4}$$

[2] "Regular" circuits is defined in [CMT12, Theorem A.1]. Roughly speaking, it means the mutilinear extension of its wiring predicates can be evaluated at a random point in time $O(\log |C|)$.

Protocol 3 (Zero Knowledge Argument in [XZZPS19a]). Let λ be the security parameter, $\mathbb{F}$ be a prime field. Let $C : \mathbb{F}^n \to \mathbb{F}$ be a layered arithmetic circuit over $\mathbb{F}$ with D layers, input in and witness w such that $|\mathsf{in}| + |w| \leq n$ and $1 = C(\mathsf{in}; w)$.

- $\mathcal{G}(1^\lambda)$: set pp as $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$.
- $\langle \mathcal{P}(\mathsf{pp}, w), \mathcal{V}(\mathsf{pp}) \rangle(\mathsf{in})$:
  1. $\mathcal{P}$ selects a random bivariate polynomial R_D. $\mathcal{P}$ commits to the witness of C by sending $\mathsf{com}_D \leftarrow \mathsf{zkVPD.Commit}(\dot{V}_D, r_{V_D}, \mathsf{pp})$ to $\mathcal{V}$, where $\dot{V}_D$ is defined by Equation 4.12.
  2. $\mathcal{P}$ randomly selects polynomials $R_i : \mathbb{F}^2 \to \mathbb{F}$ and $\delta_i : \mathbb{F}^{2s_{i+1}+1} \to \mathbb{F}$ for $i = 0, \ldots, D-1$. $\mathcal{P}$ commits to these polynomials by sending $\mathsf{com}_{i,1} \leftarrow \mathsf{zkVPD.Commit}(R_i, r_{R_i}, \mathsf{pp})$ and $\mathsf{com}_{i,2} \leftarrow \mathsf{zkVPD.Commit}(\delta_i, r_{\delta_i}, \mathsf{pp})$ to $\mathcal{V}$. $\mathcal{P}$ also reveals R_0 to $\mathcal{V}$, as V_0 is known to $\mathcal{V}$.
  3. $\mathcal{V}$ evaluates $\dot{V}_0(u^{(0)})$ and $\dot{V}_0(v^{(0)})$ for randomly chosen $u^{(0)}, v^{(0)} \in \mathbb{F}^{s_0}$.
  4. For $i = 0, \ldots, D-1$:
     a) $\mathcal{P}$ sends $H_i = \sum_{x,y \in \{0,1\}^{s_{i+1}}, z \in \{0,1\}} \delta_i(x, y, z)$ to $\mathcal{V}$.
     b) $\mathcal{V}$ picks $\alpha_i, \beta_i, \gamma_i$ randomly in $\mathbb{F}$.
     c) $\mathcal{V}$ and $\mathcal{P}$ execute a sumcheck protocol on Equation 3.4. At the end of the sumcheck, $\mathcal{V}$ receives a claim of f_i' at point $u^{(i+1)}, v^{(i+1)} \in \mathbb{F}^{s_{i+1}}, g_i \in \mathbb{F}$ selected randomly by $\mathcal{V}$.
     d) $\mathcal{P}$ opens $R_i(u^{(i)}, g_i)$, $R_i(v^{(i)}, g_i)$ and $\delta_i(u^{(i+1)}, v^{(i+1)}, g_i)$ using $\mathsf{zkVPD.Open}$. $\mathcal{P}$ sends $\dot{V}_0(u^{(i+1)})$ and $\dot{V}_0(v^{(i+1)})$ to $\mathcal{V}$.
     e) $\mathcal{V}$ validates $R_i(u^{(i)}, g_i)$, $R_i(v^{(i)}, g_i)$ and $\delta_i(u^{(i+1)}, v^{(i+1)}, g_i)$ using $\mathsf{zkVPD.Verify}$. If any of them outputs 0, abort and output 0.
     f) $\mathcal{V}$ checks the claim of f_i' using $R_i(u^{(i)}, g_i)$, $R_i(v^{(i)}, g_i)$, $\delta_i(u^{(i+1)}, v^{(i+1)}, g_i)$, $\dot{V}_0(u^{(i+1)})$ and $\dot{V}_0(v^{(i+1)})$. If it fails, output 0.
  5. $\mathcal{P}$ runs $(y_1, \pi_1) \leftarrow \mathsf{zkVPD.Open}(\dot{V}_D, r_{V_D}, u^{(D)}, \mathsf{pp})$, $(y_2, \pi_2) \leftarrow \mathsf{zkVPD.Open}(\dot{V}_D, r_{V_D}, v^{(D)}, \mathsf{pp})$ and sends y_1, π_1, y_2, π_2 to $\mathcal{V}$.
  6. $\mathcal{V}$ runs $\mathsf{Verify}(\pi_1, y_1, \mathsf{com}_D, u^{(D)}, \mathsf{pp})$ and $\mathsf{Verify}(\pi_2, y_2, \mathsf{com}_D, v^{(D)}, \mathsf{pp})$ and output 0 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. If all checks above pass, $\mathcal{V}$ output 1.

where $\gamma_i \in \mathbb{F}$ is randomly selected by $\mathcal{V}$, and f_i' is defined by $\alpha_i, \beta_i, \gamma_i, u^{(i)}, v^{(i)}, Z_i(u^{(i)}), Z_i(v^{(i)})$[3]. Now $\mathcal{V}$ and $\mathcal{P}$ can execute the sumcheck and GKR protocol on Equation 3.4. In each round, $\mathcal{P}$ additionally opens R_i and δ_i at $R_i(u_1^{(i)}, g^{(i)}), R_i(v_1^{(i)}, g^{(i)}), \delta_i(u^{(i+1)}, v^{(i+1)}, g^{(i)})$ for $g^{(i)} \in \mathbb{F}$ randomly selected by $\mathcal{V}$. With these values, $\mathcal{V}$ reduces the correctness of two evaluations $\dot{V}_i(u^{(i)}), \dot{V}_i(v^{(i)})$ to two evaluations $\dot{V}_i(u^{(i+1)}), \dot{V}_i(v^{(i+1)})$ on one layer above like before. In addition, as f_i is masked by δ_i, the sumcheck protocol is zero knowledge;

[3]Formally, f_i' is $I(0,z)f_i(\dot{V}_{i+1}(x), \dot{V}_{i+1}(y)) + I((x,y),0)(\alpha_i Z_i(u^{(i)})R(u_1^{(i)}, z) + \beta_i Z_i(v^{(i)})R(v_1^{(i)}, z)) + \gamma_i \delta_i(x, y, z)$, where $I(a, b)$ is an identity polynomial $I(a, b) = 0$ iff $a = b$. We will not use f_i' explicitly in our constructions later.

as $\tilde{V}_i$ is masked by R_i, the two evaluations of $\dot{V}_i$ do not leak information. The full zero knowledge argument protocol in [XZZPS19a] is given in Protocol 3. We have the following theorem:

Lemma 3.2.5. *[XZZPS19a]. Let $C : \mathbb{F}^n \to \mathbb{F}$ be a layered arithmetic circuit with D layers, input* in *and witness w. Protocol 3 is a zero knowledge argument of knowledge under Definition 6.2.2 for the relation defined by $1 = C(\mathsf{in}; w)$.*

The variable degree of R_i is $O(1)$. $\delta_i(x, y, z) = \delta_{i,1}(x_1) + \ldots + \delta_{i,s_{i+1}}(x_{s_{i+1}}) + \delta_{i,s_{i+1}+1}(y_1) + \ldots + \delta_{i,2s_{i+1}}(y_{s_{i+1}}) + \delta_{i,2s_{i+1}+1}(z)$ is the summation of $2s_{i+1} + 1$ univariate polynomials of degree $O(1)$. Other than the zkVPD instantiations, the proof size is $O(D \log |C|)$ and the prover time is $O(|C|)$. When C is regular, the verification time is $O(n + D \log |C|)$.

3.2.4 Univariate Sumcheck

Our transparent zkVPD protocol is inspired by the univariate sumcheck protocol recently proposed by Ben-Sasson et al.in [BSCRSVW19]. As the name indicates, the univariate sumcheck protocol allows the verifier to validate the result of the sum of a univariate polynomial on a subset $\mathbb{H}$ of the field $\mathbb{F}$: $\mu = \sum_{a \in \mathbb{H}} f(a)$. The key idea of the protocol relies on the following lemma:

Lemma 3.2.6. *[BC99]. Let $\mathbb{H}$ be a multiplicative coset[4] of $\mathbb{F}$, and let $g(x)$ be a univariate polynomial over $\mathbb{F}$ of degree strictly less that $|\mathbb{H}|$. Then $\sum_{a \in \mathbb{H}} g(a) = g(0) \cdot |\mathbb{H}|$.*

Because of Lemma 3.2.6, to test the result of $\sum_{a \in \mathbb{H}} f(a)$ for f with degree less than k, we can decompose f into two parts $f(x) = g(x) + Z_{\mathbb{H}}(x) \cdot h(x)$, where $Z_{\mathbb{H}}(x) = \prod_{a \in \mathbb{H}}(x - a)$ (i.e., $Z_{\mathbb{H}}(a) = 0$ for all $a \in \mathbb{H}$), and the degrees of g and h are strictly less than $|\mathbb{H}|$ and $k - |\mathbb{H}|$. This decomposition is unique for every f. As $Z_{\mathbb{H}}(a)$ is always 0 for $a \in \mathbb{H}$, $\mu = \sum_{a \in \mathbb{H}} f(a) = \sum_{a \in \mathbb{H}} g(a) = g(0) \cdot |\mathbb{H}|$ by Lemma 3.2.6. Therefore, if the claimed sum μ sent by the prover is correct, $f(x) - Z_{\mathbb{H}}(x) \cdot h(x) - \mu/|\mathbb{H}|$ must be a polynomial of degree less than $|\mathbb{H}|$ with constant term 0, or equivalently polynomial

$$p(x) = \frac{|\mathbb{H}| \cdot f(x) - |\mathbb{H}| \cdot Z_{\mathbb{H}}(x) \cdot h(x) - \mu}{|\mathbb{H}| \cdot x} \tag{3.5}$$

must be a polynomial of degree less than $|\mathbb{H}| - 1$. To test this, the univariate sumcheck uses a low degree test (LDT) protocol on Reed-Solomon (RS) code, which we define below.

Reed-Solomon Code. Let $\mathbb{L}$ be a subset of $\mathbb{F}$, an RS code is the evaluations of a polynomial $\rho(x)$ of degree less than m ($m < \mathbb{L}$) on $\mathbb{L}$. We use the notation $\rho|_{\mathbb{L}}$ to denote the vector of the evaluations $(\rho(a))_{a \in \mathbb{L}}$, and use $RS[\mathbb{L}, m]$ to denote the set of all such vectors generated by polynomials of degree less than m. Note that any vector of size $|\mathbb{L}|$ can be viewed as some univariate polynomial of degree less than $|\mathbb{L}|$ evaluated on $\mathbb{L}$, thus we use vector and polynomial interchangeably.

Low Degree Test and Rational Constraints. Low degree test allows a verifier to test whether a polynomial/vector belongs to an RS code, i.e., the vector is the evaluations of some polynomial of degree less than m on $\mathbb{L}$.

[4] In [BSCRSVW19], the protocols are mainly using additive cosets. We require $\mathbb{H}$ to be a multiplicative coset for our constructions over prime fields and extensions. The univariate sumsheck on multiplicative cosets is also stated in [BSCRSVW19].

In our constructions, we use the LDT protocol in [BSCRSVW19, Protocol 8.2], which was used to transform an RS-encoded IOP to a regular IOP. It applies the LDT protocol proposed in [BSBHR18] protocol to a sequence of polynomials $\vec{\rho}$ and their *rational constraint* p, which is a polynomial that can be computed as the division of the polynomials in $\vec{\rho}$. In the case of univariate sumcheck, the sequence of polynomials is $\vec{\rho} = (f, h)$ and the rational constraint is given by Equation 3.5.

The high level idea is as follows. First, the verifier multiplies each polynomial in $\vec{\rho}$ and the rational constraint p with an appropriate monomial such that they have the same degree max, and takes their random linear combination. Then the verifier tests that the resulting polynomial is in $RS[\mathbb{L}, \mathsf{max} + 1]$. At the end of the protocol, the verifier needs oracle access to κ evaluations of each polynomial in $\vec{\rho}$ and the rational constraint p at points in $\mathbb{L}$ indexed by $\mathcal{I}$, and checks that each evaluation of p is consistent with the evaluations of the polynomials in $\vec{\rho}$. We denote the protocol as $\langle \mathsf{LDT}.\mathcal{P}(\vec{\rho}, p), \mathsf{LDT}.\mathcal{V}(\vec{m}, \deg(p))\rangle(\mathbb{L})$, where $\vec{\rho}$ is a sequence of polynomials over $\mathbb{F}$, $p(x)$ is their rational constraint, $\vec{m}, \deg(p)$ is the degrees of the polynomials and the rational constraint to test, and $\mathbb{L}$ is a multiplicative coset of $\mathbb{F}$. We state the properties of the protocol in the following lemma:

Lemma 3.2.7. *There exist an LDT protocol* $\langle \mathsf{LDT}.\mathcal{P}(\vec{\rho}, p), \mathsf{LDT}.\mathcal{V}(\vec{m}, \deg(p))\rangle(\mathbb{L})$ *that is complete and sound with soundness error* $O(\frac{|\mathbb{L}|}{|\mathbb{F}|}) + \mathsf{negl}(\kappa)$*, given oracle access to evaluations of each polynomial in* $\vec{\rho}$ *at* κ *points indexed by* $\mathcal{I}$ *in* $\mathbb{L}$*. The proof size and the verification time are* $O(\log |\mathbb{L}|)$ *other than the oracle access, and the prover time is* $O(\mathbb{L})$*.*

The LDT protocol can be made zero knowledge in a straight-forward way by adding a random polynomial of degree max in $\vec{\rho}$. That is, there exists a simulator $\mathcal{S}$ such that given the random challenges of $\mathcal{I}$ of any PPT algorithm $\mathcal{V}^*$, it can simulate the view of $\mathcal{V}^*$ such that $\mathsf{View}(\langle \mathsf{LDT}.\mathcal{P}(\vec{\rho}, p), \mathcal{V}^*(\vec{m}, \deg(p))\rangle(\mathbb{L})) \approx \mathcal{S}^{\mathcal{V}^*}(\deg(p))$. In particular, $\mathcal{S}$ generates $p^* \in RS[\mathbb{L}, \deg(p)]$ and can simulate the view of any sequence of random polynomials $\vec{\rho}^*$ subject to the constraint that their evaluations at points indexed by $\mathcal{I}$ are consistent with the oracle access of p^*.

Merkle Tree. Merkle hash tree proposed by Ralph Merkle in [Mer] is a common primitive to commit a vector and open it at an index with logarithmic proof size and verification time. It consists of three algorithms:

- $\mathsf{root}_c \leftarrow \mathsf{MT.Commit}(c)$
- $(c_{idx}, \pi_{idx}) \leftarrow \mathsf{MT.Open}(idx, c)$
- $(1, 0) \leftarrow \mathsf{MT.Verify}(\mathsf{root}_c, idx, c_{idx}, \pi_{idx})$

The security follows the collision-resistant property of the hash function used to construct the Merkle tree.

With these tools, the univariate sumcheck protocol works as follows. To prove $\mu = \sum_{a \in \mathbb{H}} f(a)$, the verifier and the prover picks $\mathbb{L}$, a multiplicative coset of $\mathbb{F}$ and a superset of $\mathbb{H}$, where $|\mathbb{L}| > k$. $\mathcal{P}$ decompose $f(x) = g(x) + Z_{\mathbb{H}}(x) \cdot h(x)$ as defined above, and computes the vectors $f|_{\mathbb{L}}$ and $h|_{\mathbb{L}}$. $\mathcal{P}$ then commits to these two vectors using Merkle trees. $\mathcal{P}$ then defines a polynomial $p(x) = \frac{|\mathbb{H}| \cdot f(x) - |\mathbb{H}| \cdot Z_{\mathbb{H}}(x) \cdot h(x) - \mu}{|\mathbb{H}| \cdot x}$, which is a rational constraint of f and h. As explained above, in order to ensure the correctness of μ, it suffices to test that the degree of $(f, h), p$ is less than $(k, k - |\mathbb{H}|), |\mathbb{H}| - 1$, which is done through the low degree test. At the end of the LDT, $\mathcal{V}$ needs oracle access to κ points of $f|_{\mathbb{L}}$ and $h|_{\mathbb{L}}$. $\mathcal{P}$ sends these points with their Merkle tree proofs, and $\mathcal{V}$ validates their correctness. The formal protocol and the lemma is presented in Protocol 4. As shown in [BSCRSVW19], it suffices to set $|\mathbb{L}| = O(|\mathbb{H}|)$. And we have the following lemma:

Lemma 3.2.8. *Let $f : \mathbb{F} \to \mathbb{F}$ be a univariate poynomial with degree less than k and $\mathbb{H} \subseteq \mathbb{L} \subseteq \mathbb{F}$ and $|\mathbb{L}| > k$. Protocol 4 is an interactive proof to prove $\mu = \sum_{a\in\mathbb{H}} f(a)$ with soundness $O(\frac{\mathbb{L}}{\mathbb{F}} + \mathsf{negl}(\kappa))$. The proof size and the verification time are $O(\log^2 |\mathbb{L}|)$ and the prover time is $O(|\mathbb{L}| \log |\mathbb{L}|)$.*

Protocol 4 (Univariate Sumcheck). Let f be a degree k univariate polynomial on $\mathbb{F}$ with degree less than k and $\mathbb{H}, \mathbb{L}$ be a multiplicative coset of $\mathbb{F}$ such that $\mathbb{H} \subset \mathbb{L} \subset \mathbb{F}$ and $|\mathbb{L}| > k$. To prove $\mu = \sum_{a\in\mathbb{H}} f(a)$, a univariate sumcheck protocol has the following algorithms.

- $\mathsf{SC.com} \leftarrow \mathsf{SC.Commit}(f)$:
  1. $\mathcal{P}$ computes polynomial h such that $f(x) = g(x) + Z_{\mathbb{H}}(x) \cdot h(x)$. $\mathcal{P}$ evaluates of $f|_{\mathbb{L}}$ and $h|_{\mathbb{L}}$.
  2. $\mathcal{P}$ commits to the vectors using Merkle tree $\mathsf{root}_f \leftarrow \mathsf{MT.Commit}(f|_{\mathbb{L}})$ and $\mathsf{root}_h \leftarrow \mathsf{MT.Commit}(h|_{\mathbb{L}})$. $\mathcal{P}$ sends $\mathcal{V}$ $\mathsf{com} = (\mathsf{root}_f, \mathsf{root}_h)$.
- $\langle \mathsf{SC.Prove}(f), \mathsf{SC.Verify}(\mathsf{com}, \mu)\rangle$:
  1. Let $p(x) = \frac{|\mathbb{H}|\cdot f(x) - \mu - |\mathbb{H}|\cdot \mathbb{Z}_{\mathbb{H}}(x) h(x)}{|\mathbb{H}|\cdot x}$.
  2. $\mathcal{P}$ and $\mathcal{V}$ invoke the low degree test: $\langle \mathsf{LDT}.\mathcal{P}((f,h),p), \mathsf{LDT}.\mathcal{V}((k, k-|\mathbb{H}|), |\mathbb{H}|-1)\rangle(\mathbb{L})$. If the test fails, $\mathcal{V}$ aborts and output `0`. Otherwise, at then end of the test, $\mathcal{V}$ needs oracle access to κ points of f, h and p in $\mathbb{L}$. We denote their indices as $\mathcal{I}$.
  3. For each index $i \in \mathcal{I}$, $\mathcal{P}$ opens $\mathsf{MT.Open}(i, f|_{\mathbb{L}})$ and $\mathsf{MT.Open}(i, h|_{\mathbb{L}})$.
  4. $\mathcal{V}$ executes $\mathsf{MT.Verify}$ for all points opened by $\mathcal{P}$. If any verification fails, abort and output `0`.
  5. $\mathcal{V}$ completes the low degree test with these points. If all checks above pass, $\mathcal{V}$ outputs `1`.

3.3 Transparent Zero Knowledge Polynomial Delegation

In this section, we present our main construction, a zero knowledge verifiable polynomial delegation scheme without trusted setup. We first construct a VPD scheme that is correct and sound, then extend it to be zero knowledge. Our construction is inspired by the univariate sumcheck [BSCRSVW19] described in Section 3.2.4.

Our main idea is as follows. To evaluate an ℓ-variate polynomial f with variable degree d at point $t = (t_1, \ldots, t_\ell)$, we model the evaluation as the inner product between the vector of coefficients in f and the vector of all monomials in f evaluated at t. Formally speaking, let $N = |\mathcal{W}_{\ell,d}| = (d+1)^\ell$ be the number of possible monomials in an ℓ-variate polynomial with variable degree d, and let $c = (c_1, \ldots, c_N)$ be the coefficients of f in the order defined by $\mathcal{W}_{\ell,d}$ such that $f(x_1, \ldots, x_\ell) = \sum_{i=1}^N c_i W_i(x)$, where $W_i(x)$ is the i-th monomial in $\mathcal{W}_{\ell,d}$. Define the vector $T = (W_1(t), \ldots, W_N(t))$, then naturally the evaluation equals $f(t) = \sum_{i=1}^N c_i \cdot T_i$, the inner product of the two vectors. We then select a multiplicative coset $\mathbb{H}$ such that $|\mathbb{H}| = N$, [5] and interpolate vectors c and T to find the unique univariate polynomials that evaluate to c and T

[5] If such coset does not exist, we can pad N to the nearest number with a coset of that size, and pad vector T with 0s at the end.

Protocol 5 (Verifiable Polynomial Delegation). Let $\mathcal{F}$ be a family of ℓ-variate polynomial over $\mathbb{F}$ with variable-degree d and $N = (d+1)^\ell$.We use $\mathcal{W}_{\ell,d} = \{W_i(x_1, \ldots, x_\ell)\}_{i=1}^N$ to denote the collection of all monomials in $\mathcal{F}$. $r_f = \perp$ and we omit if in the algorithms.

- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda)$: Pick a hash function from the collision-resistant hash function family for Merkle tree. Find a multiplicative coset $\mathbb{H}$ of $\mathbb{F}$ such that $|\mathbb{H}| = (d+1)^\ell$. Find a multiplicative coset $\mathbb{L}$ of $\mathbb{F}$ such that $|\mathbb{L}| = O(|\mathbb{H}|) > 2|\mathbb{H}|$ and $\mathbb{H} \subset \mathbb{L} \subset \mathbb{F}$.
- $\mathsf{com} \leftarrow \mathsf{Commit}(f, \mathsf{pp})$: For a polynomial $f \in \mathcal{F}$ of the form $f(x) = \sum_{i=1}^N c_i W_i(x)$, find the unique univariate polynomial $l(x) : \mathbb{F} \to \mathbb{F}$ such that $l|_{\mathbb{H}} = (c_1, \ldots, c_N)$. $\mathcal{P}$ evaluates $l|_{\mathbb{L}}$ and runs $\mathsf{root}_l \leftarrow \mathsf{MT.Commit}(l|_{\mathbb{L}})$. Output $\mathsf{com} = \mathsf{root}_l$.
- $((\mu, \pi); \{0,1\}) \leftarrow \langle \mathsf{Open}(f), \mathsf{Verify}(\mathsf{com})\rangle(t, \mathsf{pp})$: This is an interactive protocol between $\mathcal{P}$ and $\mathcal{V}$.
  1. $\mathcal{P}$ computes $\mu = f(t)$ and sends it to $\mathcal{V}$.
  2. $\mathcal{P}$ evaluates $T = (W_1(t), \ldots, W_N(t))$. $\mathcal{P}$ finds the unique univariate polynomial $q(x) : \mathbb{F} \to \mathbb{F}$ such that $q|_{\mathbb{H}} = T$.
  3. $\mathcal{P}$ computes $l(x) \cdot q(x)$. $\mathcal{P}$ uniquely decomposes $l(x) \cdot q(x) = g(x) + Z_{\mathbb{H}}(x) \cdot h(x)$, where $Z_{\mathbb{H}}(x) = \prod_{a \in \mathbb{H}}(x - a)$ and the degrees of g and h are strictly less than $|\mathbb{H}|$ and $|\mathbb{H}| - 1$. $\mathcal{P}$ evaluates $h|_{\mathbb{L}}$ and runs $\mathsf{root}_h \leftarrow \mathsf{MT.Commit}(h|_{\mathbb{L}})$ and sends root_h to $\mathcal{V}$.
  4. Let $p(x) = \frac{|\mathbb{H}| \cdot l(x) \cdot q(x) - \mu - |\mathbb{H}| \cdot \mathbb{Z}_{\mathbb{H}}(x) h(x)}{|\mathbb{H}| \cdot x}$. $\mathcal{P}$ and $\mathcal{V}$ invoke a low degree test: $\langle \mathsf{LDT}.\mathcal{P}((l \cdot q, h), p), \mathsf{LDT}.\mathcal{V}((2|\mathbb{H}| - 1, |\mathbb{H}| - 1), |\mathbb{H}| - 1)\rangle(\mathbb{L})$. If the test fails, $\mathcal{V}$ aborts and output 0. Otherwise, at then end of the test, $\mathcal{V}$ needs oracle access to κ points of $l(x) \cdot q(x), h(x)$ and $p(x)$ at indices $\mathcal{I}$.
  5. For each index $i \in \mathcal{I}$, let a_i be the corresponding point in $\mathbb{L}$. $\mathcal{P}$ opens $(l(a_i), \pi_i^l) \leftarrow \mathsf{MT.Open}(i, l|_{\mathbb{L}})$ and $(h(a_i), \pi_i^h) \leftarrow \mathsf{MT.Open}(i, h|_{\mathbb{L}})$.
  6. $\mathcal{V}$ executes $\mathsf{MT.Verify}(\mathsf{root}_l, i, l(a_i), \pi_i^l)$ and $\mathsf{MT.Verify}(\mathsf{root}_h, i, h(a_i), \pi_i^h)$ for all points opened by $\mathcal{P}$. If any verification fails, abort and output 0.
  7. To complete the low degree test, $\mathcal{P}$ and $\mathcal{V}$ runs $\langle \mathsf{GKR}.\mathcal{P}, \mathsf{GKR}.\mathcal{V}\rangle(C, t)$, where circuit C computes the evaluations of $q|_{\mathbb{L}}$ and outputs the elements $q(a_i)$ for $i \in \mathcal{I}$ (see Figure 3.1). If any of the checks in GKR fails, $\mathcal{V}$ aborts and outputs 0.
  8. For each $i \in \mathcal{I}$, $\mathcal{V}$ computes $l(a_i) \cdot q(a_i)$. Together with $h(a_i)$, $\mathcal{V}$ completes the low degree test. If all checks above pass, $\mathcal{V}$ outputs 1.

on $\mathbb{H}$. We denote the polynomials as $l(x)$ and $q(x)$ such that $l|_{\mathbb{H}} = c$ and $q|_{\mathbb{H}} = T$. With these definitions, $f(t) = \sum_{i=1}^N c_i \cdot T_i = \sum_{a \in \mathbb{H}} l(a) \cdot q(a)$, which is the sum of the polynomial $l(x) \cdot q(x)$ on $\mathbb{H}$. The verifier can check the evaluation through a univariate sumcheck protocol with the prover. The detailed protocol is presented in step 1-4 of Protocol 5.

Up to this point, the construction for validating the inner product between a vector committed by $\mathcal{P}$ and a public vector is similar to and simpler than the protocols to check linear constraints proposed in [AHIV17; BSCRSVW19]. However, naively applying the univariate sumcheck protocol incurs a linear overhead for the verifier. This is because as described in Section 3.2.4, at the end of the univariate sumcheck, due to the low

degree test, the verifier needs oracle access to the evaluations of $l(x) \cdot q(x)$ at κ points on $\mathbb{L}$, a superset of $\mathbb{H}$. As $l(x)$ is defined by c, i.e. the coefficients of f, the prover can commit to $l|_{\mathbb{L}}$ at the beginning of the protocol, and opens to points the verifier queries with their Merkle tree proofs. $q(x)$, however, is defined by the public vector T, and the verifier has to evaluate it locally, which takes linear time. This is the major reason why the verification time in the zero knowledge proof schemes for generic arithmetic circuits in [AHIV17; BSCRSVW19] is linear in the size of the circuits.

Reducing the verification time. In this paper, we propose an approach to reduce the cost of the verifier to poly-logarithmic for VPD. We observe that in our construction, though the size of T and $q(x)$ is linear in N, it is defined by only $\ell = O(\log N)$ values of the evaluation point t. This means that the oracle access of κ points of $q(x)$ can be modeled as a function that: (1) Takes t as input, evaluates all monomials $W_i(t)$ for all $W_i \in \mathcal{W}_{\ell,d}$ as a vector T; (2) Extrapolates the vector T to find polynomial $q(x)$, and evaluates $q(x)$ on $\mathbb{L}$; (3) Outputs κ points of $q|_{\mathbb{L}}$ chosen by the verifier. Although the size of the function modeled as an arithmetic circuit is $\Omega(N)$ with $O(\log N)$ depth, and the size of its input and output is only $O(\log N + \kappa)$. Therefore, instead of evaluating the function locally, the verifier can delegate this computation to the prover, and validate the result using the GKR protocol, as presented in Section 3.2.3. In this way, we eliminate the linear overhead to evaluate these points locally, making the verification time of the overall VPD protocol poly-logarithmic. The formal protocol is presented in Protocol 5.

To avoid any asymptotic overhead for the prover, we also design an efficient layered arithmetic circuit for the function mentioned above. The details of the circuit are presented in Figure 3.1. In particular, in the first part, each value t_i in the input t is raised to powers of $0, 1, \ldots, d$. Then they are expanded to T, the evaluations of all monomials in $\mathcal{W}_{\ell,d}$, by multiplying one t_i at a time through a $(d+1)$-ary tree. The size of this part is $O(N) = O((d+1)^{\ell})$ and the depth is $O(\log d + \ell)$. In the second part, the polynomial $q(x)$ and the vector $q|_{\mathbb{L}}$ is computed from T directly using FFTs. We first construct a circuit for an inverse FFT to compute the coefficients of polynomial $q(x)$ from its evaluations T. Then we run an FFT to evaluate $q|_{\mathbb{L}}$ from the coefficients of $q(x)$. We implement FFT and IFFT using the Butterfly circuit [CLRS09]. The size of the circuit is $O(N \log N)$ and the depth is $O(\log N)$. Finally, κ points are selected from $q|_{\mathbb{L}}$. As the whole delegation of the GKR protocol is executed at the end in Protocol 5 after these points being fixed by the verifier, the points to output are directly hard-coded into the circuit with size $O(\kappa)$ and depth 1. No heavy techniques for random accesses in the circuit is needed. Therefore, the whole circuit is of size $O(N \log N)$ and depth $O(\log N)$, with ℓ inputs and κ outputs.

Theorem 3.3.1. *Protocol 5 is a verifiable polynomial delegation protocol that is complete and sound under Definition 4.4.2.*

Proof. **Completeness.** By the definition of $l(x)$ and $q(x)$, if $\mu = f(t)$, then $\mu = \sum_{a \in H} l(a) \cdot q(a) = \sum_{a \in H} g(a) = g(0) \cdot |\mathbb{H}|$ by Lemma 3.2.6. Thus, $p(x) = \frac{|\mathbb{H}| \cdot l(x) \cdot q(x) - |\mathbb{H}| \cdot \mathbb{Z}_{\mathbb{H}}(x) h(x) - \mu}{|\mathbb{H}| \cdot x} = \frac{g(x) - g(0)}{x}$, which is in $RS[\mathbb{L}, |\mathbb{H}| - 1]$. The rest follows the completeness of the LDT protocol and the GKR protocol.
Soundness. Let $\varepsilon_{\mathsf{LDT}}, \varepsilon_{\mathsf{MT}}, \varepsilon_{\mathsf{GKR}}$ be the soundness error of the LDT, Merkle tree and GKR protocols. There are two cases for a malicious prover $\mathcal{P}$.
Case 1: $\nexists l^* \in RS[\mathbb{L}, |\mathbb{H}| + 1]$ such that $\mathsf{com} = \mathsf{MT.Commit}(l^*|_{\mathbb{L}})$, i.e. com is not a valid commitment.

- By the check in step 6, if com is not a valid Merkle tree root, the verification passes with probability less than $\varepsilon_{\mathsf{MT}}$.
- If $\exists l^{**} \notin RS[\mathbb{L}, |\mathbb{H}| + 1]$ such that $\mathsf{com} \leftarrow \mathsf{MT.Commit}(l^{**}|_{\mathbb{L}})$, if the points v_i^* opened by $\mathcal{P}$ in step 5 $v_i^* \neq l^{**}(a_i)$ for some i, the verification passes with probability no more than $\varepsilon_{\mathsf{MT}}$.

Input: $t = (t_1, \ldots, t_\ell)$
Output: $q()$

1. Computing vector $T = (W_1(t), \ldots, W_N(t))$:
   - Compute $(t_i^0, t_i^1, \ldots, t_i^d)$ for $i = 1, \ldots, \ell$.
   - Initialize vector $T_0 = (1)$.
   - For $i = 1, \ldots, \ell$:
     $T_i = (t_i^0 \cdot T_{i-1}, \ldots, t_i^d \cdot T_{i-1})$, where " $\cdot$ " here is scalar multiplication between a number and a vector and "," means concatenation. Set $T = T_\ell$.
2. Computing $q|_{\mathbb{L}}$:
   - $q|_{\mathbb{L}} = \mathsf{FFT}(\mathsf{IFFT}(T, \mathbb{H}), \mathbb{L})$
3. Outputting evaluations indexed by I_q:

Figure 3.1: Arithmetic circuit C computing evaluations of $q(x)$ at κ points in $\mathbb{L}$ indexed by $\mathcal{I}$.

- If the output q_i^* returned by $\mathcal{P}$ in step 7 is $q_i^* \neq q(a_i)$ for some i, the verification passes with probability less than $\varepsilon_{\mathsf{GKR}}$.
- Otherwise, as $l^{**}(x) \cdot q(x) \notin RS[\mathbb{L}, 2|\mathbb{H}| + 1]$, by the checks of LDT in step 4, the verification passes with probability no more than $\varepsilon_{\mathsf{LDT}}$.

Case 2: $\exists l^* \in RS[\mathbb{L}, |\mathbb{H}| + 1]$ such that $\mathsf{com} = \mathsf{MT.Commit}(l^*|_{\mathbb{L}})$. Let $c^* = l^*|_{\mathbb{H}}$ and $f^*(x) = \sum_{i=1}^{N} c_i^* W_i(x)$, then $\mathsf{com} = \mathsf{Commit}(f^*, \mathsf{pp})$. Suppose $\mu^* \neq f^*(t)$, then $\mu^* \neq \sum_{a \in \mathbb{H}} l^*(a) q(a)$. Then by Lemma 3.2.6, for all $h \in RS[\mathbb{L}, |\mathbb{H}|+1]$, $p^* \notin RS[\mathbb{L}, |\mathbb{H}|-1]$, as $\sum_{a \in \mathbb{H}}(p^*(a) \cdot a) = \sum_{a \in \mathbb{H}} \frac{|\mathbb{H}| \cdot l^*(a) \cdot q(a) - \mu^*}{|\mathbb{H}|} = \sum_{a \in \mathbb{H}}(l^*(a) \cdot q(a)) - \mu^* \neq 0$. Therefore,

- Similar to case 1, if the commitment in step 3 is not a valid Merkle tree root, or the points opened by $\mathcal{P}$ in step 5 are inconsistent with h or l^*, the verification passes with probability no more than $\varepsilon_{\mathsf{MT}}$.
- If the output q_i^* returned by $\mathcal{P}$ in step 7 $q_i^* \neq q(a_i)$ for some i, the verification passes with probability no more than $\varepsilon_{\mathsf{GKR}}$.
- Otherwise, as $l^* \cdot q \in RS[\mathbb{L}, 2|\mathbb{H}| + 1]$, either $h \notin RS[\mathbb{L}, |\mathbb{H}| + 1]$ or $p \notin RS[\mathbb{L}, |\mathbb{H}| - 1]$ as explained above. By the check in step 4, the verification passes with probability no more than $\varepsilon_{\mathsf{LDT}}$.

By the union bound, the probability of the event of a malicious prover is no more than $O(\varepsilon_{\mathsf{LDT}} + \varepsilon_{\mathsf{MT}} + \varepsilon_{\mathsf{GKR}})$. As stated in Section 5.2, $\varepsilon_{\mathsf{LDT}} = O(\frac{|\mathbb{L}|}{|\mathbb{F}|}) + \mathsf{negl}(\kappa)$, $\varepsilon_{\mathsf{GKR}} = O(\frac{\log^2 N}{|\mathbb{F}|})$ and $\varepsilon_{\mathsf{MT}} = \mathsf{negl}(\lambda)$. Therefore, with proper choice of parameters, the probability is $\leq \mathsf{negl}(\lambda)$. □

Efficiency. The running time of Commit is $O(N \log N)$. C in step 7 is a regular circuit with size $O(N \log N)$, depth $O(\ell + \log d)$ and size of input and output $O(\ell + \kappa)$. By Lemma 3.2.4 and 3.2.8, the prover time is $O(N \log N)$, the proof size and the verification time are $(\log^2 N)$.

Extending to other ZKP schemes. We notice that our technique can be potentially applied to generic zero knowledge proof schemes in [AHIV17; BSCRSVW19] to improve the verification time for circuits/constraint systems with succinct representation. As mentioned previously, the key step that introduces linear verification time in these schemes is to check a linear constraint system, i.e., $y = \mathbf{A}w$, where w is a vector of all values on the wires of the circuit committed by the prover, and $\mathbf{A}$ is a public matrix derived from the circuit such that $\mathbf{A}w$ gives a vector of left inputs to all multiplication gates in the circuit. (This check is executed 2 more times to also give right inputs and outputs.) To check the relationship, it is turned into a vector inner product $\mu = ry = r\mathbf{A} \cdot w$ by multiplying both sides by a random vector r. Similar to our naive protocol to check inner product, the verification time is linear in order to evaluate the polynomial defined by $r\mathbf{A}$ at κ points. With our new protocol, if the circuit can be represented succinctly in sublinear or logarithmic space, $\mathbf{A}$ can be computed by a function with sublinear or logarithmic number of inputs. We can use the GKR protocol to delegate the computation of $r\mathbf{A}$ and the subsequent evaluations to the prover in a similar way as in our construction, and the verification time will only depend on the space to represent the circuit, but not on the total size of the circuit. This is left as a future work.

3.3.1 Achieving Zero Knowledge

Our VPD protocol in Protocol 5 is not zero knowledge. Intuitively, there are two places that leak information about the polynomial f: (1) In step 6 of Protocol 5, $\mathcal{P}$ opens evaluations of $l(x)$, which is defined by the coefficients of f; (2) In step 4, $\mathcal{P}$ and $\mathcal{V}$ execute low degree tests on $(l(x) \cdot q(x), h(x)), p(x)$ and the proofs of LDT reveal information about the polynomials, which are related to f.

To make the protocol zero knowledge, we take the standard approaches proposed in [AHIV17; BSCRSVW19]. To eliminate the former leakage of queries on $l(x)$, the prover picks a random degree κ polynomial $r(x)$ and masks it as $l'(x) = l(x) + Z_{\mathbb{H}}(x) \cdot r(x)$, where as before, $Z_{\mathbb{H}}(x) = \prod_{a \in \mathbb{H}}(x - a)$. Note here that $l'(a) = l(a)$ for $a \in \mathbb{H}$, yet any κ evaluations of $l'(x)$ outside $\mathbb{H}$ do not reveal any information about $l(x)$ because of the masking polynomial $r(x)$. The degree of $l'(x)$ is $|\mathbb{H}| + \kappa$, and we denote domain $\mathbb{U} = \mathbb{L} - \mathbb{H}$.

To eliminate the latter leakage, $\mathcal{P}$ samples a random polynomial $s(x)$ of the same degree as $l'(x) \cdot q(x)$, sends $S = \sum_{a \in \mathbb{H}} s(a)$ to $\mathcal{V}$ and runs the univariate sumcheck protocol on their random linear combination: $\alpha\mu + S = \sum_{a \in \mathbb{H}}(\alpha l'(x) \cdot q(x) + s(x))$ for a random $\alpha \in \mathbb{F}$ chosen by $\mathcal{V}$. This ensures that both μ and S are correctly computed because of the random linear combination and the linearity of the univariate sumcheck, while leaking no information about $l'(x) \cdot q(x)$ during the protocol, as it is masked by $s(x)$.

One advantage of our construction is that the GKR protocol used to compute evaluations of $q(x)$ in step 7 of Protocol 5 remains unchanged in the zero knowledge version of the VPD. This is because $q(x)$ and its evaluations are independent of the polynomial f or any prover's secret input. Therefore, it suffices to apply the plain version of GKR without zero knowledge, avoiding any expensive cryptographic primitives.

The full protocol for our zkVPD is presented in Protocol 6. Note that all the evaluations are on $\mathbb{U} = \mathbb{L} - \mathbb{H}$ instead of $\mathbb{L}$, as evaluations on $\mathbb{H}$ leaks information about the original $l(x)$. $s(x)$ is also committed and opened using Merkle tree for the purpose of correctness and soundness. The efficiency of our zkVPD protocol is asymptotically the same as our VPD protocol in Protocol 5, and the concrete overhead in practice is also small. We have the following theorem:

Theorem 3.3.2. *Protocol 6 is a zero knowledge verifiable polynomial delegation scheme by Definition 4.4.2.*

Proof. **Completeness.** It follows the completeness of Protocol 5.

Protocol 6 (Zero Knowledge Verifiable Polynomial Delegation). Let $\mathcal{F}$ be a family of ℓ-variate polynomial over $\mathbb{F}$ with variable-degree d and $N = (d+1)^\ell$.We use $\mathcal{W}_{\ell,d} = \{W_i(x_1, \ldots, x_\ell)\}_{i=1}^N$ to denote the collection of all monomials in $\mathcal{F}$.

- $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$: Same as KeyGen in Procotol 5. Define $\mathbb{U} = \mathbb{L} - \mathbb{H}$.
- $\mathsf{com} \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$: For a polynomial $f \in \mathcal{F}$ of the form $f(x) = \sum_{i=1}^N c_i W_i(x)$, find the unique univariate polynomial $l(x) : \mathbb{F} \to \mathbb{F}$ such that $l|_{\mathbb{H}} = (c_1, \ldots, c_N)$. $\mathcal{P}$ samples a polynomial $r(x)$ with degree κ randomly and sets $l'(x) = l(x) + Z_{\mathbb{H}}(x) \cdot r(x)$, where $Z_{\mathbb{H}}(x) = \prod_{a\in\mathbb{H}}(x-a)$. $\mathcal{P}$ evaluates $l'|_{\mathbb{U}}$ and runs $\mathsf{root}_{l'} \leftarrow \mathsf{MT.Commit}(l'|_{\mathbb{U}})$. Output $\mathsf{com} = \mathsf{root}_{l'}$.
- $((\mu, \pi); \{0,1\}) \leftarrow \langle \mathsf{Open}(f, r_f), \mathsf{Verify}(\mathsf{com})\rangle(t, \mathsf{pp})$: This is an interactive protocol between $\mathcal{P}$ and $\mathcal{V}$. It replaces the univariate sumscheck on $l(x) \cdot q(x)$ by $l'(x) \cdot q(x) + \alpha s(x)$ and $\mathbb{L}$ by $\mathbb{U}$ in Protocol 5.
  1. $\mathcal{P}$ computes $\mu = f(t)$ and sends it to $\mathcal{V}$.
  2. $\mathcal{P}$ evaluates $T = (W_1(t), \ldots, W_N(t))$. $\mathcal{P}$ finds the unique univariate polynomial $q(x) : \mathbb{F} \to \mathbb{F}$ such that $q|_{\mathbb{H}} = T$.
  3. $\mathcal{P}$ samples randomly a degree $2|\mathbb{H}| + \kappa - 1$ polynomial $s(x)$. $\mathcal{P}$ sends $\mathcal{V}$ $S = \sum_{a\in\mathbb{H}} s(a)$ and $\mathsf{root}_s \leftarrow \mathsf{MT.Commit}(s|_{\mathbb{U}})$.
  4. $\mathcal{V}$ picks $\alpha \in \mathbb{F}$ randomly and sends it to $\mathcal{P}$.
  5. $\mathcal{P}$ computes $\alpha l'(x) \cdot q(x) + s(x)$. $\mathcal{P}$ uniquely decomposes it as $g(x) + Z_{\mathbb{H}}(x) \cdot h(x)$, where the degrees of g and h are strictly less than $|\mathbb{H}|$ and $|\mathbb{H}| + \kappa$. $\mathcal{P}$ evaluates $h|_{\mathbb{U}}$ and sends $\mathsf{root}_h \leftarrow \mathsf{MT.Commit}((h|_{\mathbb{U}})$ to $\mathcal{V}$.
  6. Let $p(x) = \frac{|\mathbb{H}|\cdot(\alpha l'(x)\cdot q(x) + s(x)) - (\alpha\mu + S) - |\mathbb{H}|\cdot Z_{\mathbb{H}}(x) h(x)}{|\mathbb{H}|\cdot x}$. $\mathcal{P}$ and $\mathcal{V}$ invoke the low degree test: $\langle \mathsf{LDT}.\mathcal{P}((l' \cdot q, h, s), p), \mathsf{LDT}.\mathcal{V}((2|\mathbb{H}| + \kappa, |\mathbb{H}| + \kappa, 2|\mathbb{H}| + \kappa), |\mathbb{H}| - 1)\rangle(\mathbb{U})$. If the test fails, $\mathcal{V}$ aborts and output 0. Otherwise, at the end of the test, $\mathcal{V}$ needs oracle access to κ points of $l'(x) \cdot q(x), h(x), s(x)$ and $p(x)$ at indices $\mathcal{I}$.
  7. For each index $i \in \mathcal{I}$, let a_i be the corresponding point in $\mathbb{U}$. $\mathcal{P}$ opens $(l'(a_i), \pi_i^{l'}) \leftarrow \mathsf{MT.Open}(i, l'|_{\mathbb{U}})$, $(h(a_i), \pi_i^h) \leftarrow \mathsf{MT.Open}(i, h|_{\mathbb{U}})$ and $(s(a_i), \pi_i^s) \leftarrow \mathsf{MT.Open}(i, s|_{\mathbb{U}})$.
  8. $\mathcal{V}$ executes $\mathsf{MT.Verify}(\mathsf{root}_{l'}, i, l'(a_i), \pi_i^{l'})$, $\mathsf{MT.Verify}(\mathsf{root}_h, i, h(a_i), \pi_i^h)$ and $\mathsf{MT.Verify}(\mathsf{root}_s, i, s(a_i), \pi_i^s)$ for all points opened by $\mathcal{P}$. If any verification fails, abort and output 0.
  9. To complete the low degree test, $\mathcal{P}$ and $\mathcal{V}$ runs $\langle \mathsf{GKR}.\mathcal{P}, \mathsf{GKR}.\mathcal{V}\rangle(C, t)$, where circuit C computes the evaluations of $q|_{\mathbb{U}}$ and outputs the elements $q(a_i)$ for $i \in \mathcal{I}$. If any of the checks in GKR fails, $\mathcal{V}$ aborts and outputs 0.
  10. For each $i \in \mathcal{I}$, $\mathcal{V}$ computes $l'(a_i) \cdot q(a_i)$. Together with $h(a_i)$ and $s(a_i)$, $\mathcal{V}$ completes the low degree test. If all checks above pass, $\mathcal{V}$ outputs 1.

Soundness. It follows the soundness of Protocol 5 and the random linear combination. In particular, in Case 2 of the proof of Theorem 3.3.1, if $\exists l'^* \in RS[\mathbb{L}, |\mathbb{H}| + \kappa + 1]$, it can always be uniquely decomposed as $l^*(x) = l'^*(x) - Z_{\mathbb{H}}(x) r^*(x)$ such that $\sum_{a\in\mathbb{H}} l'^*(a) = \sum_{a\in\mathbb{H}} l^*(a)$ and the degree of $l^*(x)$ is $|\mathbb{H}|$ and the

degree of $r(x)$ is κ. If $\mu^* \neq \mu = \sum_{a\in\mathbb{H}}(l^*(a)\cdot q(a)) = \sum_{a\in\mathbb{H}}(l'^*(a)\cdot q(a))$, let $S^* = \sum_{a\in\mathbb{H}} s^*(a)$ where $s^*(x)$ is committed by $\mathcal{P}$ in step 5, then $\sum_{a\in\mathbb{H}}(\alpha l'^*(a)\cdot q(a) + s^*(a)) = \alpha\mu^* + S^* = \alpha\mu + S$ if and only if $\alpha = \frac{S-S^*}{\mu^*-\mu}$, which happens with probability $1/|\mathbb{F}|$. The probability of other cases are the same as the proof of Theorem 3.3.1, and we omit the details here.

Zero knowledge. The simulator is given in Figure 3.2.

To prove zero knowledge, l'_{sim} in $\mathcal{S}_1$ and l' in zkVPD.Commit are both uniformly distributed. In $\mathcal{S}_2$, steps 1, 2 and 9 are the same as the real world in Protocol 6. No message is sent in steps 4, 8 and 10.

In step 3 and 7, s_{sim} and s are both randomly selected and their commitments and evaluations are indistinguishable. As $r(x)$ is a degree-κ random polynomial in the real world in Protocol 6, κ evaluations of $l'(x)$ opened in step 7 are independent and randomly distributed, which is indistinguishable from step 7 of $\mathcal{S}_2$ in the ideal world. Finally, in step 7 of the ideal world, $\mathcal{V}^*$ receives κ evaluations of h_{sim} at point indexed by $\mathcal{I}$. Together with $l'_{\mathsf{sim}}\cdot q$ and s_{sim}, by Lemma 3.2.7, the view of steps 5-7 simulated by $\mathsf{LDT}.\mathcal{S}$ is indistinguishable from the real world with $h, l'\cdot q$ and s, which completes the proof. □

Our zkVPD protocol is also a proof of knowledge. Here we give the formal definition of knowledge soundness of a zkVPD protocol in addition to Definition 4.4.2 and prove that our protocol has knowledge soundness.

Knowledge Soundness. For any PPT adversary $\mathcal{A}$, there exists a PPT extractor $\mathcal{E}$ such that given access to the random tape of $\mathcal{A}$, for every $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$, the following probability is $\mathsf{negl}(\lambda)$:

$$\Pr\begin{bmatrix}(\mathsf{com}^*, t) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}),\\ ((y^*, \pi^*); 1) \leftarrow \langle \mathcal{A}(), \mathsf{zkVPD.Verify}(\mathsf{com}^*)\rangle(t, \mathsf{pp}),\\ (f, r_f) \leftarrow \mathcal{E}(1^\lambda, \mathsf{pp}) :\\ \mathsf{com}^* \neq \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp}) \vee f(t) \neq y^*\end{bmatrix}$$

Our zkVPD protocol is a proof of knowledge in the random oracle model because of the extractability of Merkle tree, as proven in [Val08; BSCS16]. Informally speaking, given the root and sufficiently many authentication paths, there exists a PPT extractor that reconstructs the leaves with high probability. Additionally, in our protocol the leaves are RS encoding of the witness, which can be efficiently decoded by the extractor. We give a proof similar to [Val08; BSCS16] below.

Proof. Suppose the Merkle tree in our protocol is based on a random oracle $\mathcal{R} : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda$. We could construct a polynomial extractor $\mathcal{E}$ with the same random type of $\mathcal{A}$ working as follows:

Simulate $\mathcal{A}^{\mathcal{R}}$, and let $q_1, q_2, \cdots, q_t$ be the queries made by $\mathcal{A}$ to $\mathcal{R}$ in the order they are made where duplicates omitted. Define $q_i \in \mathcal{R}(q_j)$ if the first λ bits or the last λ bits of q_i is $\mathcal{R}(q_j)$. If there exist some $i \neq j$, $\mathcal{R}(q_i) = \mathcal{R}(q_j)$, or some $i \leq j$ $q_i \in R(q_j)$, $\mathcal{E}$ aborts and outputs a random string as (f, r_f).

$\mathcal{E}$ constructs an acyclic directed graph G according to the query set $Q = \{q_1, q_2, \cdots, q_t\}$. There is an edge from q_i to q_j in G if and only if $q_i \in R(q_j)$. The outdegree of each node is at most 2. When $\mathcal{A}$ generates $\mathsf{root}_{l'}$ in step 2 of Protocol 6, if $\mathsf{root}_{l'}$ does not equal $\mathcal{R}(q)$ for some $q \in Q$, $\mathcal{E}$ aborts and outputs a random string as (f, r_f), otherwise we suppose $\mathcal{R}(q_r) = \mathsf{root}_{l'}$. If a verification path of π^* is not valid, $\mathcal{E}$ aborts and outputs a random string as (f, r_f).

Since $\mathcal{E}$ knows the correct depth of the Merkle tree, it could read off all leaf strings with this depth from the binary tree rooted at q_r. If there exists missing leaf, $\mathcal{E}$ aborts and outputs a random string as (f, r_f), otherwise, it concatenates these leaf strings as $w' = l'|_{\mathbb{U}}$, and decodes $w = l'|_{\mathbb{H}}$ using an efficient

- $\mathsf{com} \leftarrow \mathcal{S}_1(1^\lambda, \mathsf{pp})$: Pick a random polynomial $l'_{\mathsf{sim}}(x) \in RS[\mathbb{L}, |\mathbb{H}| + \kappa + 1]$. Evaluate $l'_{\mathsf{sim}}|_{\mathbb{U}}$ and output $\mathsf{root}_{l'_{\mathsf{sim}}} \leftarrow \mathsf{MT.Commit}(l'_{\mathsf{sim}}|_{\mathbb{U}})$.
- $\mathcal{S}_2(t, \mathsf{pp})$:
  1. Given oracle access to $\mu = f(t)$, send it to $\mathcal{V}^*$.
  2. Evaluate $T = (W_1(t), \ldots, W_N(t))$. Find the unique univariate polynomial $q(x) : \mathbb{F} \to \mathbb{F}$ such that $q|_{\mathbb{H}} = T$.
  3. Pick a degree $2|\mathbb{H}| + \kappa - 1$ polynomial $s_{\mathsf{sim}}(x)$ randomly. Send $\mathcal{V}$ $S_{\mathsf{sim}} = \sum_{a \in \mathbb{H}} s_{\mathsf{sim}}(a)$ and $\mathsf{root}_{s_{\mathsf{sim}}} \leftarrow \mathsf{MT.Commit}(s_{\mathsf{sim}}|_{\mathbb{U}})$.
  4. Receive $\alpha \in \mathbb{F}$ from $\mathcal{V}$.
  5. Let $\mathsf{LDT}.\mathcal{S}$ be the simulator the LDT protocol described in Section 3.2.4. Given the random challenges $\mathcal{I}$ of $\mathcal{V}^*$, call $\mathsf{LDT}.\mathcal{S}$ to generate $p^*(x) \in RS[\mathbb{L}, |\mathbb{H}| - 1]$. For each point a_i in $\mathcal{I}$, compute h_i such that $p^*(a_i) = \frac{|\mathbb{H}| \cdot (\alpha l'_{\mathsf{sim}}(a_i) \cdot q(a_i) + s_{\mathsf{sim}}(a_i)) - (\alpha\mu + S_{\mathsf{sim}}) - |\mathbb{H}| \cdot \mathbb{Z}_{\mathbb{H}}(a_i) h_i}{|\mathbb{H}| \cdot a_i}$. Interpolate h_i to get polynomial h_{sim} and sends $\mathsf{root}_{h_{\mathsf{sim}}} \leftarrow \mathsf{MT.Commit}((h_{\mathsf{sim}}|_{\mathbb{U}})$ to $\mathcal{V}^*$.
  6. Call $\mathsf{LDT}.\mathcal{S}$ to simulate the view of the low degree test $\mathsf{LDT}.\mathcal{S}^{\mathcal{V}^*}$.
  7. For each index $i \in \mathcal{I}$, let a_i be the corresponding point in $\mathbb{U}$. $\mathcal{P}$ opens $(l'_{\mathsf{sim}}(a_i), \pi_i^{l'_{\mathsf{sim}}}) \leftarrow \mathsf{MT.Open}(i, l'_{\mathsf{sim}}|_{\mathbb{U}})$, $(h_i, \pi_i^{h_{\mathsf{sim}}}) \leftarrow \mathsf{MT.Open}(i, h_{\mathsf{sim}}|_{\mathbb{U}})$ and $(s_{\mathsf{sim}}(a_i), \pi_i^{s_{\mathsf{sim}}}) \leftarrow \mathsf{MT.Open}(i, s_{\mathsf{sim}}|_{\mathbb{U}})$.
  8. Wait $\mathcal{V}^*$ to validate the points.
  9. Run $\langle \mathsf{GKR}.\mathcal{P}, \mathsf{GKR}.\mathcal{V} \rangle(C, t)$ with $\mathcal{V}^*$, where circuit C computes the evaluations of $q|_{\mathbb{U}}$ and outputs the elements $q(a_i)$ for $i \in \mathcal{I}$.
  10. Wait $\mathcal{V}^*$ for validation.

Figure 3.2: Simulator $\mathcal{S}$ of the zkVPD protocol.

Reed–Solomon decoding algorithm (such as Berlekamp–Welch). $\mathcal{E}$ could easily output (f, r_f) according to w.

Let E_1 denote the event $((y^*, \pi^*); 1) \leftarrow \langle \mathcal{A}(), \mathsf{zkVPD.Verify}(\mathsf{com}^*) \rangle(t, \mathsf{pp})$ and E_2 denote the event $\mathsf{com}^* \neq \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp}) \vee f(t) \neq y^*$, next we show $\Pr[E_1 \wedge E_2] \leq \mathsf{negl}(\lambda)$.

The probability that $\mathcal{E}$ aborts before constructing the graph G is $\mathsf{negl}(\lambda)$ because of the collision-resistant property of the random oracle. If some node on a verification path(possibly including the root) of the proof π^* does not lie in the graph G, $\mathcal{A}$ has to guess the value to construct a valid verification path, which propability is also $\mathsf{negl}(\lambda)$ since $\mathcal{R}$ is noninvertible. Additionally, if one leaf of the tree is missing, then $\mathcal{V}$ will be convinced with probability $\mathsf{negl}(\lambda)$ once it queries this leaf. And the probability this leaf is not be queried by $\mathcal{V}$ is at most $(1 - \frac{1}{|\mathbb{U}|})^\kappa = \mathsf{negl}(\lambda)$ as $\kappa = O(\lambda)$.

If $\mathcal{E}$ does not abort, it could always extract some (f, r_f) satisfying $\mathsf{com}^* = \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp})$.

Protocol 7 (Our Zero Knowledge Argument). Let λ be the security parameter, $\mathbb{F}$ be a prime field. Let $C : \mathbb{F}^n \to \mathbb{F}$ be a layered arithmetic circuit over $\mathbb{F}$ with D layers, input x and witness w such that $|x|+|w| \leq n$ and $1 = C(x; w)$.

- $\mathcal{G}(1^\lambda)$: set pp as $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$.
- $\langle \mathcal{P}(\mathsf{pp}, w), \mathcal{V}(\mathsf{pp}) \rangle(\mathsf{in})$:

  1. $\mathcal{P}$ selects a random bivariate polynomial R_D. $\mathcal{P}$ commits to the witness of C by sending $\mathsf{com}_D \leftarrow \mathsf{zkVPD.Commit}(\dot{V}_D, \mathsf{pp})$ to $\mathcal{V}$, where $\dot{V}_D$ is defined by Equation 4.12.
  2. $\mathcal{P}$ randomly selects polynomials $R_i : \mathbb{F}^2 \to \mathbb{F}$ and $\delta_i : \mathbb{F}^{2s_{i+1}+1} \to \mathbb{F}$ for $i = 0, \ldots, D-1$. $\mathcal{P}$ commits to these polynomials by sending $\mathsf{com}_{i,1} \leftarrow \mathsf{zkVPD.Commit}(R_i, \mathsf{pp})$ and $\mathsf{com}_{i,2} \leftarrow \mathsf{zkVPD.Commit}(\delta_i, \mathsf{pp})$ to $\mathcal{V}$. $\mathcal{P}$ also reveals R_0 to $\mathcal{V}$, as V_0 is defined by out and is known to $\mathcal{V}$.
  3. $\mathcal{V}$ evaluates $\dot{V}_0(u^{(0)})$ and $\dot{V}_0(v^{(0)})$ for randomly chosen $u^{(0)}, v^{(0)} \in \mathbb{F}^{s_0}$.
  4. For $i = 0, \ldots, D-1$:
     a) $\mathcal{P}$ sends $H_i = \sum_{x,y \in \{0,1\}^{s_{i+1}}, z \in \{0,1\}} \delta_i(x, y, z)$ to $\mathcal{V}$.
     b) $\mathcal{V}$ picks $\alpha_i, \beta_i, \gamma_i$ randomly in $\mathbb{F}$.
     c) $\mathcal{V}$ and $\mathcal{P}$ execute a sumcheck protocol on Equation 3.4. At the end of the sumcheck, $\mathcal{V}$ receives a claim of f_i' at point $u^{(i+1)}, v^{(i+1)} \in \mathbb{F}^{s_{i+1}}, g_i \in \mathbb{F}$ selected randomly by $\mathcal{V}$.
     d) $\mathcal{P}$ opens $R_i(u^{(i)}, g_i)$, $R_i(v^{(i)}, g_i)$ and $\delta_i(u^{(i+1)}, v^{(i+1)}, g_i)$ using $\mathsf{zkVPD.Open}$. $\mathcal{P}$ sends $\dot{V}_0(u^{(i+1)})$ and $\dot{V}_0(v^{(i+1)})$ to $\mathcal{V}$.
     e) $\mathcal{V}$ validates $R_i(u^{(i)}, g_i)$, $R_i(v^{(i)}, g_i)$ and $\delta_i(u^{(i+1)}, v^{(i+1)}, g_i)$ using $\mathsf{zkVPD.Verify}$. If any of them outputs 0, abort and output $\mathtt{0}$.
     f) $\mathcal{V}$ checks the claim of f_i' using $R_i(u^{(i)}, g_i)$, $R_i(v^{(i)}, g_i)$, $\delta_i(u^{(i+1)}, v^{(i+1)}, g_i)$, $\dot{V}_0(u^{(i+1)})$ and $\dot{V}_0(v^{(i+1)})$. If it fails, output $\mathtt{0}$.
  5. $\mathcal{P}$ runs $(y_1, \pi_1) \leftarrow \mathsf{zkVPD.Open}(\dot{V}_D, u^{(D)}, \mathsf{pp})$, $(y_2, \pi_2) \leftarrow \mathsf{zkVPD.Open}(\dot{V}_D, v^{(D)}, \mathsf{pp})$ and sends y_1, π_1, y_2, π_2 to $\mathcal{V}$.
  6. $\mathcal{V}$ runs $\mathsf{Verify}(\pi_1, y_1, \mathsf{com}_D, u^{(D)}, \mathsf{pp})$ and $\mathsf{Verify}(\pi_2, y_2, \mathsf{com}_D, v^{(D)}, \mathsf{pp})$ and output $\mathtt{0}$ 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. If all checks above pass, $\mathcal{V}$ output $\mathtt{1}$.

In this case, $\mathcal{V}$ accepts the statement with probability $\mathsf{negl}(\lambda)$ if $f(t) \neq y^*$ according to the soundness of zkVPD.

Therefore, $\Pr[E_1 \wedge E_2] = \Pr[E_1 \wedge E_2 | \mathcal{E} \text{ aborts}] + \Pr[E_1 \wedge E_2 | \mathcal{E} \text{ does not abort}] \leq \Pr[E_1 | \mathcal{E} \text{ aborts}] + \Pr[E_1 \wedge E_2 | \mathcal{E} \text{ does not abort}] \leq \mathsf{negl}(\lambda) + \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$ □

3.4 Zero Knowledge Argument

Following the framework of [XZZPS19a], we can instantiate the zkVPD in Protocol 3 with our new construction of transparent zkVPD in Protocol 6 to obtain a zero knowledge argument of knowledge scheme for layered arithmetic circuits without trusted setup. In this section, we present two optimizations to improve the asymptotic performance, followed by the formal description of the scheme.

3.4.1 zkVPD for Input Layer

As presented in Section 3.2.3, to extend the GKR protocol to a zero knowledge argument, we need a zkVPD protocol for the low degree extension $\dot{V}_D$ of polynomial V_D defined by Equation 4.12. The variable degree of $\dot{V}_D$ for $x_2, \ldots, x_{s_D}$ is 2, and the variable degree for x_1 is 3. Naively applying our zkVPD protocol in Section 3.3.1 would incur a prover time of $O(s_D 3^{s_D})$, superlinear in the size of the input $n = O(2^{s_D})$.

Instead, we observe that the low degree extension in Equation 4.12 is of a special form: it is the sum of the multilinear extension $\tilde{V}_D$ defined by Equation 3.1 and $Z_D(x) \sum_{z\in\{0,1\}} R_D(x_1, z)$, where Z_D is publicly known and $\sum_{z\in\{0,1\}} R_D(x_1, z)$ is a degree-1 univariate polynomial, i.e. $\sum_{z\in\{0,1\}} R_D(x_1, z) = a_0 + a_1 x_1$. Therefore, the evaluation of $\dot{V}_D$ at point $t \in \mathbb{F}^{s_D}$ can be modeled as the inner product between two vectors T and c of length $n + 2$. The first n elements in T are $\prod_{i=1}^{s_D}((1 - t_i)(1 - b_i) + t_i b_i)$ for all $b \in \{0, 1\}^{s_D}$, concatenated by two more elements $Z_D(t), Z_D(t) \cdot t_1$. Similarly, the first n elements of c are $V_D(b)$ for all $b \in \{0, 1\}^{s_D}$, concatenated by a_0, a_1.

Therefore, $\mathcal{P}$ and $\mathcal{V}$ replace vectors T and c in Protocol 6 by ones described above. In addition, the first part of the GKR circuit shown in Figure 3.1 to compute T from $t_1, \ldots t_{s_D}$ is also changed according to the definition of T above. The rest of the protocol remains the same and it is straight forward to prove that the modified protocol is still correct, sound and zero knowledge. In this way, the prover time is $O(n \log n)$, the proof size is $O(\log^2 n)$ and the verification time is $O(\log^2 n)$.

3.4.2 zkVPD for Interior Layers

The second place that uses zkVPD in Protocol 3 is on the masking polynomials R_i and δ_i in each layer. By Theorem 7.2.5, $\delta_i : \mathbb{F}^{2s_{i+1}+1} \to \mathbb{F}$ is a sparse polynomial that can be expressed as the sum of $2s_{i+1} + 1$ univariate polynomials of degree $\deg(\delta_i) = O(1)$ on each variable. Therefore, instead of using the generic zkVPD in Protocol 6 with $d = \deg(\delta_i)$, we model the evaluation of δ_i as a vector inner product between two dense vectors of size $(\deg(\delta_i) + 1) \cdot (2s_{i+1} + 1)$. The vector committed by $\mathcal{P}$ consists of all coefficients in δ_i, and the one known to $\mathcal{V}$ consists of the value of each variable raised to degree $0, 1, \ldots, \deg(\delta_i)$. In addition, as the size of the vector is asymptotically the same as the number of variables, in step 9-10 of Protocol 6, $\mathcal{V}$ can compute the evaluations of $q(x)$ directly in time $O(s_{i+1})$ and it is not necessary to delegate the computation to $\mathcal{P}$ using GKR anymore. With this approach, the prover time for evaluating the masking polynomials R_i and δ_i of all layers is $O(D \log C \log\log C)$, the proof size is $O(D \log\log^2 C)$ and the verification time is $O(D \log C)$. As shown in Lemma 7.2.5, this does not introduce any asymptotic overhead for the zero knowledge argument scheme.

To further improve the efficiency in practice, we can also combine all the evaluations of R_i and δ_i into one big vector inner product using random linear combinations.

3.4.3 Putting Everything Together

With the optimizations above, the full protocol of our transparent zero knowledge argument scheme is presented in Protocol 7. Consider the following theorem:

Theorem 3.4.1. *For a finite field $\mathbb{F}$ and a family of layered arithmetic circuit $\mathcal{C}_{\mathbb{F}}$ over $\mathbb{F}$, Protocol 7 is a zero knowledge argument of knowledge for the relation*

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

as defined in Definition 6.2.2.

Moreover, for every $(C, x; w) \in \mathcal{R}$, the running time of $\mathcal{P}$ is $O(|C| + n \log n)$ field operations, where $n = |x| + |w|$. The running time of $\mathcal{V}$ is $O(|x| + D \cdot \log |C| + \log^2 n)$ if C is regular with D layers. $\mathcal{P}$ and $\mathcal{V}$ interact $O(D \log |C|)$ rounds and the total communication (proof size) is $O(D \log |C| + \log^2 n)$. In case D is $\mathsf{polylog}(|C|)$, the protocol is a succinct argument.

Soundness follows the knowledge soundness of our zkVPD protocol (Protocol 6) and Lemma 3.2.4. To prove zero knowledge, we present the simulator in Figure 3.3. The efficiency follows Lemma 7.2.5 and the efficiency of our instantiations of the zkVPD protocol with optimizations described above.

Proof. **Completeness.** It follows the completeness of Protocol 6 and the completeness of the GKR protocol in [XZZPS19a].

Soundness. It follows the soundness of Protocol 6 and the soundness of the GKR protocol with masking polynomials as proven in [CFS17; XZZPS19a]. The proof of knowledge property follows the knowledge soundness of our zkVPD protocol. In particular, the witness can be extracted using the extractor presented in Section 3.3. More formally speaking, our construction is an interactive oracle proof (IOP) as defined in [BSCS16]. Applying the transformation from IOP to an argument system using Merkle tree preserves the proof of knowledge property. Our underlying IOP is proof of knowledge as the proofs are RS codes and the witness can be efficiently extracted through decoding.

Zero knowledge. The simulator is given in Figure 3.3. $\mathcal{V}^*$ can behave arbitrarily in Step 3, 4(b), 4(e), 4(f) and 6. We include these steps as place holders to compare to Protocol 7.

To prove zero-knowledge, Step 1, 2, 4(d) and 5 of both worlds are indistinguishable because of the zero knowledge property of the zkVPD protocol in Protocol 6. As the commitments and proofs are simulated in step 2 and 4(d) by $\mathcal{S}_{vpd}$ without knowing the polynomials, Step 4(c) of both worlds are indistinguishable as shown in [XZZPS19a, Theorem 3]. Step 4(a) in both worlds are indistinguishable as δ are randomly selected in both worlds.

Removing interactions. Similar to [XZZPS19a], our construction can be made non-interactive in the random oracle model using Fiat-Shamir heuristic [FS]. As shown in recent work [BSCS16; CCHLRR18], applying Fiat-Shamir on the GKR protocol only incurs a polynomial soundness loss in the number of rounds.

Regular circuits and log-space uniform. In our scheme, the verification time is succinct only when the circuit is regular. This is the best that can be achieved for transparent ZKP, as in the worst case, the verifier must read the entire circuit, which takes linear time. In fact, as shown in [GKR15], the verification time is succinct for all log-space uniform circuits. However, it introduces an extra overhead on the prover time, thus we state all of our results on regular circuits.

In practice, with the help of auxiliary input and circuit squashing, most computations can be expressed as regular circuits with low depth, such as matrix multiplication, image scaling and Merkle hash tree in

Let λ be the security parameter, $\mathbb{F}$ be a prime field. Let $C : \mathbb{F}^n \to \mathbb{F}$ be a layered arithmetic circuit over $\mathbb{F}$ with D layers, input x and witness w such that $|x| + |w| \leq n$ and $\mathsf{out} = C(x; w)$. We construct the simulator $\mathcal{S}$ given the circuit C, the output out and input size n. Let $\mathcal{S}_{vpd}$, $\mathcal{S}_{vpd,R_i}$ and $\mathcal{S}_{vpd,\delta_i}$ be simulators of zkVPD for the witness and masking polynomials. Let $\mathcal{S}_{sc}$ be the simulator of the sumcheck protocol on Equation 3.4, given by [XZZPS19a, Theorem 3].

- $\mathcal{G}(1^\lambda)$: set pp as $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$.
- $(\mathcal{S}(\mathsf{pp}, C, \mathsf{out}, 1^n), \mathcal{V}^*(C, \mathsf{pp}))$:
  1. $\mathcal{S}$ invokes $\mathcal{S}_{vpd}$ to generate $\mathsf{com} \leftarrow \mathcal{S}_{vpd}(1^\lambda, \mathsf{pp})$ and sends com to $\mathcal{V}^*$.
  2. $\mathcal{S}$ randomly selects polynomials $R_{sim,i} : \mathbb{F}^2 \to \mathbb{F}$ and $\delta_{sim,i} : \mathbb{F}^{2s_{i+1}+1} \to \mathbb{F}$ for $i = 0, \ldots, D-1$ that have the same monomials as R_i and δ_i in step 2 of Protocol 7. $\mathcal{S}$ invokes $\mathcal{S}_{vpd,R_i}$ and $\mathcal{S}_{vpd,\delta_i}$ to generate $\mathsf{com}_{i,1} \leftarrow \mathcal{S}_{vpd,R_i}(1^\lambda, \mathsf{pp}_{R_i})$ and $\mathsf{com}_{i,2} \leftarrow \mathcal{S}_{vpd,\delta_i}(1^\lambda, \mathsf{pp}_{\delta_i})$ and send them to $\mathcal{V}^*$, where pp_{R_i} and pp_{δ_i} are corresponding public parameters. $\mathcal{S}$ also reveals $R_{sim,0}$ to $\mathcal{V}$, as V_0 is defined by out and is known to $\mathcal{V}^*$.
  3. Wait $\mathcal{V}^*$ to evaluate $\dot{V}_0(u^{(0)})$ and $\dot{V}_0(v^{(0)})$ for randomly chosen $u^{(0)}, v^{(0)} \in \mathbb{F}^{s_0}$.
  4. For $i = 0, \ldots, D-1$:
     a) $\mathcal{S}$ sends $H_i = \sum_{x,y \in \{0,1\}^{s_{i+1}}, z \in \{0,1\}} \delta_{sim,i}(x, y, z)$ to $\mathcal{V}^*$.
     b) Receive $\alpha_i, \beta_i, \gamma_i$ from $\mathcal{V}^*$.
     c) $\mathcal{S}$ simulates the sumcheck protocol on Equation 3.4 using $\mathcal{S}_{sc}$. At the end of the sumcheck, $\mathcal{S}$ receives queries of $\delta_{sim,i}$ and $R_{sim,i}$ at point $u^{(i+1)}, v^{(i+1)} \in \mathbb{F}^{s_{i+1}}, g_i \in \mathbb{F}$ selected by $\mathcal{V}^*$. $\mathcal{S}$ randomly computes $\dot{V}_{i+1}(u^{(i+1)}), \dot{V}_{i+1}(v^{(i+1)})$ satisfying Equation 3.4 at point $u^{(i+1)}, v^{(i+1)}, g_i$ and send them to $\mathcal{V}^*$.
     d) $\mathcal{S}$ computes $R_{sim,i}(u^{(i)}, g_i)$, $R_{sim,i}(v^{(i)}, g_i)$ and $\delta_{sim,i}(u^{(i+1)}, v^{(i+1)}, g_i)$ and invokes $\mathcal{S}_{vpd,R_i}$ and $\mathcal{S}_{vpd,\delta_i}$ to generate the proofs of these evaluations.
     e) Wait for $\mathcal{V}^*$ to validate $R_{sim,i}(u^{(i)}, g_i)$, $R_{sim,i}(v^{(i)}, g_i)$ and $\delta_{sim,i}(u^{(i+1)}, v^{(i+1)}, g_i)$.
     f) Wait for $\mathcal{V}^*$ to check the last claim of the sumcheck about f_i' using $R_{sim,i}(u^{(i)}, g_i)$, $R_{sim,i}(v^{(i)}, g_i)$, $\delta_{sim,i}(u^{(i+1)}, v^{(i+1)}, g_i)$, $\dot{V}_{i+1}(u^{(i+1)})$ and $\dot{V}_{i+1}(v^{(i+1)})$.
  5. In last part of the protocol, $\mathcal{S}$ needs to prove to $\mathcal{V}^*$ the values of $\dot{V}_D(u^{(D)})$ and $\dot{V}_D(v^{(D)})$, where $u^{(D)} \in \mathbb{F}^n$ and $v^{(D)} \in \mathbb{F}^n$ are chosen by $\mathcal{V}^*$. $\mathcal{S}$ gives $u^{(D)}$, $\dot{V}_D(u^{(D)})$ to $\mathcal{S}_{vpd}$ and invokes $\mathcal{S}_2$ of $\mathcal{S}_{vpd}$ in Figure 3.2 to simulate this process. Do the same process again for $v^{(D)}$, $\dot{V}_D(v^{(D)})$.
  6. Wait for $\mathcal{V}$ to run $\mathsf{zkVPD.Verify}$ to validate the value of $\dot{V}_D(u^{(D)})$ and $\dot{V}_D(v^{(D)})$.

Figure 3.3: Simulator $\mathcal{S}$ of Virgo.

Section 7.5. Asymptotically, as shown in [BSCTV; ZGKPP18; BSBHR19], all random memory access (RAM)

programs can be validated by circuits that are regular 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.

3.5 Implementation and Evaluation

We implement Virgo, a zero knowledge proof system based on our construction in Section 3.4. The system is implemented in C++. There are around 700 lines of code for our transparent zkVPD protocol and 2000 lines for the GKR part.

Hardware. We run all of the experiments on AMD Ryzen™ 3800X Processor with 64GB RAM. 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, unless specified otherwise.

3.5.1 Choice of Field with Efficient Arithmetic

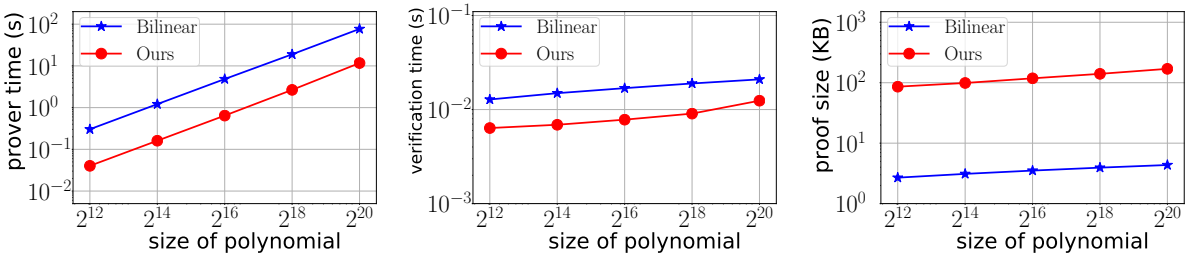
One important optimization we developed during the implementation is on the choice of the underlying field. Our scheme is transparent and does not use any discrete log or bilinear pairing as in [ZGKPP17c; ZGKPP17a; WTSTW18; XZZPS19a]. However, there is one requirement on the finite field: in order to run the low degree test protocol in [BSBHR18], either the field is an extension of $\mathbb{F}_2$, or there exists a multiplicative subgroup of order 2^k in the field for large enough k (one can think of $2^k \geq |\mathbb{L}| = O(|\mathbb{H}|) = O(n)$). Existing zero knowledge proof systems that use the LDT protocol as a building block such as Stark [BSBHR19] and Aurora [BSCRSVW19] run on the extension fields $\mathbb{F}_{2^{64}}$ and $\mathbb{F}_{2^{192}}$. Modern CPUs (e.g., AMD Ryzen™ 3800X Processor) have built-in instructions for field arithmetics on these extension fields, which improves the performance of these systems significantly. However, the drawback is that the arithmetic circuits representing the statement of ZKP must also operate on the same field, and the additions (multiplications) are different from integer or modular additions (multiplications) that are commonly used in the literature. Because of this, Stark [BSBHR19] has to design a special SHA-256 circuit on $\mathbb{F}_{2^{64}}$, and Aurora [BSCRSVW19] only reports the performance versus circuit size (number of constraints), but not on any commonly used functions.

One could also use a prime field p with an order-2^k multiplicative subgroup. Equivalently, this requires that 2^k is a factor of $p-1$. In fact, there exist many such primes and Aurora [BSCRSVW19] also supports prime fields. However, the speed of field arithmetic is much slower than extension fields of $\mathbb{F}_2$ (see Table 3.1).

	128-bit prime	$\mathbb{F}_{2^{64}}$	$\mathbb{F}_{2^{192}}$	Our field
$+$	6.29ns	2.16ns	4.75ns	1.23ns
$\times$	30.2ns	7.29ns	15.8ns	8.27ns

Table 3.1: Speed of basic arithmetic on different fields. The time is averaged over 100 million runs and is in nanosecond.

In this paper, we provide an alternative to achieve the best of both cases. A first attempt is to use Mersenne primes, primes that can be expressed as $p = 2^m - 1$ for integers m. As shown in [CMT12; Tha13b], multiplications modulo Mersenne primes is known to be very efficient. However, Mersenne primes do not



(a) $\mathcal{P}$ time (b) $\mathcal{V}$ time (c) Proof size

Figure 3.4: Comparison of our zkVPD and the pairing-based zkVPD in [ZGKPP17a].

satisfy the requirement of the LDT, as $p-1 = 2^m - 2 = 2 \cdot (2^{m-1} - 1)$ only has a factor 2^1. Instead, we propose to use the extension field of a Mersenne prime $\mathbb{F}_{p^2}$.The multiplicative group of $\mathbb{F}_{p^2}$ is a cyclic group of order $p^2 - 1 = (2^m - 1)^2 - 1 = 2^{2m} - 2^{m+1} = 2^{m+1}(2^{m-1} - 1)$, thus it has a multiplicative subgroup of order 2^{m+1}, satisfying the requirement of LDT when m is reasonably large. Meanwhile, to construct an arithmetic circuit representing the statement of the ZKP, we still encode all the values in the first slot of the polynomial ring defined by $\mathbb{F}_{p^2}$. In this way, the additions and multiplications in the circuit are on $\mathbb{F}_p$ and our system can take the same arithmetic circuits over prime fields in prior work. Meanwhile, the LDT, zkVPD and GKR protocol are executed on $\mathbb{F}_{p^2}$, preserving the soundness over the whole field.

With this alternative approach, we can implement modular multiplications on $\mathbb{F}_{p^2}$ using 3 modular multiplications on $\mathbb{F}_p$. (The modular multiplication is analog to multiplications of complex numbers.) In our implementation, we choose Mersenne prime $p = 2^{61} - 1$, thus our system provides 100+ bits of security. We implement modular multiplications on $\mathbb{F}_p$ for $p = 2^{61} - 1$ with only one integer multiplication in C++ (two 64-bit integers to one 128-bit integer) and some bit operations. As shown in Table 3.1, the field arithmetic on $\mathbb{F}_{p^2}$ is comparable to $\mathbb{F}_{2^{64}}$, $2\times$ faster than $\mathbb{F}_{2^{192}}$ and $4\times$ faster than a 128-bit prime field. Encoding numbers in $\mathbb{F}_p$ for $p = 2^{61} - 1$ is enough to avoid overflow for all computations used in our experiments in Section 3.5.2. For other computations requiring larger field, one can set p as $2^{89} - 1, 2^{107} - 1$ or $2^{127} - 1$, which incurs a moderate slow down. For example, the multiplication over $\mathbb{F}_{p^2}$ for $p = 2^{89} - 1$ is $2.7\times$ slower than $p = 2^{61} - 1$.

This optimization can also be applied to Stark [BSBHR19] and Aurora [BSCRSVW19], which use the same LDT in [BSBHR18]. Currently they run on $\mathbb{F}_{2^{64}}$ and $\mathbb{F}_{2^{192}}$ and their performances are reported in Section 3.5.3. With our optimization, they can run on $\mathbb{F}_{p^2}$ with similar efficiency while taking arithmetic circuits in $\mathbb{F}_p$.

3.5.2 Performance of zkVPD

In this section, we present the performance of our new transparent zkVPD protocol, and compare it with the existing approach based on bilinear maps. We use the open-source code of [XZZPS19a], which implements the zkVPD scheme presented in [ZGKPP17a]. For our new zkVPD protocol, we implement the univariate sumcheck and the low degree test described in Section 3.2.4. We set the repetition parameter κ in Lemma 3.2.7 as 33, and the rate of the RS code as 32 (i.e., $|\mathbb{L}| = 32|\mathbb{H}|$). These parameters provide 100+ bits of security,

based on Theorem 1.2 and Conjecture 1.5 in [BSBHR18], and are consistent with the implementation of Aurora [BSCRSVW19]. In addition, we use the field $\mathbb{F}_{p^2}$ with $p = 2^{61} - 1$, which has a multiplicative subgroup of order 2^{m+1}. Thus $|\mathbb{L}|$ can be as big as 2^{60} and the size of the witness $|\mathbb{H}|$ is up to 2^{55}. We pad the size of the witness to a power of 2, which introduces an overhead of at most $2\times$.

Figure 3.4 shows the prover time, verification time and proof size of the two schemes. We fix the variable degree of the polynomial as 1 and vary the number of variables from 12 to 20. The size of the multilinear polynomial is 2^{12} to 2^{20}. As shown in the figure, the prover time of our new zkVPD scheme is 8-10$\times$ faster than the pairing-based one. It only takes 11.7s to generate the proof for a polynomial of size 2^{20}. This is because our new scheme does not use any heavy cryptographic operations, while the scheme in [ZGKPP17a] uses modular exponentiations on the base group of a bilinear map. In terms of the asymptotic complexity, though the prover time is claimed to be linear in [ZGKPP17a], there is a hidden factor of $\log |\mathbb{F}|$ because of the exponentiations. The prover complexity of our scheme is $O(n \log n)$, which is strictly better than $O(n \log |\mathbb{F}|)$ field operations. Additionally, as explained in Section 3.5.1, our scheme is on the extension field of a Mersenne prime, while the scheme in [ZGKPP17a] is on a 254-bit prime field with bilinear maps, the basic arithmetic of which is slower.

The verification time of our zkVPD scheme is also comparable to that of [ZGKPP17a]. For $n = 2^{20}$, it takes 12.4ms to validate the proof in our scheme, and 20.9ms in [ZGKPP17a].

The drawback of our scheme is the proof size. As shown in Figure 3.4(c), the proof size of our scheme is 30-40$\times$ larger than that of [ZGKPP17a]. This is due to the opening of the commitments using Merkle tree, which is a common disadvantage of all IOP-based schemes [AHIV17; BSBHR19; BSCRSVW19]. The proof size of our scheme can be improved by a factor of $\log n$ using the vector commitment scheme with constant-size proofs in [BBF18], with a compromise on the prover time. This is left as a future work.

Finally, the scheme in [ZGKPP17a] requires a trusted setup phase, which takes 12.6s for $n = 2^{20}$. We remove the trusted setup completely in our new scheme.

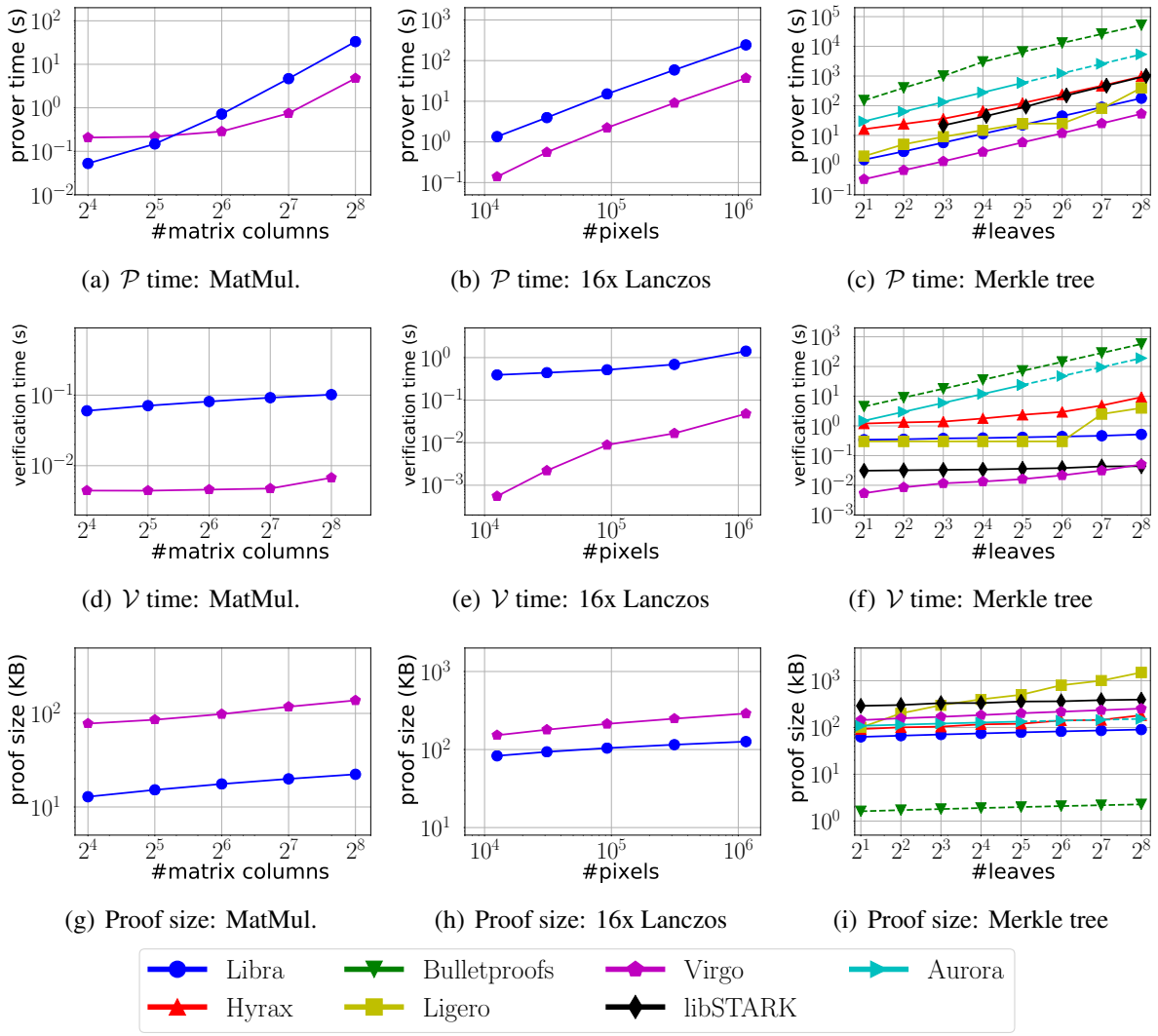
3.5.3 Performance of Virgo

In this section, we present the performance of our ZKP scheme, Virgo, and compare it with existing ZKP systems.

Methodology. We first compare with Libra [XZZPS19a], as our scheme follows the same framework and replaces the zkVPD with our new transparent one. We use the open-source implementation and the layered arithmetic circuits at [Liba] for all the benchmarks. The circuits are generated using [Tan11].

We then compare the performance of Virgo to state-of-the-art transparent ZKP systems: Ligero [AHIV17], Bulletproofs [BBBPWM], Hyrax [WTSTW18], Stark [BSBHR19] and Aurora [BSCRSVW19]. We use the open-source implementations of Hyrax, Bulletproofs and Aurora at [Hyr] and [Aur]. As the implementation of Aurora runs on $\mathbb{F}_{2^{192}}$, we execute the system on a random circuit with the same number of constraints. For Ligero, as the system is not open-source, we use the same number reported in [AHIV17] on computing hashes. For Stark, after communicating with the authors, we obtain numbers for proving the same number of hashes in the 3rd benchmark. The experiments were executed on a server with 512GB of DDR3 RAM (1.6GHz) and 16 cores (2 threads per core) at speed of 3.2GHz.

Benchmarks. We evaluate the systems on three benchmarks: matrix multiplication, image scaling and Merkle tree, which are used in [WTSTW18; XZZPS19a].



(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 3.5: Comparisons of prover time, proof size and verification time between Virgo and existing ZKP systems.

- Matrix multiplication: $\mathcal{P}$ proves to $\mathcal{V}$ that it knows two matrices whose product equals a public matrix. We evaluate on different dimensions from 4×4 to 256×256, and the size of the circuit is n^3.
- 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: $k(x) = \text{sinc}(x)/\text{sinc}(ax)$, if $-a < x < a; k(x) = 0$, otherwise, where a is the scaling parameter and $\text{sinc}(x) = \sin(x)/x$. We evaluate by fixing the window size as 16×16 and increase the image size from 112x112 to 1072x1072.
- Merkle tree: $\mathcal{P}$ proves to $\mathcal{V}$ that it knows the value of the leaves of a Merkle tree 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

	Ligero	Bulletproofs	Hyrax	Stark	Aurora	Virgo
$\mathcal{P}$ time	$O(C \log C)$	$O(C)$	$O(C \log C)$	$O(C \log^2 C)$	$O(C \log C)$	$O(C + n \log n)$
$\mathcal{V}$ time	$O(C)$	$O(C)$	$O(D \log C + \sqrt{n})$	$O(\log^2 C)$	$O(C)$	$O(D \log C + \log^2 n)$
Proof size	$O(\sqrt{C})$	$O(\log C)$	$O(D \log C + \sqrt{n})$	$O(\log^2 C)$	$O(\log^2 C)$	$O(D \log C + \log^2 n)$

Table 3.2: Performance of transparent ZKP systems. C is the size of the regular circuit with depth D, and n is witness size.

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. The circuit size of each SHA256 is roughly 2^{18} gates and the size of the largest Merkle tree instance is around 2^{26} gates.

Comparing to Libra. Figure 3.5 shows the prover time, verification time and proof size of our ZKP system, Virgo, and compares it with Libra. The prover time of Virgo is 7-10× faster than Libra on the first two benchmarks, and 3-5× faster on the third benchmark. The speedup comes from our new efficient zkVPD. As shown in Section 3.5.2, the prover time of our zkVPD is already an order of magnitude faster. Moreover, the GKR protocol for the whole arithmetic circuit must operate on the same field of the zkVPD. In Libra, it runs on a 254-bit prime field matching the base group of the bilinear map, though the GKR protocol itself is information-theoretic secure and can execute on smaller fields. This overhead is eliminated in Virgo, and both zkVPD and GKR run on our efficient extension field of Mersenne prime, resulting in an order of magnitude speedup for the whole scheme. It only takes 53.40s to generate the proof for a circuit of 2^{26} gates. Our improvement on the third benchmark is slightly less, as most input and values in the circuit are binary for SHA-256, which is more friendly to exponentiation used in Libra.

The verification time of Virgo is also significantly improved upon Libra, leading to a speedup of 10-30× in the benchmarks. This is because in Libra, the verification time of the zkVPD for the input layer is similar to that for the masking polynomials in each layer, both taking $O(\log C)$ bilinear pairings. Thus the overall verification time is roughly D times one instance of zkVPD verification. This is not the case in Virgo. As explained in the optimization in Section 3.4.2, we combine all the evaluations into one inner product through random linear combinations. Therefore, the verification time in Virgo is only around twice of the zkVPD verification time, ranging from 7ms to 50ms in all the benchmarks.

Because of the zkVPD, the proof size of Virgo is larger than Libra. For example, Virgo generates a proof of 253KB for Merkle tree with 256 leaves, while the proof size of Libra is only 90KB. However, the gap is not as big as the zkVPD schemes themselves in Section 3.5.2, as the proof size of Libra is dominated by the GKR protocol of the circuit, which is actually improved by 2× in Virgo because of the smaller field. Finally, Libra requires a one-time trusted setup for the pairing-based zkVPD, while Virgo is transparent.

Comparing to other transparent ZKP Systems. Table 3.2 and Figure 3.5 show the comparison between Virgo and state-of-the-art transparent ZKP systems. As shown in Figure 3.5, Virgo is the best among all systems in terms of practical prover time, which is faster than others by at least an order of magnitude. The verification time of Virgo is also one of the best thanks to the succinctness of our scheme. It only takes 50ms to verify the proof of constructing a Merkle tree with 256 leaves, a circuit of size 2^{26} gates. The verification time is competitive to Stark, and faster than all other systems by 2 orders of magnitude. The proof size of Virgo is also competitive to other systems. It is larger than Bulletproofs [BBBPWM] and is similar to Hyrax, Stark and Aurora.

In particular, our scheme builds on the univariate sumcheck proposed in [BSCRSVW19]. Compared to the system Aurora, Virgo significantly improves the prover time due to our efficient field and the fact that the univariate sumcheck is only on the witness, but not on the whole circuit. For the computation in Figure 3.5, the witness size is $16\times$ smaller than the circuit size. E.g., the witness size for one hash is around 2^{14} while the circuit size is 2^{18}. In the largest instance in the figure, the witness size is 2^{22} while the circuit size is 2^{26}. The verification time is also much faster as we reduce the complexity from linear to logarithmic. The proof size is similar to Aurora. Essentially the proof size is the same as that in Aurora on the same number of constraint as the witness size, plus the size of the GKR proofs in the zkVPD and for the whole circuit.

3.6 Applications

In this section, we discuss several applications of our new zkVPD and ZKP schemes.

3.6.1 Verifiable Secret Sharing

Verifiable polynomial delegations (or polynomial commitments) are widely used in secret sharing to achieve malicious security. In Shamir's secret sharing [Sha79], the secret is embedded as the constant term of a univariate polynomial $f(x)$, and the shares hold by party i is the evaluation of the polynomial $f(i)$. To update the shares, in proactive secret sharing [HJKY95], each party generates a random polynomial $\delta(x)$ with constant term 0, and sends the evaluation of the polynomial $\delta(i)$ to party i. To prevent adversaries from changing the secret or sending inconsistent shares, the random polynomial is committed using a polynomial commitment scheme together with a proof that $\delta(0) = 0$, and each evaluation to party i comes with a proof of the polynomial evaluation. Similar mechanism is used in mobile secret sharing [SLL08] to change the parties.

Existing schemes mainly apply the VPD scheme in [KZG], which requires a trusted setup phase to generate the structured reference string. In addition, the computation time to generate the proofs are high because of the use of modular exponentiation. For example, in a recent paper, Maram et al. [Mar+19] proposed a mobile and proactive secret sharing scheme on blockchain. As it is using the pairing-based VPD, the SRS has to be generated by a trusted party, posted on the blockchain while the trapdoor must be destroyed after the setup. Moreover, as shown in [Mar+19, Figure 5], it takes 185s for each party to generate the proofs of the polynomial evaluations in each phase of the scheme for a committee of 1000 parties, which is the bottleneck of the system.

Using our new VPD scheme in Protocol 5, we can completely remove the trusted setup phase of these secret sharing schemes for the first time, while maintaining the succinct proof size and verification time. Additionally, the proof generation time is significantly improved. Based on Figure 3.4, it will take around 11s to generate the proofs for 1000 parties. The proof size will definitely increase. However, as the proofs are sent offline among the parties in [Mar+19], the overall throughput will be improved by at least an order of magnitude with reasonable bandwidth between parties.

[5] When the circuit is data parallel, the prover time of Hyrax [WTSTW18] is $O(C + C' \log C')$ where C' is the size of each copy in the data parallel circuit. Hyrax has the option with proof size $O(D \log C + n^{\tau})$ and verification time $O(D \log C + n^{1-\tau})$ for $\tau \in [0, \frac{1}{2}]$.

3.6.2 Privacy on Blockchain

Zero knowledge proof is widely used in blockchain systems to provide privacy for cryptocurrencies (e.g., Zcash [Ben+14]), smart contracts (e.g., Hawk [KMSWP]) and zero knowledge contingent payment [CGGN17]. As mentioned in the introduction, the most commonly deployed ZKP scheme, SNARK [BSCTV], requires a trusted setup phase. A trusted party is usually absent in the setting of blockchains and an expensive "ceremony" [BSCGTV15] among multiple parties is usually deployed to generate the SRS. To address this issue, there are recent attempts to use transparent ZKP schemes. For example, in [BAZB19], Bünz et at. proposed Zether, which uses a variant of Bulletproofs [BBBPWM] to hide account balances and provide confidentiality for applications such as auction. However, due to the high prover time and verification time of Bulletproofs for general computations, providing full anonimity still remain impractical.

As shown in Section 3.5.3, among all transparent ZKP schemes, Virgo achieves the best prover time and one of the best verification time, which are critical for applications of ZKP on blockchains. Compared to existing GKR-based ZKP scheme, Virgo removes the trusted setup of Libra [XZZPS19a], and improves the verification time of both Libra and Hyrax [WTSTW18] by 1-2 orders of magnitude. These make Virgo a good candidate to build privacy-preserving cryptocurrencies and smart contract without trusted setup. The overhead on the proof size is comparable to schemes based on IOPs, which is acceptable in scenarios such as permissioned blockchain and can be potentially reduced through proof composition [BSCTV14].

3.6.3 Large Scale Zero Knowledge Proof

Other than blockchain, there are many other applications of ZKP that require proving large statements. For example, defense advanced research project agency (DARPA) recently intended to use ZKP to prove the behavior of complicated programs without leaking sensitive information [Dar]. Such applications require scaling ZKP schemes to circuits with billions of gates. The obstacles in all existing ZKP schemes are the high overhead of running time and memory consumption on the prover. In our new scheme, we completely removes the operations of modular exponentiation in Hyrax [WTSTW18] and Libra [XZZPS19a], which is the bottleneck of both the prover time and memory usage. Our implementation, Virgo, is purely based on symmetric-key operations, which are fast and memory friendly. As shown in the experiments, Virgo is promising to scale to large circuits and enable applications such as proving program behavior on secret data or states.

Chapter 4

Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time

We propose a new doubly efficient interactive proof protocol for general arithmetic circuits. The protocol generalizes the interactive proof for layered circuits proposed by Goldwasser, Kalai and Rothblum to arbitrary circuits, while preserving the optimal prover complexity that is strictly linear to the size of the circuits. The proof size remains succinct for low depth circuits and the verifier time is sublinear for structured circuits. We then construct a new zero knowledge argument scheme for general arithmetic circuits using our new interactive proof protocol together with polynomial commitments.

Our key technique is a new sumcheck equation that reduces a claim about the output of one layer to claims about its input only, instead of claims about all the layers above which inevitably incurs an overhead proportional to the depth of the circuit. We developed efficient algorithms for the prover to run this sumcheck protocol and to combine multiple claims back into one in linear time in the size of the circuit.

Not only does our new protocol achieve optimal prover complexity asymptotically, but it is also efficient in practice. Our experiments show that it only takes 0.3 seconds to generate the proof for a circuit with more than 600,000 gates, which is 13 times faster than the original interactive proof protocol on the corresponding layered circuit. The proof size is 208 kilobytes and the verifier time is 66 milliseconds. Our implementation can take general arithmetic circuits directly, without transforming them to layered circuits with a high overhead on the size of the circuit.

This work was previously published in [Zha+20].

4.1 Introduction

Interactive proofs allow a powerful yet untrusted prover to convince a verifier through a sequence of interactions that the result of a computation is correctly computed. Since they were introduced by Goldwasser, Micali, and Rackoff [GMR89] in the 1980s, interactive proofs have expanded people's understanding on traditional static mathematical proofs and led to many important theoretical results in complexity theory, such as IP=PSPACE [LFKN92; Sha92] and MIP=NEXP [BFL91].

In recent years, there is great progress on turning interactive proofs from purely theoretical constructions to practical schemes with efficient implementations. In the seminal work of [GKR15], Goldwasser, Kalai and Rothblum proposed doubly efficient interactive proofs where the prover can convince the verifier the correctness of the evaluation of a layered arithmetic circuit with addition gates and multiplication gates of fan-in two. The time for the prover to generate all the messages during the protocol (prover time) is a polynomial on the size of the circuit, and the time to validate the result (verifier time) is close to linear in the size of the input for log-space uniform circuits, thus the name "doubly efficient". The total communication between the prover and the verifier is only poly-logarithmic in the size of the circuit and linear in the depth of the circuit, which is *succinct* for bounded-depth circuits. We refer the protocol in [GKR15] as the *GKR* protocol in this paper. Later, researchers spent great effort improving the concrete efficiency of the GKR protocol. The prover time was improved to quasi-linear ($O(|C| \log |C|)$) in [CMT12], and then to close to linear for various circuits with different structures [Tha13b; Wah+17; ZGKPP18]. Finally, in [XZZPS19b], Xie et al. proposed an algorithm to improve the prover time to strictly linear ($O(|C|)$) for layered arithmetic circuits without assuming any structures, which is asymptotically optimal and very efficient in practice.

Another important advance of interactive proofs is using them to construct efficient zero knowledge argument schemes. In [ZGKPP17c], Zhang et al. first proposed to combine the GKR protocol with polynomial commitments to build argument systems, where the prover can further prove to the verifier the computations on the prover's witness, without sending the witness directly to the verifier. Following the framework, there are many subsequent zero knowledge argument constructions based on interactive proofs, including [WTSTW18; ZGKPP17a; XZZPS19b; ZXZS; Set20a]. These schemes demonstrate great prover efficiency and can achieve sublinear verifier time for structured circuits, thanks to the advantages of the interactive proofs and the GKR protocol.

Despite the progress of the GKR protocol, a major drawback is that the protocol only works for layered arithmetic circuits. Each gate can only connect to the layer above, due to the layer-by-layer reduction of the GKR protocol. In practice, it introduces a high overhead to pad general circuits to layered circuits using relay gates. Asymptotically, the circuit size increases from $O(|C|)$ to $O(d|C|)$ where $|C|$ is the size of the general circuit and d is the depth of the circuit. This is easily 1-2 orders of magnitude larger in practice as we show in our experiments, and introduces a big overhead on the prover time. Moreover, it is also inconvenient to implement circuits in a strictly layered way, and most existing tools such as rank-1-constraint-system (R1CS) cannot be used directly. Therefore, in this paper we ask the following question:

Is it possible to generalize the GKR protocol to directly support general circuits, without introducing any overhead on the prover time?

4.1.1 Our Contributions

We answer the above question affirmatively by proposing a generalized doubly efficient interactive proofs for arbitrary arithmetic circuits, where each gate can take the output of any gate as input. The prover time is still

linear to the size of the circuit, and is very efficient in practice. In particular, our contributions are:

- We generalize the GKR protocol to work on arbitrary arithmetic circuits efficiently for the first time. For a general circuit of size $|C|$ and depth d, the prover time is $O(|C|)$, the same as the original GKR protocol on a layered circuit with the same size. The overhead on the proof size and the verifier time is minimal. The proof size in our new protocol is $\min\{O(d \log |C| + d^2), O(|C|)\}$. For structured circuits, the verifier time is also $\min\{O(d \log |C| + d^2), O(|C|)\}$. Those in the original GKR are $\min\{O(d \log |C|), O(|C|)\}$.
- Together with zero knowledge polynomial commitments, we construct zero knowledge arguments for general arithmetic circuits. The zero knowledge version of our interactive proof protocols does not incur any overhead asymptotically on the prover time, the proof size and the verifier time compared to the plain version without zero knowledge.
- We fully implement a system, virgo++, for our new interactive proof protocols and zero knowledge arguments. We show that on random circuits with $d = 50$ and $d = 75$, our new protocols are 9-13$\times$ faster than the state-of-the-art GKR protocol on the corresponding layered circuits. The prover time per gate (the constant in the linear complexity) is only 1.3$\times$ more than the original GKR protocol on layered circuits. Therefore, as long as padding the general circuit to layered circuit makes the size 1.3$\times$ or larger, our new protocol will have faster prover time. The verifier time of our new protocols is 17-25$\times$ faster, while the proof size is only slightly larger than GKR on layered circuits.

4.1.2 Technical Overview

The key idea of the GKR protocol is to write the values in the i-th layer of the circuit as an equation of the values in the previous layer $i + 1$. Then starting from the output layer (layer 0), $\mathcal{P}$ and $\mathcal{V}$ reduce the correctness of the values in layer i to the correctness of the values in layer $i + 1$ recursively, and eventually to the correctness of the input. $\mathcal{V}$ can then validates the correctness of the input on her own, which completes the reduction and guarantees that the output is correctly computed. To do so, we use the notation of multilinear extension $\tilde{V}_i()$ of the i-th layer in [Tha15], which is a multilinear polynomial that agrees with all the values in the i-th layer on the Boolean hypercube, i.e., $\tilde{V}_i(0, 0, \ldots, 0) = \mathbf{V}_i[0], \tilde{V}_i(0, 0, \ldots, 1) = \mathbf{V}_i[1], \ldots$ where $\mathbf{V}_i$ is the array representing the values in the i-th layer of the circuit. Assuming for simplicity that all layers have S gates and $\tilde{V}$ takes $s = \log S$ variables, we can write $\tilde{V}_i()$ as a equation of $\tilde{V}_{i+1}()$:

$$\tilde{V}_i(z) = \sum_{x,y \in \{0,1\}^s} (\tilde{add}_{i+1}(z, x, y)(\tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(y)) + \tilde{mult}_{i+1}(z, x, y)\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y))$$

for all $z \in \{0, 1\}^s$, where $\tilde{add}_{i+1}(z, x, y)$ and $\tilde{mult}_{i+1}(z, x, y)$ are polynomials describing the addition/multiplication gates and their connections in the circuit between layer i and layer $i + 1$. With this equation, the GKR protocol invokes the sumcheck protocol (See Section 7.2.2), which reduces the correctness of $\tilde{V}_i(g)$ at a random point $g \in \mathbb{F}^s$ to the correctness of $\tilde{V}_{i+1}(u)$ and $\tilde{V}_{i+1}(v)$ at two random points $u, v \in \mathbb{F}^s$. Then $\tilde{V}_{i+1}(u)$ and $\tilde{V}_{i+1}(v)$ can be combined back to a single evaluation of $\tilde{V}_{i+1}(w)$ for $w \in \mathbb{F}^s$. At this point, $\tilde{V}_{i+1}(w)$ can be further reduced to an evaluation of $\tilde{V}_{i+2}$ using the same equation and protocol for layer $i + 1$. Therefore, starting from the output layer, $\mathcal{P}$ and $\mathcal{V}$ perform the reduction layer by layer to the input layer, which can be validated by $\mathcal{V}$ directly. The prover time is $O(S)$ in each layer [XZZPS19b] and the proof size is only $O(\log S)$. Therefore, the total prover time is $O(dS) = O(|C|)$ and the proof size is

$O(d \log S) = O(d \log |C|)$.

Extending GKR to general circuits naively. The above equation relies on the fact that gates in layer i can only take input from gates in layer $i+1$. In a general circuit, a gate in layer i can take input from any gate in layer j for $j > i$. As circuits cannot contain cycles (otherwise we cannot get outputs of the circuit), we can still assign a layer number to each gate in the topological order. Thus a gate can take input from any gate in layers above, but not below. More interestingly, every gate in layer i has to have at least one input from layer $i+1$, otherwise it cannot belong to layer i in the topological order. Because of this generalization, we can write $\tilde{V}_i()$ as:

$$\begin{aligned}\tilde{V}_i(z) = \sum\nolimits_{x,y\in\{0,1\}^s} & (\tilde{add}_{i+1,i+1}(z,x,y)(\tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(y)) \\ & + \tilde{mult}_{i+1,i+1}(z,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)) \\ & + \tilde{add}_{i+1,i+2}(z,x,y)(\tilde{V}_{i+1}(x) + \tilde{V}_{i+2}(y)) + \tilde{mult}_{i+1,i+2}(z,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+2}(y)) \\ & + \ldots + \tilde{add}_{i+1,d}(z,x,y)(\tilde{V}_{i+1}(x) + \tilde{V}_d(y)) + \tilde{mult}_{i+1,d}(z,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_d(y))).\end{aligned}$$

Namely, we have multiple parts in the summation, one for each layer $j = i+1, i+2, \ldots, d$. $\mathcal{P}$ and $\mathcal{V}$ run the sumcheck protocol on this equation, which reduces the correctness of $\tilde{V}_i(g)$ at a random point $g \in \mathbb{F}^s$ to the correctness of $\tilde{V}_{i+1}(u)$ and $\tilde{V}_{i+1}(v), \tilde{V}_{i+2}(v), \ldots, \tilde{V}_d(v)$ at random points $u, v \in \mathbb{F}^s$. Moreover, when reaching layer $i+1$, now $\mathcal{V}$ has many evaluations about $\tilde{V}_{i+1}$ instead of just two. In the sumcheck protocols of all layers below, $\mathcal{V}$ has received one evaluation of $\tilde{V}_{i+1}$ from the sumcheck of layer $k = 0, \ldots, i-1$, and two evaluations from layer i. Nevertheless, $\mathcal{V}$ can combine all these evaluations into one evaluation $\tilde{V}_{i+1}(w)$ using the original protocol multiple times with $\mathcal{P}$. $\mathcal{P}$ and $\mathcal{V}$ can then run the protocol recursively layer by layer just as the original GKR protocol to reduce the correctness of the output layer to the input layer.

It is not hard to show that the generalized protocol is secure. However, it introduces a big overhead on the prover time. The size of all the polynomials in the generalized equation becomes $O((d-i)S)$, and the total prover time for the sumcheck protocol of all layers becomes $O(dS + (d-1)S + \ldots + S) = O(d^2 S) = O(d|C|)$. This is as bad as padding the general circuit to a layered circuit and running the original GKR protocol on it. Even worse, the second step of combining multiple evaluations into one also introduces a prover time of $O(d|C|)$, because there are now $i+1$ evaluations to combine instead of two.

Extending GKR to general circuits with optimal prover time. In order to preserve the linear prover time, we introduce two new techniques. First, we observe that the key reason why the prover time of the sumcheck protocol on the generalized equation becomes $O((d-i)S)$ is that the multilinear extension $\tilde{V}_j$ of the entire layer j for $j > i$ is used. As layer j has S gates and its multilinear extension is uniquely defined by these gates, merely writing out all the polynomials $\tilde{V}_j$ for $j > i$ takes $O((d-i)S)$ time. There is no hope to reduce the prover time if we define the equation in this way. Meanwhile, it is also not necessary to use all the gates in layers above, because gates in layer i can at most take input from $2S$ gates in total. Therefore, we propose a new equation to write $\tilde{V}_i$ as a function of multilinear extensions define by *only those values used by layer* i from layer $j > i$. In particular, we have

$$\begin{aligned}\tilde{V}_i(z) = &\sum\nolimits_{x,y\in\{0,1\}^{s'}} (\tilde{add}_{i+1,i+1}(z,x,y)(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,i+1}(y)) \\ &+ \tilde{mult}_{i+1,i+1}(z,x,y)(\tilde{V}_{i,i+1}(x)\tilde{V}_{i,i+1}(y)) \\ &+ \tilde{add}_{i+1,i+2}(z,x,y)(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,i+2}(y)) + \tilde{mult}_{i+1,i+2}(z,x,y)\tilde{V}_{i,i+1}(x)\tilde{V}_{i,i+2}(y) \\ &+ \ldots + \tilde{add}_{i+1,d}(z,x,y)(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,d}(y)) + \tilde{mult}_{i+1,d}(z,x,y)\tilde{V}_{i,i+1}(x)\tilde{V}_{i,d}(y)),\end{aligned}$$

where $\tilde{V}_{i,j}$ is the multilinear extension of values used by layer i from layer j for $j > i$ arranged in a pre-defined order, i.e., a subset of the values in the entire layer j. Now the total size of the all these polynomials is bounded by $2S$. We also change the range of the summation from $\{0,1\}^s$ to $\{0,1\}^{s'}$ to informally denote that now the number of variables is smaller. We will show how to deal with different sizes from different layers in our formal protocols. We then design a new algorithm for the prover to run the sumcheck protocol on the equation above with time complexity $O(S)$ by utilizing the sparsity of the polynomials $\tilde{add}$ and $\tilde{mult}$. The formal algorithms are presented in Section 4.3.2.

Combining evaluations of different multilinear extensions. At the end of the sumcheck protocol on the equation above, $\mathcal{P}$ and $\mathcal{V}$ reduce the correctness of $\tilde{V}_i(g)$ at a random point $g \in \mathbb{F}^s$ to the correctness of $\tilde{V}_{i,i+1}(u)$ and $\tilde{V}_{i,i+1}(v), \tilde{V}_{i,i+2}(v), \ldots, \tilde{V}_{i,d}(v)$ at random points $u, v \in \mathbb{F}^{s'}$. When reaching layer $i+1$, $\mathcal{V}$ has many evaluations of multilinear extensions of *subsets* of $\mathbf{V}_{i+1}$. Now we cannot even use the original protocol to combine these points into one, as they are evaluations of *different* multilinear extensions, not to mention that we want to reduce the complexity of the prover time in this step. Our second technique is to compute them using a layered arithmetic circuit and reduce these evaluations to a single evaluation of $\tilde{V}_i$ through the original GKR protocol. At this point, the random points in these evaluations are already fixed by the verifier. We construct a circuit whose input is the values $\mathbf{V}_{i+1}$ of the entire layer $i+1$, and all the random points in the evaluations, denoted as $v^{(0)}, v^{(1)}, \ldots, v^{(i)}$ and $u^{(i)}$. The output of the circuit is exactly the evaluations of the multilinear extensions of the subsets, received from the sumcheck protocols for all layers below, i.e., $\tilde{V}_{0,i+1}(v^{(0)}), \tilde{V}_{1,i+1}(v^{(1)}), \ldots, \tilde{V}_{i,i+1}(v^{(i)})$ and $\tilde{V}_{i,i+1}(u^{(i)})$. To compute the output, the circuit selects all the subsets from input $\mathbf{V}_i$ and arrange them in the predefined order, which can be determined by the structure of the general circuit. The circuit then evaluates the multilinear extensions defined by these subsets at points from input $v^{(0)}, v^{(1)}, \ldots, v^{(i)}$ and $u^{(i)}$. By executing the original GKR protocol on this circuit, $\mathcal{P}$ and $\mathcal{V}$ reduce the correctness of $\tilde{V}_{0,i+1}(v^{(0)}), \tilde{V}_{1,i+1}(v^{(1)}), \ldots, \tilde{V}_{i,i+1}(v^{(i)})$ and $\tilde{V}_{i,i+1}(u^{(i)})$ to a single evaluation of the input. As part of the input $v^{(0)}, v^{(1)}, \ldots, v^{(i)}$ and $u^{(i)}$ are known to the verifier, it is easy to subtract it from the evaluation and obtain $\tilde{V}_i(w)$, a single evaluation of the multilinear extension $\tilde{V}_i$ at a random point $w \in \mathbb{F}^s$. With this single evaluation, $\mathcal{P}$ and $\mathcal{V}$ can continue the sumcheck for layer $i+1$ recursively and proceed all the way to the input layer. With proper design, we are able to bound the total size of this circuit in all rounds by $O(|C|)$. Therefore, the prover complexity in this step is also $O(|C|)$. See Figure 4.1 and Section 4.3.3 for the design of the circuit and the details of the protocol.

Furthermore, inspired by the structure of this circuit, we are able to design a single sumcheck protocol to combine multiple claims on the subsets to a single evaluation of $\tilde{V}_i$ at a random point. This second approach further improves the prover time, the proof size and the verifier time. Putting the two steps together, we are able to construct a generalized GKR protocol for arbitrary arithmetic circuits while maintaining the optimal prover time of $O(|C|)$.

Building zero knowledge arguments. Finally, following the framework of [XZZPS19b; ZXZS; ZGKPP17c; WTSTW18; CFS17], we build zero knowledge arguments for general arithmetic circuits using our new protocol. We use the standard techniques of zero knowledge sumcheck and low degree extensions in [CFS17; XZZPS19b] to lift our generalized GKR protocol to be zero knowledge, and use the polynomial commitment scheme in [ZXZS] to make the protocol a zero knowledge argument.

4.1.3 Related Work

Interactive proofs were formalized by Goldwasser, Micali, and Rackoff in [GMR89]. In the seminal work of [GKR15], Goldwasser et al. proposed the doubly efficient interactive proof for layered arithmetic circuits. Later, Cormode et al. improved the prover time of the GKR protocol from $O(|C|^3)$ to $O(|C|\log|C|)$ using multilinear extensions instead of low degree extensions in [CMT12]. Several follow-up papers further reduce the prover time for circuits with special structures. Justin Thaler [Tha13b] introduced a protocol with $O(|C|)$ prover time for regular circuits where the wiring pattern can be described in constant space and time. In the same work, a protocol with prove time $O(|C|\log|C'|)$ was proposed for data parallel circuits with many copies of small circuits of size $|C'|$. The complexity was further 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. [ZGKPP18] showed a protocol with $O(|C|\log|C'|)$ prover time. Eventually, Xie et al. [XZZPS19b] proposed a variant of the GKR protocol with $O(|C|)$ prover time for arbitrary layered arithmetic circuits. All these existing works follow the layered structure of the GKR protocol and doubly efficient interactive proofs for general arithmetic circuits have not been considered before.

In [ZGKPP17c], Zhang et al. extended the GKR protocol to an argument system using polynomial commitments. Subsequent works [WTSTW18; ZGKPP17a; XZZPS19b; ZXZS; Set20a] followed the framework and constructed efficient zero knowledge argument schemes based on interactive proofs. We follow the approach of [CFS17; XZZPS19b; ZXZS] and constructs zero knowledge arguments for general circuits instead of layered circuits. Notably, there is a recent work [Set20a] on constructing interactive proof-based zero knowledge arguments for R1CS. The protocol reduces the R1CS to a polynomial commitment on the entire extended witness of all the values in the circuit using one sumcheck. On the contrary, the GKR protocols reduce the evaluation of the circuit to only the input of the circuit. As the polynomial commitments are usually the overhead of the zero knowledge argument schemes, we expect that our scheme has faster prover time, while the scheme in [Set20a] has smaller proof size. We give detailed comparisons in Section 4.5.2. In addition, the scheme in [Set20a] cannot be used for delegation of computations, which is the original goal of the GKR protocols. In a recent manuscript [SL20], the proof size of the scheme in [Set20a] is improved from square-root to logarithmic in the size of the R1CS instance, but the prover time is $3.8\times$ slower. In a different setting, Blumberg et al. [BTVW14] construct argument schemes using interactive proofs with two provers.

There is a rich literature of zero knowledge arguments other than schemes based on interactive proofs. Categorized by their underlying techniques, there are schemes based on quadratic arithmetic programs (QAP) [PHGR13], interactive oracle proofs (IOP) [**ben2019aurora**], discrete-log [BBBPWM], MPC-in-the-head [AHIV17] and lattice [BBCDPGL18]. We refer the readers to surveys [WB15] and recent papers [Set20a] on zero knowledge proofs and arguments for a more comprehensive list of schemes.

4.2 Preliminaries

We use $\mathsf{negl}(\cdot) : \mathbb{N} \to \mathbb{N}$ to denote the negligible function, where for each positive polynomial $f(\cdot)$, $\mathsf{negl}(k) < \frac{1}{f(k)}$ for sufficiently large integer k. Let λ denote the security parameter. "PPT" stands for probabilistic polynomial time. We use $f(), g()$ 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 $\mathbf{A}$ to represent arrays. For a multivariate polynomial f, its "variable-degree" is the maximum degree of f in any of its variables.

4.2.1 Interactive Proofs

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 $C(x) = y$. We formalize interactive proofs in the following:

Definition 4.2.1. *Let C be a 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 $C(x) = y$ it holds that* $\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x) = accept] = 1$.
- **ϵ-Soundness.** *For any x with $C(x) \neq y$ and any $\mathcal{P}^*$ it holds that* $\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = accept] \leq \epsilon$

We say an interactive proof scheme has succinct proof size (verifier time) if the total communication (verifier time) is $O(\mathsf{polylog}(|C|, |x|))$.

4.2.2 Doubly Efficient Interactive Proofs for Layered Circuits

In [GKR15], Goldwasser et al. proposed an efficient interactive proof protocol for layered arithmetic circuits. We present the details of the protocol here.

4.2.2.1 Sumcheck Protocol

The GKR protocol uses the sumcheck protocol as a major building block. The problem is to sum a multivariate polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ on the Boolean hypercube: $\sum_{b_1, b_2, ..., b_\ell \in \{0,1\}} f(b_1, b_2, ..., b_\ell)$. 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 24.

Protocol 8 (**Sumcheck**). The protocol proceeds in ℓ rounds.

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

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

  $\mathcal{V}$ 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

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

  $\mathcal{V}$ 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

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

  $\mathcal{V}$ 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.

The proof size of the sumcheck protocol is $O(\tau\ell)$, where τ 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 $\tau + 1$ points. The verifier time of the protocol is $O(\tau\ell)$. 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 $\epsilon = \frac{\tau\ell}{|\mathbb{F}|}$.

Definition 4.2.2 (Multi-linear Extension). *Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. The multilinear extension of V is the unique polynomial $\tilde{V} : \mathbb{F}^\ell \to \mathbb{F}$ such that $\tilde{V}(x_1, x_2, ..., x_\ell) = V(x_1, x_2, ..., x_\ell)$ for all $x_1, x_2, \ldots, x_\ell \in \{0,1\}$. $\tilde{V}$ can be expressed as:*

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

where b_i is i-th bit of b.

Multilinear extensions of arrays. Inspired by the closed-form equation of the multilinear extension given above, we can view an array $\mathbf{A} = (a_0, a_1, \ldots, a_{n-1})$ as a function $A : \{0,1\}^{\log n} \to \mathbb{F}$ such that $\forall i \in [0, n-1], A(i_1, \ldots, i_{\log n}) = a_i$. Here we assume n is a power of 2. Therefore, in this paper, we abuse the use of multilinear extension on an array as the multilinear extension $\tilde{A}$ of A.

Definition 4.2.3 (Identity function). *Let $\beta : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}$ be the identity function such that $\beta(x, y) = 1$ if $x = y$, and $\beta(x, y) = 0$ otherwise. Suppose $\tilde{\beta}$ is the multilinear extension of β. Then $\tilde{\beta}$ can be expressed as: $\tilde{\beta}(x, y) = \prod_{i=1}^{\ell}((1 - x_i)(1 - y_i) + x_i y_i)$.*

GKR Protocol. Using the sumcheck protocol as a building block, Goldwasser et al. [GKR15] showed an interactive proof protocol for layered arithmetic circuits. 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 $\mathcal{V}$. If the check passes, $\mathcal{V}$ accepts the claimed output.

Notation. We follow the convention in prior works of GKR protocols [CMT12; Tha13b; ZGKPP17c; XZZPS19b; ZXZS]. 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 to 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, for any $z \in \{0,1\}^{s_i}$, V_i can be written as:

$$\begin{aligned} V_i(z) = \sum_{x,y\in\{0,1\}^{s_{i+1}}} &(add_{i+1}(z,x,y)(V_{i+1}(x) + V_{i+1}(y)) \\ &+ mult_{i+1}(z,x,y)V_{i+1}(x)V_{i+1}(y)) \end{aligned} \tag{4.1}$$

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 multilinear extensions:

$$\begin{aligned} \tilde{V}_i(g) =& \sum_{x,y\in\{0,1\}^{s_{i+1}}} f_i(g,x,y) \\ =& \sum_{x,y\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(g,x,y)(\tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(y)) \\ &+ \tilde{mult}_{i+1}(g,x,y)\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)) \,, \end{aligned} \tag{4.2}$$

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

Protocol. With Equation 7.1, 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 $\tilde{V}_0(g)$ for a random $g \in \mathbb{F}^{s_0}$. $\mathcal{V}$ and $\mathcal{P}$ then invoke a sumcheck protocol on Equation 7.1 with $i = 0$. As described in Section 7.2.2.1, at the end of the sumcheck, $\mathcal{V}$ needs an oracle access to $f_i(g,u,v)$, where u, v are randomly selected in $\mathbb{F}^{s_{i+1}}$. To compute $f_i(g,u,v)$, $\mathcal{V}$ computes $\tilde{add}_{i+1}(g,u,v)$ and $\tilde{mult}_{i+1}(g,u,v)$ locally (they only depend on the wiring pattern of the circuit, not on the values), asks $\mathcal{P}$ to send $\tilde{V}_1(u)$ and $\tilde{V}_1(v)$ and computes $f_i(g,u,v)$ 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 $\tilde{V}_1(u)$ and $\tilde{V}_1(v)$ recursively to layers above, but the number of the sumcheck protocols would increase exponentially.

Combining two claims using a random linear combination. One way to combine two claims $\tilde{V}_i(u)$ and $\tilde{V}_i(v)$ is using random linear combinations, as proposed in [CFS17; WTSTW18]. Upon receiving the two claims $\tilde{V}_i(u)$ and $\tilde{V}_i(v)$, $\mathcal{V}$ selects $\alpha_{i,1}, \alpha_{i,2} \in \mathbb{F}$ randomly and computes $\alpha_{i,1}\tilde{V}_i(u) + \alpha_{i,2}\tilde{V}_i(v)$. Based on Equation 7.1, this random linear combination can be written as

$$\begin{aligned} &\alpha_{i,1}\tilde{V}_i(u) + \alpha_{i,2}\tilde{V}_i(v) \\ =&\alpha_{i,1} \sum_{x,y\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(u,x,y)(\tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(y)) + \tilde{mult}_{i+1}(u,x,y)\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)) \\ +&\alpha_{i,2} \sum_{x,y\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(v,x,y)(\tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(y)) + \tilde{mult}_{i+1}(v,x,y)\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)) \\ =& \sum_{x,y\in\{0,1\}^{s_{i+1}}} ((\alpha_{i,1}\tilde{add}_{i+1}(u,x,y) + \alpha_{i,2}\tilde{add}_{i+1}(v,x,y))(\tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(y)) \\ &+ (\alpha_{i,1}\tilde{mult}_{i+1}(u,x,y) + \alpha_{i,2}\tilde{mult}_{i+1}(v,x,y))\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)) \end{aligned} \tag{4.3}$$

$\mathcal{V}$ and $\mathcal{P}$ then execute the sumcheck protocol on Equation 6.3 instead of Equation 7.1. 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. With the optimal algorithms with a linear prover time proposed in [XZZPS19b], we have the following theorem:

Protocol 9. Let $\mathbb{F}$ be a finite field. Let C: $\mathbb{F}^n \to \mathbb{F}^k$ be a d-depth layered arithmetic circuit. $\mathcal{P}$ wants to convince that **out** $= C($**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.

1. 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 $\tilde{V}_0(g)$.

2. $\mathcal{P}$ and $\mathcal{V}$ run a sumcheck protocol on

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

   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 $\tilde{add}_1(g^{(0)},u^{(1)},v^{(1)})$ $(\tilde{V}_1(u^{(1)})+\tilde{V}_1(v^{(1)}))+\tilde{mult}_1(g^{(0)},$ $u^{(1)},v^{(1)})$ $\tilde{V}_1(u^{(1)})\tilde{V}_1(v^{(1)})$ equals to the last message of the sumcheck.

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

$$\alpha_{i,1}\tilde{V}_i(u^{(i)}) + \alpha_{i,2}\tilde{V}_i(v^{(i)}) =$$
$$\sum_{x,y\in\{0,1\}^{s_{i+1}}} ((\alpha_{i,1}\tilde{add}_{i+1}(u^{(i)},x,y) + \alpha_{i,2}\tilde{add}_{i+1}(v^{(i)},x,y))(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))$$
$$+(\alpha_{i,1}\tilde{mult}_{i+1}(u^{(i)},x,y) + \alpha_{i,2}\tilde{mult}_{i+1}(v^{(i)},x,y))\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y))$$

   - 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 following and checks if it equals to the last message of the sumcheck. For simplicity, let $Mult_{i+1}(x) = \tilde{mult}_{i+1}(x,u^{(i+1)},v^{(i+1)})$ and $Add_{i+1}(x) = \tilde{add}_{i+1}(x,u^{(i+1)},v^{(i+1)})$.

$$(\alpha_{i,1}Mult_{i+1}(u^{(i)}) + \alpha_{i,2}Mult_{i+1}(v^{(i)})(\tilde{V}_{i+1}(u^{(i+1)})\tilde{V}_{i+1}(v^{(i+1)}))+$$
$$(\alpha_{i,1}Add_{i+1}(u^{(i)}) + \alpha_{i,2}Add_{i+1}(v^{(i)})(\tilde{V}_{i+1}(u^{(i+1)}) + \tilde{V}_{i+1}(v^{(i+1)}))$$

   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 `0` and aborts.

4. At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(u^{(d)})$ and $\tilde{V}_d(v^{(d)})$. $\mathcal{V}$ evaluates $\tilde{V}_d$ at $u^{(d)}$ and $v^{(d)}$ using the input and checks that they are the same as the two claims. If yes, output `1`; otherwise, output `0`.

Theorem 4.2.4. *[XZZPS19b]. Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d layered arithmetic circuit. Protocol 9 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 the running time of the prover $\mathcal{P}$ is $O(|C|)$. Let T be the time to evaluate all $\tilde{add}_i$ and $\tilde{mult}_i$ at the corresponding random points, the running time of $\mathcal{V}$ is $O(n + k + d \log |C| + T)$.*

4.3 Generalizing GKR to Arbitrary Arithmetic Circuits

Though the GKR protocol has great prover efficiency as demonstrated in [XZZPS19b; ZXZS; Tha13b; Wah+17] and is used as a major building block to construct fast zero knowledge proof schemes, one major drawback is that the protocol only works for layered arithmetic circuits, i.e., each gate can only take input from the layer above. In this section, we show how to generalize the GKR protocol to arbitrary circuits with no overhead on the prover time.

We consider a general arithmetic circuit C with fan-in 2, which can be viewed as a directed acyclic graph (DAG), G_C. Each gate in C is a vertex in G_C and each wire is a directed edge in G_C. The in-degree of each vertex is at most 2. The depth of the circuit d is defined as the length of the longest path in the DAG. Without loss of generality, we assume that all input gates are at layer d, and all output gates are at layer 0.[1] Following the order to evaluate the circuit, we can actually assign a layer number to each gate topologically. In particular, if gate g is not an input, suppose gate u and gate v are the input gates of g, then $\mathsf{layer}(g) = \min(\mathsf{layer}(u), \mathsf{layer}(v)) - 1$, where the function $\mathsf{layer}(x)$ represents the layer of the gate x. Because of this definition, an interesting observation is that a gate at layer i must take at least one input from layer $i + 1$, otherwise it cannot be labeled as in layer i. Also obviously, a gate at layer i can only take input from layer j such that $j > i$.

Same as the original GKR protocol, we use S_i as the number of gates in the i-th layer and $s_i = \lceil \log S_i \rceil$. For simplicity, we assume S_i is a power of 2, and we can pad the layer with dummy gates otherwise. The function V_i takes a binary string b and outputs the b-th gate value in layer i of C. As now every gate can take input from any layer above, we generalize the notations naturally and define $add_{i,j}, mult_{i,j} : \{0,1\}^{s_{i-1}, s_i, s_j} \to \{0,1\}$ as the wiring-predicate functions for the general circuit C. $add_{i,j}$ takes one gate label $z \in \{0,1\}^{s_{i-1}}$ in layer $i - 1$, one gate label $x \in \{0,1\}^{s_i}$ in layer i and one gate label $y \in \{0,1\}^{s_j}$ in layer j for $j \geq i$, and outputs 1 if and only if gate z is an addition gate that takes the output of gate x, y as input. $mult_{i,j}$ is defined similarly for multiplication gates. We still use $\tilde{f}$ to represent the multilinear extensions of the function f.

4.3.1 A Naive Generalization of the GKR Protocol

With these definitions, we first describe a naive generalization of the GKR protocol to general arithmetic circuits. We follow the core idea of the GKR protocol to reduce the claim about V_i layer by layer via the sumcheck protocol. In a general circuit, a gate in layer i can take the output of any gate in layer $i + 1$ to d, thus we simply extend Equation 7.1 to have one add and one $mult$ for each layer above. Recall from above

[1] Note that as we support general circuits, it takes at most one relay gate per input/output to transform an arbitrary circuit to a circuit with such property. Thus the overhead is small and we assume so for simplicity.

that every gate at layer i must have at least one input from layer $i+1$, we assume that this is the left input and rewrite the sumcheck equation in Equation 7.1 as:

$$\begin{aligned}\tilde{V}_i(g) = &\sum_{x\in\{0,1\}^{s_{i+1}}} (\sum_{y\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1,i+1}(g,x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))\\ &+\sum_{y\in\{0,1\}^{s_{i+2}}} \tilde{add}_{i+1,i+2}(g,x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_{i+2}(y))\\ &+\ldots+\sum_{y\in\{0,1\}^{s_d}} \tilde{add}_{i+1,d}(g,x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_d(y))\\ &+\sum_{y\in\{0,1\}^{s_{i+1}}} \tilde{mult}_{i+1,i+1}(g,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y))\\ &+\sum_{y\in\{0,1\}^{s_{i+2}}} \tilde{mult}_{i+1,i+2}(g,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+2}(y))\\ &+\ldots+\sum_{y\in\{0,1\}^{s_d}} \tilde{mult}_{i+1,d}(g,x,y)(\tilde{V}_{i+1}(x)\tilde{V}_d(y)))\end{aligned} \tag{4.4}$$

for any $g\in\mathbb{F}^{s_i}$. With this equation, starting from the output layer, in round i, the first step is that $\mathcal{P}$ and $\mathcal{V}$ engage the sumcheck protocol on Equation 4.4 to reduce one claim about layer i to claims about previous layers. At the end of the sumcheck protocol, $\mathcal{P}$ sends $\mathcal{V}$ evaluations of $\tilde{V}_{i+1}(u), \tilde{V}_{i+1}(v), \tilde{V}_{i+2}(v), \ldots, \tilde{V}_d(v)$ on the randomness of u and v. $\mathcal{V}$ evaluates all add and $mult$ on her own and completes the last round of the sumcheck protocol.

In the second step, when going to a new layer, $\mathcal{P}$ and $\mathcal{V}$ need to combine multiple claims about this layer. Here in the naive approach, we use the same method of random linear combinations. When reaching layer i, $\mathcal{V}$ has received the claims about $\tilde{V}_i$ from layer $0, 1, \ldots, i-1$ (twice for $i-1$). Denote the randomness of these claims as $g^{(0)}, g^{(1)}, \ldots, g^{(i)}$. $\mathcal{V}$ picks a random number $\alpha_{i,j}$ for each claim, and we can rewrite Equation 4.4 as:

$$\begin{aligned}&\alpha_{i,0}\tilde{V}_i(g^{(0)})+\alpha_{i,1}\tilde{V}_i(g^{(1)})+\ldots+\alpha_{i,i}\tilde{V}_i(g^{(i)})\\ =&\sum_{j=0}^{i}\alpha_{i,j}(\sum_{x\in\{0,1\}^{s_{i+1}}}(\sum_{y\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1,i+1}(g^{(j)},x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))\\ &+\ldots+\sum_{y\in\{0,1\}^{s_d}} \tilde{add}_{i+1,d}(g^{(j)},x,y)(\tilde{V}_{i+1}(x)+\tilde{V}_d(y))\\ &+\sum_{y\in\{0,1\}^{s_{i+1}}} \tilde{mult}_{i+1,i+1}(g^{(j)},x,y)(\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y))\\ &+\ldots+\sum_{y\in\{0,1\}^{s_d}} \tilde{mult}_{i+1,d}(g^{(j)},x,y)(\tilde{V}_{i+1}(x)\tilde{V}_d(y))))\end{aligned} \tag{4.5}$$

$\mathcal{V}$ and $\mathcal{P}$ then execute the sumcheck protocol on Equation 4.5 instead of Equation 4.4. At the end of the sumcheck protocol, $\mathcal{V}$ still receives claims about $\tilde{V}_{i+1}, \tilde{V}_{i+2}, \ldots, \tilde{V}_d$. For layer $i+1$, $\mathcal{V}$ computes their random linear combination and proceeds to the sumcheck protocol for layer $i+1$ recursively.

This protocol is a direct generalization of the GKR protocol in Protocol 9, and it is not hard to see that the protocol is sound. Unfortunately, it introduces a big overhead on the prover time. First, in the beginning of the sumcheck protocol on Equation 4.4, the equation is defined over the multilinear extensions $\tilde{V}_{i+1}, \tilde{V}_{i+2}, \ldots, \tilde{V}_d$. Hence, the prover time in this step is at least $O(S_{i+1}+S_{i+2}+\ldots+S_d)$. In fact, merely listing these polynomials and evaluating them at random points already take $O(S_{i+1}+S_{i+2}+\ldots+S_d)$ time, not to mention the prover time of the sumcheck protocol. Therefore, the total prover time is $O(dS_d+(d-1)S_{d-1}+\ldots+S_1) = O(d|C|)$ for all layers. There is a multiplicative overhead of d on the prover time, which is in fact as bad as transforming the general circuit to a layered circuit. Second, in the step of random linear combinations, as shown in Equation 4.5, $\mathcal{V}$ combines $i+1$ claims together for layer i. On the right hand side of the equation, each $\tilde{add}$ and $\tilde{mult}$ has to be evaluated on $i+1$ different random points $g^{(j)}$. This again introduces a prover time of

$O(d|C|)$. Therefore, overall the prover time of this naive generalized GKR protocol is $O(d|C|)$, as slow as naively transforming the general circuit to a layered circuit.

In the next two subsections, we will show how to remove the overhead of each of the two steps.

4.3.2 Sumcheck with Linear Prover Time

As explained above, the main overhead of the sumcheck on Equation 4.4 in the first step comes from the fact that each layer can connect to all layers above in a general circuit, and defining $\tilde{V}_{i+1}, \tilde{V}_{i+2}, \ldots, \tilde{V}_d$ already blows up the complexity. Therefore, instead of using the multilinear extension of the entire layer, we define the multilinear extension of *only those gates used in layer* i from a previous layer. As each gate only has two input gates, there are at most $2S_i$ gates connecting to gates in layer i in total. In this way, the total size of these multilinear extensions is bounded by $O(S_i)$.

Formally speaking, we also generalize the definitions of S and s such that $S_{i,j}$ denotes the number of gates connecting from layer j ($j > i$) to layer i, and $s_{i,j} = \lceil \log S_{i,j} \rceil$. We then introduce a new function $V_{i,j} : \{0,1\}^{s_{i,j}} \to \mathbb{F}$, which is defined by the subset of gates from layer j connecting to layer i in a pre-defined order. The function takes a binary string $b \in \{0,1\}^{s_{i,j}}$ and returns the b-th value in this subset. We also re-define $add_{i,j}, mult_{i,j} : \{0,1\}^{s_{i-1}+s_{i-1,i}+s_{i-1,j}} \to \{0,1\}$ to take labels from the subsets instead of the labels of the entire layers. In particular, $add_{i,j}(z,x,y) = 1$ ($multi_{i,j}(z,x,y) = 1$) if and only if gate z in layer $i-1$ is the addition (multiplication) of value $V_{i-1,i}(x)$ and $V_{i-1,j}(y)$. With these definitions, by taking their multilinear extensions, we can rewrite Equation 4.4 as

$$\begin{aligned}
\tilde{V}_i(g) = &\sum\nolimits_{x\in\{0,1\}^{s_{i,i+1}}} \Big(\sum_{y\in\{0,1\}^{s_{i,i+1}}} \tilde{add}_{i+1,i+1}(g,x,y)(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,i+1}(y)) + \\
&+ \ldots + \sum\nolimits_{y\in\{0,1\}^{s_{i,d}}} \tilde{add}_{i+1,d}(g,x,y)(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,d}(y)) \\
&+ \sum\nolimits_{y\in\{0,1\}^{s_{i,i+1}}} \tilde{mult}_{i+1,i+1}(g,x,y)(\tilde{V}_{i,i+1}(x)\tilde{V}_{i,i+1}(y)) \\
&+ \ldots + \sum\nolimits_{y\in\{0,1\}^{s_{i,d}}} \tilde{mult}_{i+1,d}(g,x,y)(\tilde{V}_{i,i+1}(x)\tilde{V}_{i,d}(y))\Big)
\end{aligned} \tag{4.6}$$

In Equation 4.6, the size of $\tilde{V}_{i,i+1}, \ldots, \tilde{V}_{i,d}$ are bounded by $O(S_i)$. Moreover, the $\tilde{add}$ and $\tilde{mult}$ polynomials are still sparse. In fact, the total number of nonzeros in all $\tilde{add}$ and $\tilde{mult}$ together is S_i. Therefore, using similar ideas proposed in [XZZPS19b], we are able to develop an algorithm for the prover to run the sumcheck in linear time $O(S_i)$, instead of $O(S_i + S_{i+1} + \ldots + S_d)$.

Before presenting the linear-time algorithm, we make one more refinement on the equation. Note that Equation 4.6 consists of multiple sums, because the number of gates connecting from layer $j > i$ to layer i is different for each j. We cannot pad them to the same length, as it would introduce an overhead asymptotically. We combine them into a single sum in the following way. Without loss of generality, we suppose $s_{i,i+1}$ is the

largest. We can then rewrite Equation 4.6 as:

$$\begin{aligned}
\tilde{V}_i(g) = &\sum_{x,y\in\{0,1\}^{s_{i,i+1}}} (\tilde{add}_{i+1,i+1}(g,x,y)(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,i+1}(y_1,\ldots,y_{s_{i,i+1}}))\\
&+ y_{s_{i,i+2}+1}\cdot\ldots\cdot y_{s_{i,i+1}} \tilde{add}_{i+1,i+2}(g,x,y_1,\ldots,y_{s_{i,i+2}})(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,i+2}(y_1,\ldots,y_{s_{i,i+2}}))\\
&+ \ldots + y_{s_{i,d}+1}\cdot\ldots\cdot y_{s_{i,i+1}} \tilde{add}_{i+1,d}(g,x,y_1,\ldots,y_{s_{i,d}})(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,d}(y_1,\ldots,y_{s_{i,d}}))\\
&+ \tilde{mult}_{i+1,i+1}(g,x,y)(\tilde{V}_{i,i+1}(x)\tilde{V}_{i,i+1}(y_1,\ldots,y_{s_{i,i+1}}))\\
&+ y_{s_{i,i+2}+1}\cdot\ldots\cdot y_{s_{i,i+1}} \tilde{mult}_{i+1,i+2}(g,x,y_1,\ldots,y_{s_{i,i+2}})\tilde{V}_{i,i+1}(x)\tilde{V}_{i,i+2}(y_1,\ldots,y_{s_{i,i+2}})\\
&+ \ldots + y_{s_{i,d}+1}\cdot\ldots\cdot y_{s_{i,i+1}} \tilde{mult}_{i+1,d}(g,x,y_1,\ldots,y_{s_{i,d}})\tilde{V}_{i,i+1}(x)\tilde{V}_{i,d}(y_1,\ldots,y_{s_{i,d}}))
\end{aligned} \tag{4.7}$$

Note that the only difference between Equation 4.6 and 4.7 is that in Equation 4.7 all the sums are over $y \in \{0,1\}^{s_{i,i+1}}$, the longest binary string. For $j = i+2,\ldots,d$, as $\tilde{add}_{i+1,j}, \tilde{mult}_{i+1,j}$ and $\tilde{V}_{i,j}$ only take $y_1,\ldots,y_{s_{i,j}}$, we multiply each term with $y_{s_{i,j}+1}\cdot y_{s_{i,j}+2}\cdot\ldots\cdot y_{s_{i,i+1}}$. This guarantees that the term only appears once in the sum, when $y_{s_{i,j}+1} = y_{s_{i,j}+2} = \ldots = y_{s_{i,i+1}} = 1$, and thus Equation 4.7 holds. In fact, $y_{s_{i,j}+1}\cdot y_{s_{i,j}+2}\cdot\ldots\cdot y_{s_{i,i+1}}$ is exactly the identity polynomial $\tilde{\beta}((y_{s_{i,j}+1}, y_{s_{i,j}+2},\ldots,y_{s_{i,i+1}}), \vec{1})$. In this way, we do not have to pad all the polynomials to the same size. We only pad the size of each subset to the nearest power of 2, which incurs at most an overhead of 2.

Next, we present an algorithm for $\mathcal{P}$ to run the sumcheck protocol on Equation 4.7 in linear time. We start with an algorithm to run sumcheck for the product of two multilinear polynomials in the literature, which we will use as a major building block.

Linear-time sumcheck for products of multilinear functions [Tha13b]. In [Tha13b], Thaler proposed a linear-time algorithm for the prover of the sumcheck protocol on the product of two multilinear polynomials f and g with ℓ variables (the algorithm runs in $O(2^\ell)$ time). We present the algorithm in Algorithm 9. Algorithm 9 invokes Algorithm 8 FunctionEvaluations() as a subroutine. The algorithms are exactly the same as Algorithm 1 and 3 in [XZZPS19b]. Both Algorithm 8 and Algorithm 9 run in time $O(2^\ell)$ and the formal proof can be found in [XZZPS19b; Tha13b]. We have a lemma as follows:

Lemma 4.3.1. *Given multilinear functions f and g on ℓ variables and a* bookkeeping table $\boldsymbol{A}_f$ *for f and a* bookkeeping table $\boldsymbol{A}_g$ *for g, the prover in Protocol 24 for $f\cdot g$ runs in $O(2^\ell)$ time. $\boldsymbol{A}_f = (f(0,\ldots,0),\ldots,f(1,\ldots,1))$ and $\boldsymbol{A}_g = (g(0,\ldots,0),\ldots,g(1,\ldots,1))$ are initialized with evaluations of f and g on the Boolean hypercube, respectively.*

Linear-time sumcheck for Equation 4.7. The idea of the prover algorithm is similar to that proposed in [XZZPS19b]. The algorithm proceeds in two phases, one summing x and the other summing y. For the ease of presentation, let us consider the sumcheck on a particular class of equations:

$$\begin{aligned}
\sum_{x,y\in\{0,1\}^\ell} &y_{k_1+1}\ldots y_\ell f_1(g,x,y_1,\ldots,y_{k_1}) s_1(y_1,\ldots,y_{k_1}) t(x) +\\
&y_{k_2+1}\ldots y_\ell f_2(g,x,y_1,\ldots,y_{k_2}) s_2(y_1,\ldots,y_{k_2}) t(x) + \ldots +\\
&y_{k_m+1}\ldots y_\ell f_m(g,x,y_1,\ldots,y_{k_m}) s_m(y_1,\ldots,y_{k_m}) t(x)\,,
\end{aligned} \tag{4.8}$$

for a fixed point $g \in \mathbb{F}^\ell$, where $t(x) : \mathbb{F}^\ell \to \mathbb{F}$ and $s_i(x) : \mathbb{F}^{k_i} \to \mathbb{F}$ are multilinear extensions of arrays $\mathbf{A}_t$ and $\mathbf{A}_{s_i}$, and all functions of $f_i(x) : \mathbb{F}^{2\ell+k_i} \to \mathbb{F}$ are multilinear extensions of sparse arrays with $O(2^\ell)$ nonzero elements in total. In addition, we require that $2^{k_1} + 2^{k_2} + \ldots + 2^{k_m} = 2^\ell$. It is not hard to see

Algorithm 8 $\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}, r_1, \ldots, r_\ell)$

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 6
10: **return** $\mathcal{F}$

Algorithm 9 $\{a_1, \ldots, a_\ell\} \leftarrow \mathsf{SumCheckProduct}(f, \mathbf{A}_f, g, \mathbf{A}_g, r_1, \ldots, r_\ell)$

Input: Multilinear f and g, initial bookkeeping tables $\mathbf{A}_f$ and $\mathbf{A}_g$, random $r_1, \ldots, r_\ell$;
Output: ℓ 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});

1: $\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}_f, r_1, \ldots, r_\ell)$
2: $\mathcal{G} \leftarrow \mathsf{FunctionEvaluations}(g, \mathbf{A}_g, r_1, \ldots, r_\ell)$
3: **for** $i = 1, \ldots, \ell$ **do**
4: **for** $t \in \{0, 1, 2\}$ **do**
5: $a_{it} = \sum_{b \in \{0,1\}^{\ell-i}} f(r_1, \ldots, r_{i-1}, t, b) \cdot g(r_1, \ldots, r_{i-1}, t, b)$ *// All evaluations needed are in $\mathcal{F}$ and $\mathcal{G}$.*
6: **end for**
7: **end for**
8: **return** $\{a_1, \ldots, a_\ell\}$;

that Equation 4.7 satisfies these properties, as there are at most S_i left input gates and S_i right input gates connected to layer i in the circuit C. If we set $\ell = s_i = \lceil \log S_i \rceil$, we have $2^{k_1} + 2^{k_2} + \ldots + 2^{k_m} = O(S_i)$ in Equation 4.7.

We use the same intuition in [XZZPS19b] of dividing the sumcheck process into two phases, one is for x and the other is for y. We rewrite Equation 4.8 as follows $\sum_{x \in \{0,1\}^\ell} t(x) h_g(x)$, where

$$\begin{aligned} h_g(x) = \sum\nolimits_{y \in \{0,1\}^\ell} & (y_{k_1+1} \ldots y_\ell f_1(g, x, y_1, \ldots, y_{k_1}) s_1(y_1, \ldots, y_{k_1}) + \\ & y_{k_2+1} \ldots y_\ell f_2(g, x, y_1, \ldots, y_{k_2}) s_2(y_1, \ldots, y_{k_2}) + \ldots + \\ & y_{k_m+1} \ldots y_\ell f_m(g, x, y_1, \ldots, y_{k_m}) s_m(y_1, \ldots, y_{k_m})) \end{aligned} \tag{4.9}$$

Algorithm 10 $\mathbf{A}_{h_g} \leftarrow$ Initialize_PhaseOne$(f_1, \ldots, f_m, s_1, \ldots, s_m, \mathbf{A}_{s_1}, \ldots, \mathbf{A}_{s_m}, g)$

Input: Multilinear $f_1, \ldots, f_m$ and $s_1, \ldots, s_m$, initial bookkeeping tables $\mathbf{A}_{s_1}, \ldots, \mathbf{A}_{s_m}$, random $g = g_1, \ldots, g_\ell$; We have $|\mathbf{A}_{s_1}| + \ldots + |\mathbf{A}_{s_m}| = 2^\ell$.
Output: Bookkeeping table $\mathbf{A}_{h_g}$;

procedure $\mathbf{G} \leftarrow$ Precompute(g) *// $\mathbf{G}$ is an array of size 2^ℓ.*
  Set $\mathbf{G}[0] = 1$
  for $i = 0, \ldots, \ell - 1$ **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 $f_i'(z, x, y)$ is non-zero **do**
  $\mathbf{A}_{h_g}[x] = \mathbf{A}_{h_g}[x] + \mathbf{G}[z] \cdot f_i'(z, x, y) \cdot \mathbf{A}_{s_i}[y_1, \ldots, y_{k_i}]$
end for
return $\mathbf{A}_{h_g}$;

Phase one. With the formula above, in the first ℓ rounds, the prover and the verifier are running exactly a sumcheck on the product of two multilinear polynomials $t(x) \cdot h_g(x)$, since functions t and h_g can be viewed as functions only in x, and y can be considered constant (it is always summed on the Boolean hypercube). To compute the sumcheck messages for the first ℓ rounds, given their bookkeeping tables, this will take $O(2^\ell)$ time by Lemma 4.3.1. It remains to show how to initialize the bookkeeping tables in linear time.
Initializing the bookkeeping tables:
Initializing the bookkeeping table for t in $O(2^\ell)$ time is trivial, since t 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 take advantage of the sparsity of f_i.

Lemma 4.3.2. *Let $\mathcal{N}_x$ be the set of $(z, y) \in \{0,1\}^{2\ell}$ such that $f_i'(z, x, y) = y_{k_i+1} \ldots y_\ell \, f_i(z, x, y_1, \ldots, y_{k_i})$ is non-zero for some $1 \le i \le m$. Then for all $x \in \{0,1\}^\ell$, it is $h_g(x) = \sum_{(z,y)\in\mathcal{N}_x} \tilde{\beta}(g, z) \cdot (\sum_{i=1}^m f_i'(z, x, y) \cdot s_i(y_1, \ldots, y_{k_i}))$.*

Proof. Since f_i is a multilinear extension, as shown in [Tha13b], we have $f_i'(g, x, y) = \sum_{z\in\{0,1\}^\ell} \tilde{\beta}(g, z) f_i'(z, x, y)$. Therefore,

$$\begin{aligned} h_g(x) &= \sum\nolimits_{z\in\{0,1\}^\ell} \tilde{\beta}(g, z) \cdot (\sum\nolimits_{i=1}^m f_i'(z, x, y) \cdot s_i(y_1, \ldots, y_{k_i})) \\ &= \sum\nolimits_{(z,y)\in\mathcal{N}_x} \tilde{\beta}(g, z) \cdot (\sum\nolimits_{i=1}^m f_i'(z, x, y) \cdot s_i(y_1, \ldots, y_{k_i})) \end{aligned}$$

Lemma 4.3.3. *The bookkeeping table $\mathbf{A}_{h_g}$ can be initialized in time $O(2^\ell)$.*

Proof. As f_i is sparse, $\sum_{x\in\{0,1\}^\ell} |\mathcal{N}_x| = O(2^\ell)$. From Lemma 4.3.2, given the evaluations of $\tilde{\beta}(g,z)$ for all $z \in \{0,1\}^\ell$, the prover can iterate through all $(z,y) \in \mathcal{N}_x$ for all x to compute $\mathbf{A}_{h_g}$. The full algorithm is presented in Algorithm 10. Since each s_i is the multilinear extension of an array, its evaluations on the Boolean hypercube are known. Therefore, we use $\mathbf{A}_{s_1}, \ldots, \mathbf{A}_{s_m}$ as the input of Algorithm 10. $|\mathbf{A}_{s_1}| + \ldots + |\mathbf{A}_{s_m}| = 2^\ell$ as $2^{k_1} + 2^{k_2} + \ldots + 2^{k_m} = 2^\ell$.

Procedure $\mathsf{Precompute}(g)$ is to evaluate $\mathbf{G}[z] = \tilde{\beta}(g,z) = \prod_{i=1}^{\ell}((1-g_i)(1-z_i)+g_iz_i))$ for $z \in \{0,1\}^\ell$. By the closed-form of $\tilde{\beta}(g,z)$, 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 4.3.2. When f_i' is represented as a map of $((z,x,y), f_i'(z,x,y))$ for non-zero values, the complexity of these steps is $O(2^\ell)$ since $\sum_{x\in\{0,1\}^\ell} |\mathcal{N}_x| = O(2^\ell)$.

In Protocol 11, the map above 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 $f_i'(z,x,y) = 1$.

With the bookkeeping tables, the prover runs Algorithm 9 for the product of multilinear polynomials and the total complexity for phase one is $O(2^\ell)$.

Phase two. At this point, the variable x is bounded to random numbers $u \in \mathbb{F}^\ell$. In the second phase, the equation to sum on becomes

$$t(u)\sum_{y\in\{0,1\}^\ell}(\sum_{i=1}^{m} y_{k_i+1}\ldots y_\ell f_i(g,u,y_1,\ldots,y_{k_i})s_i(y_1,\ldots,y_{k_i}))$$

Note here that $t(u)$ is merely a constant which the prover already computed in phase one. For the part behind the summation symbol on y, it has m products of two multilinear functions to sum. If we naively apply Algorithm 9 to each product, the prover runs in $O(m\cdot 2^\ell)$ time instead of only $O(2^\ell)$. Fortunately, we observe that we can merge some products dynamically during the sumcheck process, which reduces the number of products and removes the m factor in the complexity. To achieve the linear prover time, we generalize Lemma 4.3.1 to Lemma 4.3.4 for the summation of multiple products of multilinear functions.

Lemma 4.3.4. *Suppose we have $2m$ multilinear functions $f_1, f_2, \ldots, f_m$ and $g_1, g_2, \ldots, g_m$. Both g_i and f_i have k_i variables. Without loss of generality, suppose $\ell \geq k_m \geq k_{m-1} \geq k_1$. If $2^{k_1} + 2^{k_2} + \ldots + 2^{k_m} = 2^\ell$, given the* bookkeeping *tables $\mathbf{A}_{f_1}, \ldots, \mathbf{A}_{f_m}$ for $f_1, \ldots, f_m$ and $\mathbf{A}_{g_1}, \cdot, \mathbf{A}_{g_m}$ for $g_1, \ldots, g_m$, the prover in Protocol 24 for $\sum_{i=1}^{m}\sum_{y\in\{0,1\}^{k_i}} f_i(y)\cdot g_i(y) = \sum_{y\in\{0,1\}^\ell}\sum_{i=1}^{m} y_{k_i+1}\ldots y_\ell f_i(y_1,\ldots,y_{k_i})\cdot g_i(y_1,\ldots,y_{k_i})$ runs in $O(2^\ell)$ time.*

Proof. We present Algorithm 11 for the prover in the sumcheck. $\mathcal{P}$ runs in $O(2^\ell)$ for step 1-3 as $|\mathbf{A}_{f_1}| + \ldots + |\mathbf{A}_{f_m}| = |\mathbf{A}_{g_1}| + \ldots + |\mathbf{A}_{g_m}| = 2^\ell$. $\mathcal{P}$ runs in $O(2^\ell)$ for step 5-12 as the total number of the operations is $O(2^{k_1} + 2^{k_2} + \ldots + 2^{k_m}) = O(2^\ell)$. So $\mathcal{P}$ runs in $O(2^\ell)$ time for Algorithm 11.

The sumcheck polynomial for phase two has the same form in Lemma 4.3.4. To compute the sumcheck messages for the last ℓ rounds, given their bookkeeping tables, this will take $O(2^\ell)$ time by Lemma 4.3.4. We now show how to initialize the bookkeeping tables in linear time.

Initializing the bookkeeping tables:

Initializing the bookkeeping table for each s_i in $O(2^{k_i})$ time is trivial, since each s_i is a multilinear extension of

Algorithm 11 $\{a_1, \ldots, a_\ell\} \leftarrow$
SumCheckProduct2$(f_1, \mathbf{A}_{f_1}, g_1, \mathbf{A}_{g_1}, \ldots, f_m, \mathbf{A}_{f_m}, g_m, \mathbf{A}_{g_m}, r_1, \ldots, r_\ell)$

Input: Multilinear f_i and g_i, initial bookkeeping tables $\mathbf{A}_{f_i}$ and $\mathbf{A}_{g_i}$ for $i = 1$ to m, random $r_1, \ldots, r_\ell$; We have $|\mathbf{A}_{f_1}| + \ldots + |\mathbf{A}_{f_m}| = |\mathbf{A}_{g_1}| + \ldots + |\mathbf{A}_{g_m}| = 2^\ell$.

Output: ℓ sumcheck messages for $\sum_{y \in \{0,1\}^\ell} \sum_{i=1}^{m} y_{k_i+1} \ldots y_\ell f_i(y_1, \ldots, y_{k_i}) \cdot g_i(y_1, \ldots, y_{k_i})$. Each message a_i consists of 3 elements (a_{i0}, a_{i1}, a_{i2});

for $i = 1, \ldots, m$ **do**
  $\mathcal{F}_i \leftarrow$ FunctionEvaluations$(f_i, \mathbf{A}_{f_i}, r_1, \ldots, r_{k_i})$
  $\mathcal{G}_i \leftarrow$ FunctionEvaluations$(g_i, \mathbf{A}_{g_i}, r_1, \ldots, r_{k_i})$
end for
$temp = 0$
for $i = 0, \ldots, m$ **do**
  if $i > 0$ **then**
    $temp = temp + f_i(r_1, \ldots, r_{k_i}) \cdot g_i(r_1, \ldots, r_{k_i})$
  end if
  for $j = k_i + 1, \ldots, k_{i+1}$ **do** *// Suppose $k_0 = 0 < k_1 \leq \ldots \leq k_m \leq k_{m+1} = \ell$*
    if $j \leq \ell$ **then**
      for $q \in \{0, 1, 2\}$ **do**
        $a_{jq} = \sum_{t=i+1}^{m} \sum_{b \in \{0,1\}^{k_t - j}} f_t(r_1, \ldots, r_{j-1}, q, b) \cdot g_t(r_1, \ldots, r_{j-1}, q, b) + q \cdot temp$
        // All evaluations needed are in $\mathcal{F}_i$ and $\mathcal{G}_i$.
        $temp = temp \cdot r_j$
      end for
    end if
  end for
end for
return $\{a_1, \ldots, a_\ell\}$;

an array and therefore the evaluations on the hypercube are known. We also know $2^{k_1} + 2^{k_2} + \ldots + 2^{k_m} = O(2^\ell)$. It remains to initialize bookkeeping tables for all f_i in $O(2^\ell)$ time. Similar to phase one, we can leverage the sparsity of f_i and we have the lemma as follows:

Lemma 4.3.5. *Let $\mathcal{N}_y$ be the set of $(z, x) \in \{0,1\}^{2\ell}$ such that $f_i'(z, x, y) = y_{k_i+1} \ldots y_\ell \, f_i(z, x, y_1, \ldots, y_{k_i})$ is non-zero for some $1 \leq i \leq m$. Then for all $y \in \{0,1\}^\ell$, it is $f_i'(g, u, y) = \sum_{(z,x) \in \mathcal{N}_y} \tilde{\beta}(g, z) \tilde{\beta}(u, x) f'(z, x, y)$*

Lemma 4.3.5 is a generalization of Lemma 4.3.2 and we omit the proof.

Lemma 4.3.6. *The bookkeeping table $\boldsymbol{A}_{f_1}, \ldots, \boldsymbol{A}_{f_m}$ can be initialized in time $O(2^\ell)$.*

Proof. As f_i is sparse, $\sum_{y \in \{0,1\}^\ell} |\mathcal{N}_y| = O(2^\ell)$. From Lemma 4.3.2, given the evaluations of $\tilde{\beta}(g, z)$ and $\tilde{\beta}(u, x)$ for all $z, x \in \{0,1\}^\ell$, the prover can iterate all $(z, x) \in \mathcal{N}_x$ for all y to compute $\mathbf{A}_{f_1}, \ldots, \mathbf{A}_{f_m}$. The

Algorithm 12 $\mathbf{A}_{f_1}, \ldots, \mathbf{A}_{f_m} \leftarrow$ Initialize_PhaseTwo$(f_1, \ldots, f_m, g, u)$

Input: Multilinear $f_1, \ldots, f_m$, random $g = g_1, \ldots, g_m$ and $u = u_1, \ldots, u_\ell$;
Output: Bookkeeping tables $\mathbf{A}_{f_1}, \ldots, \mathbf{A}_{f_m}$;

1: $\mathbf{G} \leftarrow$ Precompute(g)
2: $\mathbf{U} \leftarrow$ Precompute(u)
3: $\forall y \in \{0,1\}^\ell$, set $\mathbf{A}_{f_i}[y_1, \ldots, y_{k_i}] = 0$ for all i
4: **for** every (z, x, y) such that $f_i(z, x, y_1, \ldots, y_{k_i})$ is non-zero **do**
5: $\quad \mathbf{A}_{f_i}[y_1, \ldots, y_{k_i}] = \mathbf{A}_{f_i}[y_1, \ldots, y_{k_i}] + \mathbf{G}[z] \cdot \mathbf{U}[x] \cdot f_i(z, x, y_1, \ldots, y_{k_i})$
6: **end for**
7: **return** $\mathbf{A}_{f_1}, \mathbf{A}_{f_2}, \ldots, \mathbf{A}_{f_m}$;

full algorithm is presented in Algorithm 12.

$\mathcal{P}$ runs procedure Precompute(g) and Precompute(u) in $O(2^\ell)$ time as we have shown in the proof of Lemma 4.3.3. Step 4-5 computes $f_i(y_1, \ldots, y_{k_i})$ using Lemma 4.3.5. It takes $O(2^\ell)$ time as $\sum_{y \in \{0,1\}^\ell} |\mathcal{N}_y| = O(2^\ell)$. Therefore, $\mathcal{P}$ runs in $O(2^\ell)$ time for Algorithm 12.

With the bookkeeping tables, the prover runs SumCheckProduct2$(f_1, \mathbf{A}_{f_1}, g_1, \mathbf{A}_{g_1}, \ldots, f_m, \mathbf{A}_{f_m}, g_m, \mathbf{A}_{g_m}, r_1, \ldots, r_\ell)$ in Algorithm 11 and the total complexity for phase two is $O(2^\ell)$.

Combining phase one and phase two, we know that $\mathcal{P}$ runs in $O(|C|)$ time for the sumcheck protocol on Equation 4.8.

Step one with linear prover time. Finally, the sumcheck protocol for Equation 4.7 can be decomposed into several instances that have the form of Equation 4.8. The term

$$\sum_{x,y \in \{0,1\}^{s_{i,i+1}}} (\tilde{mult}_{i+1,i+1}(z, x, y)(\tilde{V}_{i,i+1}(x)\tilde{V}_{i,i+1}(y)) + \ldots$$
$$+ y_{s_{i,d}+1} \ldots y_{s_{i,i+1}} \tilde{mult}_{i+1,d}(z, x, y_1, y_2, \ldots, y_{s_{i,d}})(\tilde{V}_{i,i+1}(x)\tilde{V}_{i,d}(y_1, y_2, \ldots, y_{s_{i,d}}))$$

is exactly the same as Equation 4.8. The term

$$\sum_{x,y \in \{0,1\}^{s_{i,i+1}}} \tilde{add}_{i+1,i+1}(z, x, y)(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,i+1}(y)) + \ldots$$
$$+ y_{s_{i,d}+1} \ldots y_{s_{i,i+1}} \tilde{add}_{i+1,d}(z, x, y_1, \ldots, y_{s_{i,d}})(\tilde{V}_{i,i+1}(x) + \tilde{V}_{i,d}(y_1, \ldots, y_{s_{i,d}}))$$

can be rewritten as the sum of

$$\sum_{x,y \in \{0,1\}^{s_{i,i+1}}} \tilde{add}_{i+1,i+1}(z, x, y)\tilde{V}_{i,i+1}(x) + \ldots$$
$$+ y_{s_{i,d}+1} \ldots y_{s_{i,i+1}} \tilde{add}_{i+1,d}(z, x, y_1, \ldots, y_{s_{i,d}})\tilde{V}_{i,i+1}(x)$$

and

$$\sum_{x,y \in \{0,1\}^{s_{i,i+1}}} \tilde{add}_{i+1,i+1}(z, x, y)\tilde{V}_{i,i+1}(y) + \ldots$$
$$+ y_{s_{i,d}+1} \ldots y_{s_{i,i+1}} \tilde{add}_{i+1,d}(z, x, y_1, \ldots, y_{s_{i,d}})\tilde{V}_{i,d}(y_1, \ldots, y_{s_{i,d}}) .$$

The first sum is the same as Equation 4.8 with $s_i(x) = 1$, and the second sum is the same as Equation 4.8 with $t(x) = 1$. 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, sum up the individual messages and send them to the verifier.

Verifier time and proof size for all sumcheck protocols on Equation 4.7. The verifier time for all sumcheck protocols on Equation 4.7 is the same as Protocol 9. $\mathcal{V}$ still runs in $O(d \log |C|)$ time to verify all sumcheck statements based on Equation 4.7. The proof size is also $O(d \log |C|)$. Note that this excludes the claims of $\tilde{V}_{i,j}$ at random points at the end of the sumcheck protocol in each layer, and we will count them in the next section combining these claims.

4.3.3 Combining Multiple Claims in Linear Time

By executing the sumcheck protocol on Equation 4.7, $\mathcal{P}$ and $\mathcal{V}$ reduce an evaluation of the multiliear extension of a layer to multiple evaluations of multilinear extensions defined by values in the layers above. As we explained in Section 4.3.1, when reaching layer i, $\mathcal{V}$ has received multiple evaluations about this layer and combining these evaluations using a random linear combination would introduce an overhead on the prover. Even worse, with the refined sumcheck on Equation 4.7 in Section 4.3.2, now the verifier has received multiple evaluations of *different* multilinear extensions defined by subsets of gates in layer i used by different layers below. Now even combining these evaluations becomes challenging, let alone reducing the overhead of the prover.

In this section, we propose two different approaches that not only combine these evaluations to a single evaluation of the multilinear extension of the entire layer i, but also incur only a linear prover time in the size of the circuit.

Combining multiple claims by an arithmetic circuit. The key idea of the first approach is that instead of trying to come up with a complicated protocol to do the combination, we simply model the computation as an arithmetic circuit! The circuit takes the values $\mathbf{V}_i$ of the entire layer i as the input. In addition, it also takes the randomness to compute these evaluations of subsets from the verifier. At this point, these randomness are already chosen by the verifier and can merely be viewed as constants known both to $\mathcal{V}$ and $\mathcal{P}$. The circuit then selects multiple subsets from $\mathbf{V}_i$ according to the wiring of the circuit (i.e., gates used by layer $j < i$ from layer i), arrange them in the pre-defined order. The circuit then computes the multilinear extensions of these subsets, and evaluates them on the corresponding points from the input. The structure of the circuit C_i is given in Figure 4.1.

The output of the circuit is exactly the multiple evaluations of the multilinear extensions, which are known to the verifier. The verifier then executes the original GKR protocol for layered arithmetic circuits (Protocol 9) on this circuit, which reduces the output to a single evaluation of the multilinear extension of the input. This can further be expressed as the evaluation of the multilinear extension of $\mathbf{V}_i$, together with the multilinear extension of all the randomnesses used to compute the output. As the latter is known to $\mathcal{V}$, $\mathcal{V}$ can compute it locally. In this way, using the circuit, $\mathcal{P}$ and $\mathcal{V}$ reduce multiple claims about subsets of layer i to one claim about $\tilde{V}_i$.

Another tricky part is that the size of the circuit is not optimal if implemented naively. As shown in Figure 4.1, the circuit expands the randomness to the bookkeeping tables exactly as described in Algorithm 8, which has logarithmic layers $\log S_i$. If the circuit also takes $\mathbf{V}_i$ as input at the same layer as the randomness, $\mathbf{V}_i$ has to be relayed by logarithmic layers and the size of the circuit is $O(S_i \log S_i)$. Instead, we feed $\mathbf{V}_i$ as input to one layer above the bookkeeping tables, as shown on the left side of Figure 4.1. The circuit selects multiple subsets out of it in one layer, and then computes the inner product between a subset and its corresponding bookkeeping table, which gives the evaluation of its multilinear extension. Now the size of the circuit is linear to the total size of all the subsets. The GKR protocol can support inputs from different layers with such a structure, as proposed in [ZGKPP17c]. We give the formal protocol in Protocol 10.

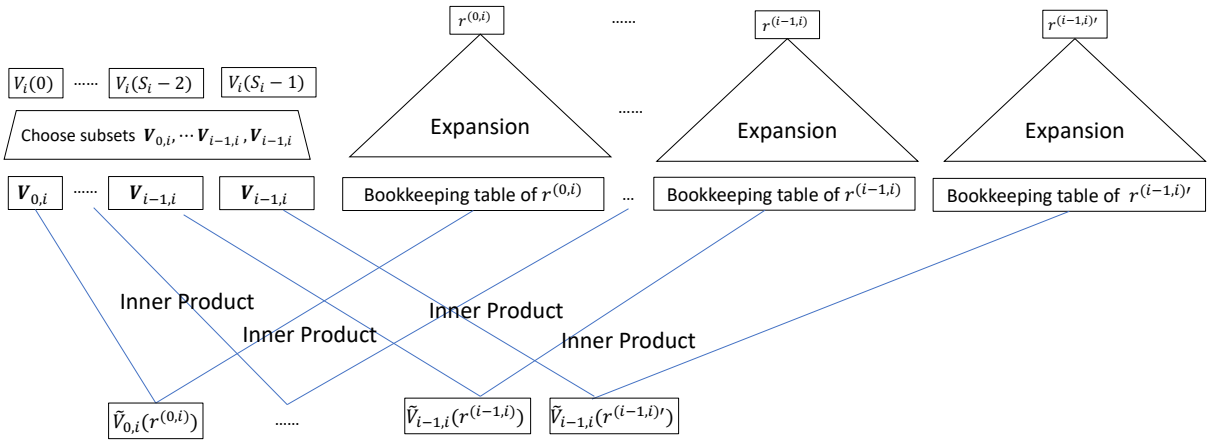


Figure 4.1: Circuit C_i computing $\tilde{V}_{0,i}(r^{(0,i)}), \tilde{V}_{i-1,i}(r^{(i-1,i)}), \tilde{V}_{i-1,i}(r^{(i-1,i)'})$

Protocol 10. Let C_i be the circuit in Figure 4.1 with input **in** consisting of two parts: $\mathbf{V}_i = (V_i(0), \ldots, V_i(S_i-1))$ and $\mathbf{R} = (r^{(0,i)}, \ldots, r^{(i-1,i)}, r^{(i-1,i)'})$, and the output **out** $= (\tilde{V}_{0,i}(r^{(0,i)}), \ldots, \tilde{V}_{i-1,i}(r^{(i-1,i)}), \tilde{V}_{i-1,i}(r^{(i-1,i)'}))$. We use $\mathbf{V} = (\mathbf{V}_{0,i}, \ldots, \mathbf{V}_{i-1,i}, \mathbf{V}_{i-1,i})$ to represent subsets of $\mathbf{V}_i$ used in layer j ($j < i$), and $\mathbf{T}_R = (T_{r^{(0,i)}}, \ldots, T_{r^{(i-1,i)}}, T_{r^{(i-1,i)'}})$ to represent bookkeeping tables after expanding $r^{(0,i)}, \ldots, r^{(i-1,i)}, r^{(i-1,i)'}$.

- $\mathcal{P}$ and $\mathcal{V}$ invoke Protocol 9 on inner products to reduce the claim about **out** to the claim about the layer of $\mathbf{V}$ and $\mathbf{T}_R$: $q = r_1 \cdot \mathbf{V}(r) + (1-r_1) \cdot \mathbf{T}_R(r)$
- $\mathcal{V}$ requires $\mathcal{P}$ to provide values of $\mathbf{V}(r)$ and $\mathbf{T}_R(r)$ to check $q = r_1 \cdot \mathbf{V}(r) + (1-r_1) \cdot \mathbf{T}_R(r)$.
- $\mathcal{P}$ and $\mathcal{V}$ invoke Protocol 9 on the left part and the right part of C_i as shown in Figure 4.1, separately. For the left part, it reduces the claim about $\mathbf{V}(r)$ to the claim about $\mathbf{V}_i(r^{(i)})$ in one layer. For the right part, it reduces the claim about $\mathbf{T}_R(r)$ to the claim about $\mathbf{R}(r^{(i)})$.
- $\mathcal{V}$ asks $\mathcal{P}$ to send $\mathbf{V}_i(r^{(i)})$ and checks the reduction for the left part. $\mathcal{V}$ computes $\mathbf{R}(r^{(i)})$ itself and checks the reduction for the right part. If both checks pass, output `1`; otherwise, output `0`.

Efficiency. In order to analyze the prover time for Protocol 10, we consider the specific structure of C_i, as shown in Figure 4.1. For layer $k < i$, there are $S_{k,i}$ gates from layer i connected to layer k. The number of gates in C_i is at most $8|\mathbf{V}|$, which is $8\sum_{k=0}^{i-1} S_{k,i}$. By Theorem 7.2.5, the prover time for the circuit C_i is $O(\sum_{k=0}^{i-1} S_{k,i})$. So the total prover time for circuits $C_1, \ldots, C_d$ is $8\sum_{i=1}^{d}\sum_{k=0}^{i-1} S_{k,i} = O(|C|)$ as $\sum_{i=1}^{d}\sum_{k=0}^{i-1} S_{k,i}$ equals to the number of all output wires in the circuit, which is at most $2|C|$.

The size of C_i is $O(S_{0,i} + \ldots + S_{i-1,i}) = O(|C|)$, the depth of C_i is $O(\log S_i) = O(\log |C|)$ and the size of input $\mathbf{R}_i$ is at most $s_{0,i} + s_{1,i} + \ldots + s_{i-1,i} + s_{i-1,i} \leq d \log |C|$. Let Q_i be the time to evaluate all $\tilde{add}$ and $\tilde{mult}$ at the corresponding random points in C_i. Therefore, the verifier time for C_i is $O(\log^2 |C| + d \log |C| + Q_i)$ and the proof size is $O(\log^2 |C|)$ by Theorem 7.2.5. In total for all layers, $\mathcal{V}$

runs in $\min\{O(d\log^2|C|+d^2\log|C|+Q),|C|\}$ time and the proof size is $\min\{O(d\log^2|C|),|C|\}$, where $Q=Q_1+Q_2+\ldots+Q_d$.

Combining multiple claims by a sumcheck protocol. Though the prover time of the first method is optimal asymptotically, the overhead in practice is still relatively high. As we will show in Section 7.5, the cost per gate is around $5\times$ slower than that of the original GKR protocol on layered circuits. In addition, it introduces an overhead of $O(\log|C|)$ on the proof size and the verifier time. Therefore, inspired by the design of the circuit in Figure 4.1, we propose the second method to combine multiple claims through a single sumcheck protocol.

The key idea is to define a function to connect the gate in V_i with the same gate in a subset $V_{k,i}$. Formally speaking, we define $C_{k,i}(z,x):\{0,1\}^{s_{k,i}}\times\{0,1\}^{s_i}\to\mathbb{F}$ such that it takes two gate labels, one in the subset $V_{k,i}$ and the other in the entire layer V_i, and $C_{k,i}(z,x)=1$ if the gate z in $V_{k,i}$ is exactly the gate x in V_i. Otherwise $C_{k,i}(z,x)=0$. Note that the function $C_{k,i}$ serves exactly the same purpose as the circuit in Figure 4.1 selecting subsets from $\mathbf{V}_i$.

With the definition of $C_{k,i}$, given $\tilde{V}_{0,i}(r^{(0,i)}),\ldots,\tilde{V}_{i-1,i}(r^{(i-1,i)}),\tilde{V}_{i-1,i}(r^{(i-1,i)'})$, $\mathcal{V}$ can combine them through a random linear combination. In particular, $\mathcal{V}$ chooses $i+1$ random values $\alpha_{0,i},\ldots,\alpha_{i-1,i},\alpha'_{i-1,i}$. Then we have

$$\begin{aligned}
&\textstyle\sum_{k=0}^{i-1}\alpha_{k,i}\tilde{V}_{k,i}(r^{(k,i)})+\alpha'_{i-1,i}\tilde{V}_{i-1,i}(r^{(i-1,i)'})\\
=&\sum_{k=0}^{i-1}\alpha_{k,i}\left(\sum_{x\in\{0,1\}^{s_i}}\tilde{C}_{k,i}(r^{(k,i)},x)\tilde{V}_i(x)\right)+\alpha'_{i-1,i}\sum_{x\in\{0,1\}^{s_i}}\tilde{C}_{i-1,i}(r^{(i-1,i)'},x)\tilde{V}_i(x)\\
=&\textstyle\sum_{x\in\{0,1\}^{s_i}}\tilde{V}_i(x)\left(\sum_{k=0}^{i-1}\alpha_{k,i}\tilde{C}_{k,i}(r^{(k,i)},x)+\alpha'_{i-1,i}\tilde{C}_{i-1,i}(r^{(i-1,i)'},x)\right)\\
=&\textstyle\sum_{x\in\{0,1\}^{s_i}}\tilde{V}_i(x)g_i(x)\,,
\end{aligned}\tag{4.10}$$

where $\tilde{C}_{k,i}$ is the multilinear extension of $C_{k,i}$ and $\tilde{C}_{k,i}(r^{(k,i)},x)=\sum_{z\in\{0,1\}^{s_{k,i}}}\tilde{\beta}(r^{(k,i)},z)C_{k,i}(z,x)$.

We define $g_i(x)=\sum_{k=0}^{i-1}\alpha_{k,i}\tilde{C}_{k,i}(r^{(k,i)},x)+\alpha'_{i-1,i}\tilde{C}_{i-1,i}(r^{(i-1,i)'},x)$. As $g_i(x)$ only depends on the structure of the circuit, $\mathcal{V}$ can compute $g_i(r^{(i)})$ herself given the randomness of $r^{(i)}$. $\mathcal{P}$ and $\mathcal{V}$ can execute a sumcheck protocol on Equation 4.10, which reduces multiple claims of subsets to a single evaluation of $\tilde{V}_i(r^{(i)})$ for the randomness of $r^{(i)}$.

It remains to show that the sumcheck can be executed by the prover in linear time. Recall that given the bookkeeping tables of two multilinear polynomials, the prover can run the sumcheck protocol in linear time using Algorithm 9. In the equation above, the bookkeeping table $\mathbf{A}_{\tilde{V}_i}$ for $\tilde{V}_i$ is already known by the prove as the values of the gates in layer i. We further describe a linear time algorithm to initialize the bookkeeping table $\mathbf{A}_{g_i}$ for $g_i(x)$ in Algorithm 13.

Lemma 4.3.7. *The bookkeeping table $\mathbf{A}_{g_i}$ can be initialized in $O(S_{0,i}+\cdots+S_{i-1,i})$ time.*

Proof. It takes $O(S_{0,i}+\ldots+S_{i-1,i}+S_{i-1,i})$ time to run procedure Precompute($r^{(0,i)}$), …, Precompute($r^{(i-1,i)}$) and Precompute($r^{(i-1,i)'}$). There are at most $S_{0,i}+\ldots+S_{i-1,i}+S_{i-1,i}$ entries such that $C_{k,i}(t,x)=1$. Therefore, the running time of Algorithm 13 is $O(S_{0,i}+\ldots+S_{i-1,i})$.

Efficiency. Next we analyze the efficiency of the second scheme to combine multiple claims to one claim. The prover costs $O(S_1+\ldots+S_d)=O(|C|)$ to compute all bookkeeping tables of $A_{\tilde{V}_i}$. By Lemma 4.3.7,

Algorithm 13 $\mathbf{A}_{g_i} \leftarrow \mathsf{Initialize}(r^{(0,i)}, \ldots, r^{(i-1,i)}, r^{(i-1,i)'})$

Input: $r^{(0,i)}, \ldots, r^{(i-1,i)}, r^{(i-1,i)'}$;
Output: $\mathbf{A}_{g_i}$;

1: $\forall x \in \{0,1\}^{s_i}$, set $\mathbf{A}_{g_i}[x] = 0$
2: **for** $k = 0, \cdots, i-1$ **do**
3: $\quad \mathbf{G} \leftarrow \text{Precompute}(r^{(k,i)})$
4: $\quad$ **for** $t \in \{0,1\}^{s_{k,i}}$ such that $C_{k,i}(t,x) = 1$ **do**
5: $\quad\quad \mathbf{A}_{g_i}[x] = \mathbf{A}_{g_i}[x] + \alpha_{k,i} \cdot \mathbf{G}[t]$
6: $\quad$ **end for**
7: **end for**
8: $\mathbf{G} \leftarrow \text{Precompute}(r^{(i-1,i)'})$
9: **for** $t \in \{0,1\}^{s_{i-1,i}}$ such that $C_{i-1,i}(t,x) = 1$ **do**
10: $\quad \mathbf{A}_{g_i}[x] = \mathbf{A}_{g_i}[x] + \alpha'_{i-1,i} \cdot \mathbf{G}[t]$
11: **end for**
12: **return** $\mathbf{A}_{g_i}$;

the prover runs Algorithm 13 to compute all bookkeeping tables of $\mathcal{A}_{g_i}$ in $O(\sum_{i=1}^{d}\sum_{k=0}^{i-1} S_{k,i}) = O(|C|)$ time. By Lemma 4.3.1, the prover runs the sumcheck protocol on Equation 4.10 for layer 1 to layer d in $O(\sum_{i=1}^{d} S_i) = O(|C|)$ time. So the total prover time of the second scheme is also linear in the circuit size.

By the efficiency of the sumcheck protocol in Protocol 24, it takes $O(\log S_i)$ for the verifier to validate the sumcheck protocol on Equation 4.10 in round i. She also needs to generate $i+1$ random numbers and computes $g_i(r^{(i)})$ in round i. Suppose $\mathcal{V}$ costs T_i to compute $g_i(r^{(i)})$ and $T = T_1 + \ldots + T_d$, the total verifier time is $O(\log S_1 + \ldots + \log S_d) + O(2 + \ldots + d + 1) + T_1 + \ldots + T_d = O(d \log |C| + d^2 + T)$. The total proof size is $O(d \log C + d^2)$. Finally, by a similar analysis to the prover time, the term $O(d^2)$ in the complexity is always bounded by $O(|C|)$. This is because in order for the prover to send a claim about $\tilde{V}_{i,j}$, there has to be a gate in layer i connecting to layer j, thus the number of claims cannot be more than $2|C|$. Therefore, the proof size of our protocol is $\min\{O(d \log C + d^2), O(|C|)\}$ and the verifier time is $\min\{O(d \log |C| + d^2 + T), O(|C|)\}$.

4.3.4 The Full Protocol for General Arithmetic Circuits

Combining the first step and the sumcheck scheme of the second step together, we give the full protocol of the generalized GKR for arbitrary arithmetic circuits in Protocol 11. As the prover time, proof size and the verifier time of the second method to combine multiple points are all better than those of the first method, we state the protocol and the theorem using the second method.

Protocol 11. Let $\mathbb{F}$ be a prime field. Let C: $\mathbb{F}^n \to \mathbb{F}^k$ be a d-depth unlayered arithmetic circuit. $\mathcal{P}$ wants to convince that $C(\mathbf{in}) = \mathbf{out}$ where **in** is the input from $\mathcal{V}$, and **out** is the output. Without loss of

generality, assume n is the power of 2 and both parties can pad them if not.

1. 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 $\tilde{V}_0(g)$.

2. $\mathcal{P}$ and $\mathcal{V}$ run a sumcheck protocol on Equation 4.7 for $i = 0$. At the end of the protocol, $\mathcal{V}$ receives $\tilde{V}_{0,1}(r^{(0,1)'})$, $\tilde{V}_{0,1}(r^{(0,1)})$, $\tilde{V}_{0,2}(r^{(0,2)})$, ..., $\tilde{V}_{0,d}(r^{(0,d)})$. $\mathcal{V}$ computes left side of the above equation by removing the summation symbol and replacing x, y with $r^{(0,1)'}, r^{(0,1)}$. If it does not equal to the last message of the sumcheck, $\mathcal{V}$ outputs 0 and aborts.

3. For $i = 1, ..., d-1, d$:

   a) Given $\tilde{V}_{0,i}(r^{(0,i)}), \ldots, \tilde{V}_{i-1,i}(r^{(i-1,i)'}), V_{i-1,i}(r^{(i-1,i)})$ and $r^{(0,i)}$,$r^{(1,i)}$, ..., $r^{(i-1,i)'}$, $r^{(i-1,i)}$, $\mathcal{V}$ chooses $i+1$ random elements $\alpha_{0,i}, \ldots, \alpha_{i-1,i}, \alpha'_{i-1,i}$ in $\mathbb{F}$ and sends them to $\mathcal{P}$. Then $\mathcal{P}$ and $\mathcal{V}$ run the sumcheck protocol on Equation 4.10. If $\mathcal{V}$ does not abort in the sumcheck protocol, he receives $\tilde{V}_i(r^{(i)})$ for some randomness $r^{(i)} \in \mathbb{F}^{s_i}$ in the last round.

   b) If $i < d$, $\mathcal{P}$ and $\mathcal{V}$ run the sumcheck on Equation 4.7 by replacing g with $r^{(i)}$. At the end of the sumcheck protocol, $\mathcal{P}$ sends $\mathcal{V}$ $\tilde{V}_{i,i+1}(r^{(i,i+1)'})$, $\tilde{V}_{i,i+1}(r^{(i,i+1)})$, ..., $\tilde{V}_{i,d}(r^{(i,d)})$.
   $\mathcal{V}$ computes the left side of the above equation by removing the summation symbol and replacing x, y with $r^{(i,i+1)'}, r^{(i,i+1)}$ and checks it equals to the last message of the sumcheck. If all checks in the sumcheck pass, V uses $\tilde{V}_{i,i+1}(r^{(i,i+1)'})$ and $\tilde{V}_{i,i+1}(r^{(i,i+1)})$ to proceed to the $(i+1)$-th layer. Otherwise, $\mathcal{V}$ outputs 0 and aborts.

4. At the input layer d, $\mathcal{V}$ has one claim of $\tilde{V}_d(r^{(d)})$. $\mathcal{V}$ computes it locally or queries the oracle of evaluations of $\tilde{V}_d$ at $r^{(d)}$ and checks that it is the same as the claim. If yes, output 1; otherwise, output 0.

Theorem 4.3.8. *Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d general arithmetic circuit. Protocol 11 is an interactive proof for the function computed by C with soundness $O(d \log |C|/|\mathbb{F}|)$. The running time of the prover $\mathcal{P}$ is $O(|C|)$. The proof size is $\min\{O(d \log C + d^2), O(|C|)\}$. Let the time to evaluate all $\tilde{add}_{i,j}$ and $\tilde{mult}_{i,j}$ at random points be T', the time to evaluate $g_i(r^{(i)})$ be T_i in Equation 4.10, and $T = T_1 + \ldots + T_d$, the running time of $\mathcal{V}$ is $\min\{O(n + d \log |C| + d^2 + T + T'), O(|C|)\}$. If in addition $d = \mathsf{polylog}(|C|)$, T and T' are in $\mathsf{polylog}(|C|)$ time in Theorem 4.3.8, Protocol 11 is an interactive proof with succinct proof size and verifier time.*

Proof. **Completeness**. The completeness is straightforward by the completeness of the sumcheck protocol. **Soundness**. For the soundness, for any PPT adversary $\mathcal{A}$, we use $\tilde{V}'$ to represent the correct messages corresponding to $\tilde{V}$ in Protocol 11 with input **in** and the correct execution for circuit C. Suppose $C(\mathbf{in}) \neq \mathbf{out}$, there must exist a layer i such that $\tilde{V}_j(r^{(j)}) = \tilde{V}'_j(r^{(j)})$ and $\tilde{V}_{k,j}(r^{(k,j)}) = \tilde{V}'_{k,j}(r^{(k,j)})$ for $j > i$ and all $k < j$ but $\tilde{V}_i(r^{(i)}) \neq \tilde{V}'_i(r^{(i)})$ or $\tilde{V}_{k,i}(r^{(k,i)}) \neq \tilde{V}'_{k,i}(r^{(k,i)})$ for some $k < i$, which event is defined as E_i. This event can be divided into three cases:

- Case 1: The random elements are chosen in a way such that $\sum_{k=0}^{i-1} \alpha_{k,i} \tilde{V}'_{k,i}(r^{(k,i)}) + \alpha'_{i-1,i} \tilde{V}'_{i-1,i}(r^{(i-1,i)'}) = \sum_{k=0}^{i-1} \alpha_{k,i} \tilde{V}_{k,i}(r^{(k,i)}) + \alpha'_{i-1,i} \tilde{V}_{i-1,i}(r^{(i-1,i)'})$. This happens with probability at most $\frac{1}{|\mathbb{F}|}$.

- Case 2: The above case does not happen, but at the end of sumcheck protocol induced by Equation 4.10, the final round random evaluation (e.g. $\tilde{V}_i(r^{(i)})$) is consistent with the single evaluation of $\tilde{V}_i'(r^{(i)})$. By the soundness of sumcheck protocol this happens with probability at most $\frac{2\lceil \log S_i \rceil}{|\mathbb{F}|}$.

- Case 3: We have $\tilde{V}_i(r^{(i)}) \neq \tilde{V}_i'(r^{(i)})$, but the verifier accepts after the sumcheck protocol for $\tilde{V}_i(r^{(i)})$. Since we assume that $\tilde{V}_j(r^{(j)}) = \tilde{V}_j'(r^{(j)})$ and $\tilde{V}_{k,j}(r^{(k,j)}) = \tilde{V}_{k,j}'(r^{(k,j)})$ for $j > i$ and all $k < j$, this happens with probability at most $\frac{2\lceil \log S_{i+1} \rceil}{|\mathbb{F}|}$ by the soundness of sumcheck protocol.

Thus the overall probability that event E_i happens is at most $\frac{2(\lceil \log S_i \rceil + \lceil \log S_{i+1} \rceil)+1}{|\mathbb{F}|} = O(\frac{\log |C|}{|\mathbb{F}|})$.

Eventually, by the union bound, we have the following statement:

$$\begin{aligned}
\Pr[C(\mathbf{in}) \neq \mathbf{out} \wedge \mathcal{V} \text{ outputs } 1] &\leq \Pr[\exists i, E_i] \\
&\leq \Pr[E_0] + \Pr[E_1] + \ldots + \Pr[E_{d-1}] \\
&\leq O(\frac{\log |C|}{|\mathbb{F}|}) + O(\frac{\log |C|}{|\mathbb{F}|}) + \ldots + O(\frac{\log |C|}{|\mathbb{F}|}) \\
&\leq O(\frac{d \log |C|}{|\mathbb{F}|})
\end{aligned}$$

The efficiency follows the efficiency analysis of Section 4.3.2 and Section 4.3.3.

4.4 Zero Knowledge Arguments from Generalized GKR

In this section, we build a new zero knowledge argument protocol for general arithmetic circuits based on Protocol 11. The construction follows the same ideas proposed in [CFS17; XZZPS19b]. In particular, as proposed in [ZGKPP17c], we combine the GKR protocol with a (zero knowledge) polynomial commitment scheme on the witness to build an argument scheme. In order to achieve zero knowledge, we apply the zero knowledge sumcheck protocol [CFS17; XZZPS19b] on Equation 4.7 and 4.10 to eliminate the leakage during the sumcheck. We then use the low degree extensions instead of multilinear extensions of V_i and $V_{i,j}$ so that their evaluations sent from the prover to the verifier do not leak information about the values in the circuit. The only difference is that for the values V_i in each layer i of the circuit, the verifier receives multiple claims, one for each of the subsets $V_{i,j}$, instead of two claims about V_i in the original GKR protocol. Thus, we use the low degree extensions of both V_i and $V_{i,j}$ with a different random masking polynomial for each. In this way, these claims leak no information about the values.

For completeness, we present the formal definitions and protocols in the following.

4.4.1 Definitions

We introduce definitions of zero knowledge arguments and zero knowledge polynomial commitments before presenting the formal protocols.

Zero knowledge arguments. 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 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 parameters pp. Formally, consider the definition below, where we assume R is known to $\mathcal{P}$ and $\mathcal{V}$.

Definition 4.4.1. *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* $(x, w) \in R$,

$$\langle \mathcal{P}(\mathsf{pp}, w), \mathcal{V}(\mathsf{pp}) \rangle(x) = 1$$

- ***Knowledge Soundness***. *For any PPT prover $\mathcal{P}^*$, there exists a PPT extractor $\mathcal{E}$ such that given the access to the entire executing process and the randomness of $\mathcal{P}^*$, $\mathcal{E}$ can extract a witness w such that* $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$, $\pi^* \leftarrow \mathcal{P}^*(x, \mathsf{pp})$ *and* $w \leftarrow \mathcal{E}^{\mathcal{P}^*}(\mathsf{pp}, x, \pi^*)$, *the following probability is* $\mathsf{negl}(\lambda)$:

$$\Pr[(x; w) \notin \mathcal{R} \wedge \mathcal{V}(x, \pi^*, \mathsf{pp}) = 1]$$

- ***Zero knowledge***. *There exists a PPT simulator $\mathcal{S}$ such that for any PPT algorithm $\mathcal{V}^*$, auxiliary input $z \in \{0,1\}^*$, $(x; w) \in \mathcal{R}$,* pp *output by* $\mathcal{G}(1^\lambda)$, *it holds that*

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

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) are* $\mathsf{poly}(\lambda, |x|, \log |w|)$.

In the definition of zero knowledge, $\mathcal{S}^{\mathcal{V}^*}$ denotes that the simulator $\mathcal{S}$ is given the randomness of $\mathcal{V}^*$ sampled from polynomial-size space. This definition is commonly used in existing transparent zero knowledge proof schemes [**ben2019aurora**; XZZPS19b; AHIV17; BBBPWM; WTSTW18; ZXZS].

Zero knowledge polynomial commitment. 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. We use $\mathcal{W}_{\ell,d}$ to denote the collection of all monomials in $\mathcal{F}$ and $N = |\mathcal{W}_{\ell,D}| = (D+1)^\ell$. A zero knowledge verifiable polynomial commitment (zkPC) for $f \in \mathcal{F}$ and $t \in \mathbb{F}^\ell$ consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$,
- $\mathsf{com} \leftarrow \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp})$,
- $((y, \pi); \{0,1\}) \leftarrow \langle \mathsf{zkVPD.Open}(f, r_f), \mathsf{zkVPD.Verify}(\mathsf{com}) \rangle(t, \mathsf{pp})$

Definition 4.4.2. *A zkPC scheme satisfies the following properties:*

- ***Completeness.*** *For any polynomial $f \in \mathcal{F}$ and value $t \in \mathbb{F}^\ell$,* $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$, $\mathsf{com} \leftarrow \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp})$, *it holds that*

$$\Pr\left[\langle \mathsf{zkVPD.Open}(f, r_f), \mathsf{zkVPD.Verify}(\mathsf{com}) \rangle(t, \mathsf{pp}) = 1\right] = 1$$

- ***Knowledge Soundness.*** *For any PPT adversary $\mathcal{A}$, $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$, there exists a PPT extractor $\mathcal{E}$. Given any tuple (pp, com^*) and the executing process of $\mathcal{A}$, $\mathcal{E}$ can extract a function $f^* \in \mathcal{F}$ and the randomness r_{f^*} such that $(f^*, r_{f^*}) \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}, \mathsf{com}^*)$ and $\mathsf{com}^* \leftarrow \mathsf{zkVPD.Commit}(f^*, r_{f^*}, \mathsf{pp})$. The following probability is negligible of λ:*

$$\Pr\left[((y^*, \pi^*); 1) \leftarrow \langle \mathcal{A}(), \mathsf{zkVPD.Verify}(\mathsf{com}^*)\rangle(t, \mathsf{pp}) \wedge (f^*, r_{f^*}) \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}, \mathsf{com}^*) \wedge f^*(t) \neq y^*\right]$$

- ***Zero Knowledge.*** *For security parameter λ, polynomial $f \in \mathcal{F}$, $\mathsf{pp} \leftarrow \mathsf{zkVPD.KeyGen}(1^\lambda)$, PPT algorithm $\mathcal{A}$, and simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$, consider the following two experiments:*

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

1. $\mathsf{com} \leftarrow \mathsf{zkVPD.Commit}(f, r_f, \mathsf{pp})$
2. $t \leftarrow \mathcal{A}(\mathsf{com}, \mathsf{pp})$
3. $(y, \pi) \leftarrow \langle \mathsf{zkVPD.Open}(f, r_f), \mathcal{A}\rangle(t, \mathsf{pp})$
4. $b \leftarrow \mathcal{A}(\mathsf{com}, y, \pi, \mathsf{pp})$
5. *Output b*

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

1. $\mathsf{com} \leftarrow \mathcal{S}_1(1^\lambda, \mathsf{pp})$
2. $t \leftarrow \mathcal{A}(\mathsf{com}, \mathsf{pp})$
3. $(y, \pi) \leftarrow \langle \mathcal{S}_2, \mathcal{A}\rangle(t_i, \mathsf{pp})$*, given oracle access to* $y = f(t)$.
4. $b \leftarrow \mathcal{A}(\mathsf{com}, y, \pi, \mathsf{pp})$
5. *Output b*

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

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

4.4.2 Zero Knowledge Sumcheck

To build a zero knowledge argument for arbitrary arithmetic circuit using Protocol 11, we follow the same blueprint of [XZZPS19b] using zkPC, zero knowledge sumcheck and low degree extensions. In the following, we present the zero knowledge version of step 3(b) and step 3(a) in Protocol 11, followed by the whole zero knowledge argument.

In step 3(b) of the full protocol, $\mathcal{P}$ and $\mathcal{V}$ execute a sumcheck protocol on Equation 4.7, 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 prevent the leakage, we take the zero knowledge sumcheck proposed by Xie et al. in [XZZPS19b]. 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 such that $g(x_1, \ldots, x_\ell) = a_0 + g_1(x_1) + g_2(x_2) + \ldots + g_\ell(x_\ell)$, where $g_i(x_i) = a_{i,1}x_i + a_{i,2}x_i^2 + \ldots + a_{i,\tau}x_i^\tau$ is a random univariate polynomial of degree τ (τ is the variable degree of f). Note here that the size of g is only $O(\tau\ell)$, while the size of f is exponential in ℓ. $\mathcal{P}$ commits to the polynomial g using $\mathsf{zkVPD.Commit}$, 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 $\rho \in \mathbb{F}$, 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 $g(r_1, \ldots, r_\ell)$ using $\mathsf{zkVPD.Open}$, and the verifier computes $f(r_1, \ldots, r_\ell)$ 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 polynomial commitment is zero knowledge, the protocol is zero knowledge. Intuitively, this is because the information of f transmitted in the sumcheck protocol is exactly masked by the randomness of g. We present the protocol in Protocol 12 and we have the following theorem:

Protocol 12. We assume the existence of a zkPC protocol defined in Section 4.4.1. For simplicity, we omit the randomness r_f and public parameters pp, vp without any ambiguity. To prove the claim $H = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$:

1. $\mathcal{P}$ selects a polynomial $g(x_1, \ldots, x_\ell) = a_0 + g_1(x_1) + g_2(x_2) + \ldots + g_\ell(x_\ell)$, where $g_i(x_i) = a_{i,1}x_i + a_{i,2}x_i^2 + \ldots + a_{i,\tau}x_i^\tau$ and all $a_{i,j}$s are uniformly random. $\mathcal{P}$ sends $H = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$, $G = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} g(x_1, x_2, \ldots, x_\ell)$ and $\mathsf{com}_g = \mathsf{zkVPD.Commit}(g, r_g, \mathsf{pp})$ 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 $h_\ell(r_\ell) = f(r_1, r_2, \ldots, r_\ell) + \rho g(r_1, r_2, \ldots, r_\ell)$. $\mathcal{P}$ opens the commitment of g at $r = (r_1, \ldots, r_\ell)$ and $\mathcal{V}$ verifies by using $\mathsf{zkVPD.Open}$ and $\mathsf{zkVPD.Verify}$. If the verification fails, $\mathcal{V}$ aborts.
5. $\mathcal{V}$ computes $h_\ell(r_\ell) - \rho g(r_1, \ldots, r_\ell)$ and compares it with the oracle access of $f(r_1, \ldots, r_\ell)$.

Theorem 4.4.3 ([XZZPS19a]). *Protocol 12 is complete and sound for the relationship of* $H = \sum_{x_1,x_2,\ldots,x_\ell \in \{0,1\}} f(x_1, x_2, \ldots, x_\ell)$. *In addition, 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 Protocol 12. The efficiency of prover time, verifier time and proof size in Protocol 12 retain the same as in Protocol 24.*

We apply the zero knowledge sumcheck directly on the sumcheck equation (Equation 4.7 and 4.10) of our new GKR protocol. It eliminates all the leakage during the sumcheck protocol.

4.4.3 Zero Knowledge GKR

Even with the zero knowledge sumcheck, the protocol still leaks information about values in the circuit. In particular, at the end of the zero knowledge sumcheck, $\mathcal{V}$ still needs an oracle access to $f(r_1, \ldots, r_\ell)$. When executed on Equation 4.7, the verifier evaluates all $\tilde{add}$ and $\tilde{mult}$ at the random point, and queries the prover for the evaluations of $\tilde{V}_{0,i}, \cdots, \tilde{V}_{i-1,i}$. These evaluations reveal information about values in the circuit.

To prevent this leakage, we use the same idea in [XZZPS19b] to replace them with their low-degree extensions $\dot{V}_{0,i}, \cdots, \dot{V}_{i-1,i}$. Let

$$\dot{V}_{j,i}(x) \stackrel{def}{=} \tilde{V}_{j,i}(x) + Z_{j,i}(x) \cdot \sum_{w \in \{0,1\}} R_{j,i}(x_1, w), \tag{4.11}$$

where $Z_{j,i}(x) = \prod_{k=1}^{s_{j,i}} x_k(1-x_k)$ is the vanishing polynomial, i.e., $Z_{j,i}(x) = 0$ for all $x \in \{0,1\}^{s_{j,i}}$, and $R_{j,i}$ is the mask polynomial with only two variables generated by $\mathcal{P}$.

Additionally, in the last round of the sumcheck on Equation 4.10, $\mathcal{V}$ asks for $\tilde{V}(r^{(i)})$, which leaks information about V_i. With exactly the same idea as above, we replace it with its low-degree extension $\dot{V}_i$ such that

$$\dot{V}_i(x_1, \ldots, x_{s_i}) \stackrel{def}{=} \tilde{V}_i(x_1, \ldots, x_{s_i}) + Z_i(x_1, \ldots, x_{s_i}) \sum_{w \in \{0,1\}} R_i(x_1, w), \tag{4.12}$$

where $Z_i(x) = \prod_{i=1}^{s_i} x_i(1-x_i)$ is still the vanishing polynomial, and R_i is still a mask multilinear polynomial with only two variables. As $R_{0,i}, \cdots, R_{i-1,i}$ and R_i are randomly selected by $\mathcal{P}$, revealing several evaluations of them does not leak information about $V_{0,i}, \cdots, V_{i-1,i}$ and V_i thus the values in the circuit. The zero knowledge polynomial commitment scheme is used to commit to these masking polynomials and later open them at random points. With these changes, Equation 4.7 becomes

$$\begin{aligned}
\dot{V}_i(g) = & \sum_{\substack{x,y \in \{0,1\}^{s_{i,i+1}} \\ w \in \{0,1\}}} [\tilde{\beta}(w, \vec{1})(\tilde{add}_{i+1,i+1}(g,x,y)(\dot{V}'_{i,i+1}(x) + \dot{V}_{i,i+1}(y_1, \ldots, y_{s_{i,i+1}})) \\
& + y_{s_{i,i+2}+1} \cdot \ldots \cdot y_{s_{i,i+1}} \tilde{add}_{i+1,i+2}(g,x,y_1, \ldots, y_{s_{i,i+2}})(\dot{V}'_{i,i+1}(x) + \dot{V}_{i,i+2}(y_1, \ldots, y_{s_{i,i+2}})) \\
& + \ldots + y_{s_{i,d}+1} \cdot \ldots \cdot y_{s_{i,i+1}} \tilde{add}_{i+1,d}(g,x,y_1, \ldots, y_{s_{i,d}})(\dot{V}'_{i,i+1}(x) + \dot{V}_{i,d}(y_1, \ldots, y_{s_{i,d}})) \\
& + \tilde{mult}_{i+1,i+1}(g,x,y)(\dot{V}'_{i,i+1}(x)\dot{V}_{i,i+1}(y_1, \ldots, y_{s_{i,i+1}})) \\
& + y_{s_{i,i+2}+1} \cdot \ldots \cdot y_{s_{i,i+1}} \tilde{mult}_{i+1,i+2}(g,x,y_1, \ldots, y_{s_{i,i+2}})(\dot{V}'_{i,i+1}(x)\dot{V}_{i,i+2}(y_1, \ldots, y_{s_{i,i+2}})) \\
& + \ldots + y_{s_{i,d}+1} \cdot \ldots \cdot y_{s_{i,i+1}} \tilde{mult}_{i+1,d}(g,x,y_1, \ldots, y_{s_{i,d}})(\dot{V}'_{i,i+1}(x)\dot{V}_{i,d}(y_1, \ldots, y_{s_{i,d}})) \\
& + \tilde{\beta}((x,y), \vec{1}) Z_i(g) R_i(g_1, w)]
\end{aligned} \tag{4.13}$$

The equation holds because $\dot{V}_{j,i}$ agrees with $\tilde{V}_{j,i}$ on the Boolean hypercube $\{0,1\}^{s_{i,j}}$, as $Z_{j,i}(z) = 0$ for binary inputs.

Now $\mathcal{P}$ and $\mathcal{V}$ instead execute the zero knowledge sumcheck protocol on Equation 4.13. At the end of the protocol, $\mathcal{V}$ receives $\dot{V}_{i,i+1}(r^{(i,i+1)'}), \dot{V}_{i,i+1}(r^{(i,i+1)}), \ldots, \dot{V}_{i,d}(r^{(i,d)})$ for random points $r^{(i,i+1)'}, r^{(i,i+1)}, \ldots, r^{(i,d)}$ chosen by $\mathcal{V}$. They no longer leak information about $V_{i,i+1}, \ldots, V_{i,d}$. $\mathcal{V}$ then evaluates $\tilde{mult}_{i,j}$ and $\tilde{add}_{i,j}$ on the randomness as before, computes $Z_i(g), \tilde{\beta}(c,1), \tilde{\beta}((r^{(i,i+1)'}, r^{(i,i+1)}), \vec{1})$ where $c \in \mathbb{F}$ is a random point chosen by $\mathcal{V}$ for the variable w. $\mathcal{V}$ also opens $R_i(g_1, w)$ at point c with $\mathcal{P}$ using zkPC, and checks that together with the points received from $\mathcal{P}$, they are consistent with the last message of the sumcheck, i.e., the oracle access to the evaluation of the polynomial in the zero knowledge sumcheck. $\mathcal{V}$ then uses these values to proceed to the second step of combining multiple evaluations, i.e., step 3(a) in Protocol 11. We have the following theorem.

Theorem 4.4.4. *For every verifier $\mathcal{V}^*$, there exists a simulator $\mathcal{S}$ such that given oracle access to $\dot{V}_i(g)$ and $\dot{V}_{i,i+1}(r^{(i,i+1)'}), \dot{V}_{i,i+1}(r^{(i,i+1)}), \ldots, \dot{V}_{i,d}(r^{(i,d)})$, $\mathcal{S}$ is able to simulate the partial view of $\mathcal{V}^*$ in the zero knowledge sumcheck protocol on Equation 4.13.*

Proof sketch. The completeness and the soundness inherits from the zero knowledge sumcheck protocol and zkVPD. For zero knowledge, we combine the simulator $\mathcal{S}_1$ in zkVPD and the simulator $\mathcal{S}_2$ in the zero knowledge sumcheck protocol to construct the simulator $\mathcal{S}$. Therefore, $\mathcal{V}$ only learns $\dot{V}_{i,i+1}(r^{(i,i+1)'})$, $\dot{V}_{i,i+1}(r^{(i,i+1)}), \ldots, \dot{V}_{i,d}(r^{(i,d)})$ at the end of the protocol, which leaks no information about $\tilde{V}_{i,i+1}, \cdots, \tilde{V}_{i,d}$ because of mask polynomials of $R_{i,i+1}, \cdots, R_{i,d}$.

Efficiency. Compared with the plain sumcheck protocol in step 3(a) of Protocol 11, the prover costs extra $O(1)$ time to compute zkVPD.Commit and zkVPD.Open for $R_i(g_1, w)$ and $O(\log |C|)$ compute zkVPD.Commit and zkVPD.Open for the mask polynomial in Protocol 12. Therefore, the total prover time is still $O(|C|)$. The verifier time is $\min\{O(d \log |C| + d^2), O(|C|)\}$ while the proof size is also $\min\{O(d \log |C| + d^2), O(|C|)\}$.

Combine multiple evaluations in zero knowledge. With low degree extensions of $\dot{V}_{0,i}, \ldots, \dot{V}_{i-1,i}$ and $\dot{V}_i$, we modify Equation 4.10 to

$$
\begin{aligned}
&\sum_{k=0}^{i-1} \alpha_{k,i} \dot{V}_{k,i}(r^{(k,i)}) + \alpha'_{i-1,i} \dot{V}_{i-1,i}(r^{(i-1,i)'}) \\
=&\sum_{k=0}^{i-1} \alpha_{k,i} \left(\sum_{x \in \{0,1\}^{s_i}} \tilde{C}_{k,i}(r^{(k,i)}, x) \dot{V}_i(x) \right) + \alpha'_{i-1,i} \sum_{x \in \{0,1\}^{s_i}} \tilde{C}_{i-1,i}(r^{(i-1,i)'}, x) \dot{V}_i(x) + \\
&\sum_{k=0}^{i-1} \alpha_{k,i} Z_{k,i}(r^{(k,i)}) \sum_{w \in \{0,1\}} R_{k,i}(r_1^{(k,i)}, w) + \alpha'_{i-1,i} Z_{i-1,i}(r^{(i-1,i)'}) \sum_{w \in \{0,1\}} R_{i-1,i}(r_1^{(i-1,i)'}, w) \qquad (4.14) \\
=&\sum_{x \in \{0,1\}^{s_i}, w \in \{0,1\}} \Bigg[\tilde{\beta}(w,1) \dot{V}_i(x) \left(\sum_{k=0}^{i-1} \alpha_{k,i} \tilde{C}_{k,i}(r^{(k,i)}, x) + \alpha'_{i-1,i} \tilde{C}_{i-1,i}(r^{(i-1,i)'}, x) \right) + \\
&\tilde{\beta}(\vec{x}, \vec{1}) \left(\sum_{k=0}^{i-1} \alpha_{k,i} Z_{k,i}(r^{(k,i)}) R_{j,i}(r_1^{(k,i)}, w) + \alpha'_{i-1,i} Z_{i-1,i}(r^{(i-1,i)'}) R_{i-1,i}(r_1^{(i-1,i)'}, w) \right) \Bigg],
\end{aligned}
$$

The equation holds because $\dot{V}_i$ agrees with $\tilde{V}_i$ on the Boolean hypercube $\{0,1\}^{s_i}$, as $Z_i(z) = 0$ for binary inputs. To execute the second step, the prover commits to mask polynomials of $R_{0,i}, \ldots, R_{i-1,i}$ using zkPC. $\mathcal{P}$ and $\mathcal{V}$ then run the zero knowledge sumcheck protocol on Equation 4.14. At the end of the protocol, the verifier receives evaluations of $R_{0,i}(r_1^{(0,i)}, c), \ldots, R_{i-1,i}(r_1^{(i-1,i)}, c)$ on a random point c chosen by $\mathcal{V}$ for the variable w. He opens $R_{0,i}(r_1^{(0,i)}, c), \ldots, R_{i-1,i}(r_1^{(i-1,i)}, c)$ using the zkPC. Then $\mathcal{V}$ evaluates $g_i(r^{(i)})$ as before, computes all $Z_{k,i}(r^{(k,i)})$, $Z_{i-1,i}(r^{(i-1,i)'})$, $\beta(c,1)$, $\beta(r^{(i)}, \vec{1})$, shaves them off to obtain the evaluation of $\dot{V}_i(r^{(i)})$. We have the following theorem.

Theorem 4.4.5. *For every verifier $\mathcal{V}^*$, there exists a simulator $\mathcal{S}$ such that given oracle access to $\dot{V}_i(r^{(i)})$ and $\dot{V}_{0,i}(r^{(0,i)}), \ldots, \dot{V}_{i-1,i}(r^{(i-1,i)}), \dot{V}_{i-1,i}(r^{(i-1,i)'})$, $\mathcal{S}$ is able to simulate the partial view of $\mathcal{V}^*$ in the zero knowledge sumcheck protocol on Equation 4.14.*

Proof sketch. The completeness and the soundness inherits from the zero knowledge sumcheck protocol and zkVPD. For zero knowledge, we combine the simulator $\mathcal{S}_1$ in zkVPD and the simulator $\mathcal{S}_2$ in the zero

knowledge sumcheck protocol to construct the simulator $\mathcal{S}$. Therefore, $\mathcal{V}$ only learns $\dot{V}_i(r^{(i)})$ at the end of the protocol, which leaks no information about $\tilde{V}_i$ because of the mask polynomail of R_i.

Efficiency. Compared with the plain sumcheck protocol in the second step, the prover costs extra $O(i)$ time to compute zkVPD.Commit and zkVPD.Open for i constant size polynomials of $R_{0,i}, \ldots, R_{i-1,i}$ and $O(\log |C|)$ compute zkVPD.Commit and zkVPD.Open for the mask polynomial in Protocol 12. Therefore, the total prover time is $O(|C| + 1 + \ldots + d) = O(|C|)$. The verifier time is $\min\{O(d \log |C| + d^2), O(|C|)\}$ while the proof size is also $\min\{O(d \log |C| + d^2), O(|C|)\}$.

As the second approach to combine multiple claims in Section 4.3.3 is better on all aspects, we focus on building zero knowledge arguments using the second approach. The first approach can also be lifted to a zero knowledge argument with similar ideas by applying a zero knowledge GKR protocol on circuit C_i in Figure 4.1.

4.4.4 Putting Everything Together

Combining the zero knowledge variants of step 3(a) and 3(b) in Protocol 11 with the zkVPD scheme, we get a zero knowledge argument protocol for general arithmetic circuits.

Theorem 4.4.6. *For an input size n and a finite field $\mathbb{F}$, let $\mathcal{C}_{\mathbb{F}}$ represent the set of general arithmetic circuits of depth d on $\mathbb{F}$, then there exists 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 6.2.2. Moreover, using the polynomial commitment scheme (Definition 4.4.2) in [ZXZS], for every $(C, x; w) \in \mathcal{R}$, the running time of $\mathcal{P}$ is $O(|C| + n \log n)$. The running time of $\mathcal{V}$ is $\min\{O(|x| + \log^2 n + d \log |C| + d^2 + T''), O(|C|)\}$, where T'' is the total time to compute all functions of $\tilde{add}$ and $\tilde{mult}$ and all functions of $g_i(r^{(i)})$ in the second step. The total proof size is $\min\{O(d \log |C| + d^2), O(|C|)\}$. In case d is polylog$(|C|)$ *and T'' is also* polylog$(|C|)$*, the protocol is a succinct argument with succinct verifier time.*

Proof Sketch of Theorem 4.4.6. The correctness and the soundness follow from those of the three building blocks, by Theorem 4.4.4, 4.4.5 and Definition 4.4.2.

To prove zero knowledge, consider a simulator $\mathcal{S}$ that calls the simulator $\mathcal{S}_1$ of zero knowledge sumcheck given in Theorem 4.4.4 for step 1, the simulator $\mathcal{S}_2$ of combining multiply claims to one claim with zero knowledge given in Theorem 4.4.5 for step 2 and the simulator $\mathcal{S}_3$ of zkVPD in Definition 4.4.2 for committing and opening of all hiding polynomials as subroutines. Then $\mathcal{S}$ can simulate the partial view of every verifier $\mathcal{V}^*$ for any general arithmetic circuit C only given oracle access to x.

The complexity of our zero knowledge argument scheme follows from the efficency of Protocol 12 and the extra complexity of applying zkVPD.Commit to the input layer demonstrated in [ZXZS].

4.5 Implementations and Evaluations

We fully implement our new interactive proof protocols for general arithmetic circuits and use them to build a zero knowledge argument system for general arithmetic circuits. We name our new system Virgo++. The implementation is in C++. There are around 1900 lines of code for Protocol 11 and 1600 lines for building the arithmetic circuit to combine multiple evaluations into one (Protocol 10). We implement two variants

		Prover time (s)			Verifier time (s)			Proof size (KB)		
		2^9	2^{11}	2^{13}	2^9	2^{11}	2^{13}	2^9	2^{11}	2^{13}
$d = 50$	GKR	0.118	0.465	1.908	0.052	0.206	0.838	83	95	107
	Our Scheme 1	0.043	0.154	0.576	0.013	0.042	0.151	280	397	535
	Our Scheme 2	0.012	0.049	0.197	0.003	0.011	0.044	93	106	120
$d = 75$	GKR	0.244	0.973	3.954	0.100	0.404	1.608	129	147	166
	Our Scheme 1	0.069	0.243	0.910	0.021	0.066	0.237	416	593	803
	Our Scheme 2	0.019	0.075	0.304	0.004	0.017	0.066	168	188	208

Table 4.1: Comparison of our scheme 1, our scheme 2 and the original GKR on random circuits.

of combining multiple claims to one claim in step 3(b) of Protocol 11 as described in Section 4.3.3. One is building the arithmetic circuit to make the reduction as shown in Figure 4.1 and the other is running the sumcheck protocol on Equation 4.10. Our protocols work on any finite field, and we choose the extension field $\mathbb{F}_{p^2}$ for the Mersenne prime $p = 2^{61} - 1$. This is the same as in [ZXZS], and we choose it so that our interactive proof protocols can be compatible with the polynomial commitments in [ZXZS] to build zero knowledge arguments. The choice of the finite field does not affect our comparison to the original GKR protocol in the next Section. Our protocols provide 100+ bits of security. We plan to make our implementation open-source.

Hardware. We ran all of the experiments on an AWS EC2 c5a.2xlarge instance with an AMD EPYC 7R32 CPU with 3.512Ghz, 8 cores and 16GB of RAM. Our current implementation is not parallelized and we only utilize a single CPU core in the experiments. We report the average running time of 10 executions.

4.5.1 Comparing to the GKR Protocol for Layered Circuits

In this section, we compare the performance of our new protocols with the original GKR protocol. For a fair comparison, we re-implement the GKR protocol for layered arithmetic circuits with the same field and libraries in C++. We generate random general circuits with depth $d = 50$ and $d = 75$. We vary the number of gates in each layer from 2^9 to 2^{13}. Our schemes can easily go beyond 2^{13}, but the original GKR protocol on the corresponding layered circuits runs out of memory on our machine. We randomly sample the type of each gate, input value and the wiring patterns. We execute our new protocols directly on these general circuits. We refer the one using the arithmetic circuit to combine multiple claims to one claim in Protocol 10 as scheme 1 and the one using the sumcheck protocol on Equation 4.10 to combine multiple claims for step 3(b) in Protocol 11 as scheme 2. We then transform the general circuits to layered circuits by relaying necessary values layer by layer, and execute the original GKR protocol on the layered circuits. We report the prover time, verifier time and proof size in Table 4.1.

First, when we transform the general circuits to layered circuits, the size of the circuit increases by $13\times$ for $d = 50$ and by $19\times$ for $d = 75$. This roughly agrees with the blowup of $O(d|C|)$ and justifies the high overhead of transforming general circuits to layered circuits. As shown in Table 4.1, when the depth is 50, the prover time of our scheme 1 is faster by 2-4$\times$ than the original GKR protocol, while our scheme 2 is faster by 9-10$\times$. When the depth is 75, the speedup increases to 3-5$\times$ for scheme 1 and 12-13$\times$ for scheme 2. Finally,

the prover time in all schemes grows linearly with the size of the circuit, and is very efficient in practice. The cost per gate in scheme 2 is only 0.49μs.

To further justify the improvement, the prover of the original GKR protocol for layered circuits takes around 21 field multiplications per gate. In the implementation of our new protocols, the cost per gate of the prover is around 120 field multiplications for scheme 1 and around 27 field multiplications for scheme 2. The average cost per gate of our scheme 2 is only 1.29$\times$ of the original GKR protocol. In other words, as long as the layered circuit has 22% or more relay gates, it is faster to remove those relay gates and run our new protocol of scheme 2 on the corresponding general circuit. The speedup in our experiments above matches the analysis here.

Our protocols introduce an overhead on the proof size compared to the original GKR protocol. In particular, the proof size of our first scheme is 3-5$\times$ larger than the GKR protocol, matching the $\log |C|$ overhead in the complexity of the proof size. However, the proof size of our second scheme is very close to the GKR protocol. It is only 1.1-1.3$\times$ larger, showing that this variant reduces the proof size significantly upon scheme 1. In fact, this overhead is introduced by the second sumcheck protocol to combine multiple points. The term d^2 in the complexity has minimal impact on the total proof size. In all cases, the proof size is succinct. The largest proof size is still less than 1MB and the proof size is always much smaller than the size of the circuit.

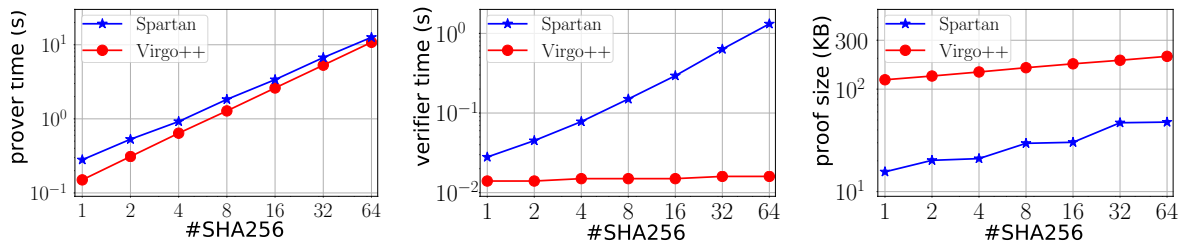
As the circuits are generated randomly, the verifier time in all schemes are linear in the circuit size. Therefore, the comparisons on the verifier time of the three protocols are similar to the comparisons on the prover time. As shown in Table 4.1, our scheme 1 is faster by 4-6$\times$ than the original GKR protocol on circuits with $d = 50$, and 5-7$\times$ faster for $d = 75$. Our scheme 2 is 17-19$\times$ faster on circuits with $d = 50$, and 23-25$\times$ faster for $d = 75$. Therefore, we observe in the experiments that our scheme 2 improves the performance of scheme 1 on all the aspects on random circuits, proving our statement in Section 4.3.3. Compared to the original GKR protocol, our scheme 2 is much faster on the prover time and the verifier time, and incurs only a small overhead on the proof size.

4.5.2 Evaluations of Our Zero Knowledge Argument

In this section, we present the performance of our new zero knowledge argument for general arithmetic circuits, as described in Section 4.4. We use the zero knowledge polynomial commitment scheme in [ZXZS] to lift our new interactive proofs to zero knowledge arguments. We import the open-source code of zero knowledge polynomial commitment scheme in [Vira]. We also compare our zero knowledge proof system with Spartan [Spa].

We do experiments on the benchmark of computing the hash functions of SHA-256. For our protocol, we modify the circuit generation file of [Hyr; Liba] to obtain the general arithmetic circuit for SHA-256. In fact, the code first generates the general circuit of SHA-256 and then pads it to the layered circuit, and our new protocol makes the circuit design even simpler. The circuit contains other types of gates such as subtraction, bit decomposition and reconstruction. We modify our protocols to support all these types of gates. Each SHA-256 circuit has 99,949 gates in total (around 2^{17}), with the input size of 7,226 (around 2^{13}). In the experiments, we vary the number of SHA-256 from 1 to 64.

Figure 4.2 shows the performance of our system (red line with circle markers). As shown in the figure, Virgo++ achieves good efficiency in practice. It only takes 0.15s to generate the proof of one SHA-256 circuit, and 0.014s to verify. In the largest instance of 64 hashes, our system takes 10.8s to generate the proof, the verifier time is 0.016s and the proof size is 209KB. The verifier time only grows slightly with the number



(a) $\mathcal{P}$ time (b) $\mathcal{V}$ time (c) Proof size

Figure 4.2: Comparison of Virgo++ and Spartan.

of hashes, as the verifier time of our new GKR protocol is only linear in the size of a single hash in the data-parallel circuit, and logarithmic in the size of the entire circuit.

Comparing to Spartan. We then compare the performance of our system with Spartan [Set20a], which also combines the sumcheck protocol and the polynomial commitments to construct zero knowledge arguments on R1CS. As described in [Set20a, Section 5], the sumcheck protocol is executed on a equation defined by the extended witness z and the matrices A, B, C in an R1CS instance. The size of the extended witness roughly maps to the number of multiplication gates in a circuit, and the number of nonzero elements in the matrices roughly maps to the number of addition gates. As Spartan is also using the linear time sumcheck protocol proposed in [XZZPS19b], the prover time of the sumcheck protocol is expected to be similar to the sumcheck protocol in our zero knowledge argument ($O(n)$ in Spartan, where n is the number of nonzeros in the matrices [Set20a], and $O(|C|)$ in Virgo++). The major improvement of Virgo++ comes from the polynomial commitment part. In our scheme, the polynomial commitment is only on the witness of the circuit, while in Spartan, the polynomial commitment is on the extended witness, which is always larger than the size of the real witness of the circuit. The improvement comes at the cost of larger proof size. In our scheme, we reduce the correctness of the output layer by layer to the real witness and the proof size is linear in the depth, while in Spartan, the sumcheck is executed on one "layer" to check the correctness of the extended witness.

We demonstrate the comparison in our experiments. We download the open-source code of Spartan from [Spa]. We use the highly-optimized R1CS for SHA-256 generated by jsnark [Jsnb]. Each SHA-256 has 25,656 (around 2^{15}) constraints and 25,546 (around 2^{15}) witnesses. The number of nonzero elements is 87,689 in A, 54,968 in B and 78,232 in C. Note that the size of the extended witness is $3.5\times$ larger than the witness of our general circuit for the same function of SHA-256, while the number of nonzeros in the matrices is roughly the same as the size of the circuit, matching our analysis above. As the open-source code of Spartan only works on randomly generated R1CS instances, we generate random R1CS instances with exactly the same number of constraints, witnesses and nonzero elements as SHA-256.

Figure 4.2 shows the performance of Spartan (blue line with star markers). As shown in the figure, the prover time of Virgo++ is 1.2–1.8$\times$ faster than Spartan. The verifier of Spartan grows linearly with the number of hashes and is significantly slower than Virgo++. We believe its verifier time can also be made sublinear for data-parallel circuits, but it is not considered in [Set20a] and its implementation. In contrast, the proof size of Virgo++ is 4.1–7.9$\times$ larger than Spartan. Other than the reason explained above, this is also partly because we are using the polynomial commitment in [ZXZS] based on interactive oracle proofs (IOP). It is known that IOP-based schemes have larger proof size compared to discrete-log based schemes including the one used in Spartan, but are plausibly post-quantum secure.

The evaluations of Spartan are in the NIZK mode. There is a SNARK mode of Spartan that has sublinear verifier time in the holographic model, but the prover time is $9\times$ slower. Finally, as described in Section 5.1.1, our new GKR protocol can also be used for delegation of computations. Spartan does not work in this setting as the size of the extended witness is always asymptotically the same as the size of the computation and the verifier does not save anything by delegating the computation using Spartan. In a recent manuscript [SL20], the proof size of Spartan is improved from square-root to logarithmic in the size of the R1CS instance, but the prover time is $3.8\times$ slower. We do not include the comparison as its implementation is not available.

Chapter 5

Zero Knowledge Proofs for Decision Tree Predictions and Accuracy

Machine learning has become increasingly prominent and is widely used in various applications in practice. Despite its great success, the integrity of machine learning predictions and accuracy is a rising concern. The reproducibility of machine learning models that are claimed to achieve high accuracy remains challenging, and the correctness and consistency of machine learning predictions in real products lack any security guarantees.

In this paper, we initiate the study of zero knowledge machine learning and propose protocols for zero knowledge decision tree predictions and accuracy tests. The protocols allow the owner of a decision tree model to convince others that the model computes a prediction on a data sample, or achieves a certain accuracy on a public dataset, without leaking any information about the model itself. We develop approaches to efficiently turn decision tree predictions and accuracy into statements of zero knowledge proofs. We implement our protocols and demonstrate their efficiency in practice. For a decision tree model with 23 levels and 1,029 nodes, it only takes 250 seconds to generate a zero knowledge proof proving that the model achieves high accuracy on a dataset of 5,000 samples and 54 attributes, and the proof size is around 287 kilobytes.

This work was previously published in [ZFZS20].

5.1 Introduction

Machine learning has seen a great development over the past years, leading to important progress in various research areas such as computer vision, data mining, and natural language processing. Despite the great success, there are many security concerns of machine learning techniques, one of which is the *integrity* of machine learning models and their predictions. Newly developed machine learning models are claimed to achieve high accuracy, yet it is challenging to reproduce the results and validate these claims in many cases. In addition, even if a high quality model exists, it may not be used consistently in real-world products. For example, an online service for image classification claiming to use a particular model may simply return a random answer, and there is no guarantee on the integrity of the result for the clients. The authors in [Bot] report an extreme case where a company claims to use machine learning techniques to build robots delivering food automatically, yet the robots are actually operated by remote workers.

These scenarios urge the need for a solution to ensure that the owner indeed has a valid machine learning model, and it is used to compute the predictions correctly or it achieves high accuracy on public datasets. A naïve approach is to release the machine learning models publicly. However, it completely sacrifices the privacy of the machine learning models. Machine learning models are becoming important intellectual properties of the companies and cannot be shared publicly for validation. Releasing the machine learning models as black-box testing software does not address the issues as well. For example, in the machine learning service scenarios above, the customers can just take the black-box software away without paying. Even worse, recent research shows that one can infer sensitive information or reconstruct the machine learning models with only black-box accesses [FJR15].

In this paper, we propose to address the problem of machine learning integrity using the cryptographic primitive of *zero knowledge proof* (ZKP). A zero knowledge proof allows a *prover* to produce a short proof π that can convince any *verifier* that the result of a public function f on the public input x and secret input w of the prover is $y = f(x, w)$. w is usually referred as the witness or auxiliary input. Zero knowledge proofs guarantee that the verifier rejects with overwhelming probability if the prover cheats on computing the result, while the proof reveals no extra information about the secret w beyond the result.

During the last decade, there has been great progress on generic ZKP schemes that are nearly practical. They allow proving arbitrary computations modeled as arithmetic circuits. In principle, we can apply these general-purpose ZKP schemes to address the problem of machine learning integrity. The prover proves that she knows a secret machine learning model that computes the prediction of an input or achieves the claimed accuracy on a public dataset, without leaking any additional information about the machine learning model. The proof is *succinct*, meaning that it is much smaller than the machine learning model and the prediction function. However, it is particularly challenging to construct efficient ZKP for machine learning predictions and accuracy tests because of the high overhead on the proof generation time. Because of these challenges, ZKP schemes for machine learning computations have not been widely studied in the literature.

Our contributions.. In this paper, we initiate the study of zero knowledge machine learning predictions and accuracy, and propose several efficient schemes for zero knowledge decision trees. We also extend our techniques with minimal changes to support variants of decision trees, including regression, multivariate decision trees and random forests. Decision trees and random forests play important roles in various applications in practice, because of their good explainability and interpretability: the predictions can be explained by meaningful rules and conditions. They are widely used for product recommendations, fraud detection and automated trading in financial applications. Our concrete contributions are:

- **Zero knowledge decision tree predictions.** First, we propose an efficient protocol for zero knowledge

decision tree predictions. After a setup phase to commit to a decision tree in linear time to the size of the decision tree, the prover time is only proportional to the length of the prediction path h, and the number of attributes d of the data. We apply several critical techniques in the literature of ZKP for computations in the random access memory (RAM) model in non-black-box ways, and translate the decision tree prediction to a small circuit of size $O(d + h)$.

- **Zero knowledge decision tree accuracy.** Second, we generalize our protocol to prove the accuracy of a decision tree in zero knowledge. We develop two important optimizations to bound the number of hashes in our ZKP backend to be exactly the number of nodes N in the decision tree. It is independent of the number of data samples to test, and is much less than 2^h if the decision tree is unbalanced.
- **Implementation and evaluations.** Finally, we fully implement our protocols and evaluate their performance on several real-world datasets. Notably, for a large decision tree with 23 levels and 1,029 nodes, it only takes 250s to generate a proof for its accuracy on a testing dataset with 5,000 samples and 54 attributes. The proof size is 287KB and the verification time is 15.6s.

5.1.1 Related Work

Zero knowledge proofs were introduced by Goldwasser et al. in [GMR89] and generic constructions based on probabilistically checkable proofs were proposed in the seminal work of Kilian [Kil92] and Micali [Mic00]. In recent years there has been significant progress in efficient ZKP protocols and systems. Categorized by their underlying techniques, there are succinct non-interactive argument of knowledge (SNARK) schemes [PHGR13; BSCGTV; BFRSBW; BSCTV14; Cos+; WSRBW15; FFGKOP16; Gro16a], discrete-log-based schemes [Gro09; BG12; BCCGP16; BBBPWM], hash-based schemes [BCGGHJ17], interactive oracle proofs (IOP) [AHIV17; BSCRSVW19; BSBHR19; ZXZS] and interactive-proof-based schemes [ZGKPP17c; ZGKPP17a; WTSTW18; XZZPS19a]. Their security relies on different assumptions and settings, and they provide trade-offs between prover time and proof size. In our construction, we use the ZKP scheme proposed in [BSCRSVW19], named Aurora, as our backend because of its fast prover time and good scalability. The proof size is relatively large compared to other schemes. Please refer to [WTSTW18; XZZPS19a; ZXZS] for more details on the performance and comparisons of different ZKP schemes.

Most ZKP schemes model the computations as arithmetic circuits, while decision tree predictions are naturally in the RAM model with comparisons and conditional branching. Several papers [BSCGTV; BSCTV; WSRBW15; ZGKPP18; BCGJM18] proposed ZKP schemes for RAM programs. We use some of their techniques in our constructions, without going through the heavy machinery of RAM-to-circuit reductions.

Zero knowledge proofs for machine learning applications have not been studied extensively before. In [GGG17], Ghodsi et al. proposed a system named SafetyNet to delegate neural network predictions to a cloud server. It assumes that the verifier has the neural network and guarantees the soundness of the predictions. The scheme does not support witness from the prover, and there is no notion of zero knowledge. In [Zha+19], Zhao et al. proposed to use SNARK to validate neural network predictions. The prover commits to the values of all intermediate layers, and the verifier validates one layer randomly with a SNARK proof. The scheme does not provide negligible soundness. It also justifies the challenge of the overhead on the prover time as we mentioned in the introduction, as it is too expensive to apply SNARK to the whole machine learning model.

Finally, there is a rich literature on privacy-preserving decision tree predictions and training using secure multiparty computations (MPC), oblivious RAM (ORAM) and fully homomorphic encryptions (FHE) [VC05;

TKK19; BPTG15]. We note here that both the focus and the techniques of these schemes are quite different from zero knowledge proofs. These schemes primarily guarantee the privacy of the data during training and predictions. They do not provide integrity of the results and the succinctness of the communication. The settings and applications are also different from our zero knowledge decision trees.

5.2 Preliminaries

We use $\mathsf{negl}(\cdot) : \mathbb{N} \to \mathbb{N}$ to denote the negligible function, where for each positive polynomial $f(\cdot)$, $\mathsf{negl}(k) < \frac{1}{f(k)}$ for sufficiently large integer k. Let λ denote the security parameter. Let $[m]$ denote the set of $\{1, 2, \cdots, m\}$. "PPT" standards for probabilistic polynomial time. We use bold letters $\mathbf{x}, \mathbf{y}, \mathbf{z}$ to represent vectors, and $\mathbf{x}[i]$ denote the i-th element in vector $\mathbf{x}$.

5.2.1 Zero-knowledge Arguments

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 w such that $(x; w) \in \mathcal{R}$ 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 parameters pp. Formally, consider the definition below, where we assume $\mathcal{R}$ is known to $\mathcal{P}$ and $\mathcal{V}$.

Definition 5.2.1. *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.*

- ***Completeness***. *For every* pp *output by* $\mathcal{G}(1^\lambda)$, $(x; w) \in \mathcal{R}$ *and* $\pi \leftarrow \mathcal{P}(x, w, \mathsf{pp})$,

$$\Pr[\mathcal{V}(x, \pi, \mathsf{pp}) = 1] = 1$$

- ***Knowledge Soundness***. *For any PPT prover $\mathcal{P}^*$, there exists a PPT extractor $\mathcal{E}$ such that given the access to the entire executing process and the randomness of $\mathcal{P}^*$, $\mathcal{E}$ can extract a witness w such that* $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$, $\pi^* \leftarrow \mathcal{P}^*(x, \mathsf{pp})$ *and* $w \leftarrow \mathcal{E}^{\mathcal{P}^*}(\mathsf{pp}, x, \pi^*)$, *the following probability is* $\mathsf{negl}(\lambda)$*:*

$$\Pr[(x; w) \notin \mathcal{R} \wedge \mathcal{V}(x, \pi^*, \mathsf{pp}) = 1]$$

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

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

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* $\mathsf{poly}(\lambda, |x|, \log |w|)$.

In the definition of zero knowledge, $\mathsf{View}(\mathcal{V}^*(\mathsf{pp}, x))$ denotes the veiw the verifier sees during the execution of the interactive process with $\mathcal{P}$ while $\mathcal{S}^{\mathcal{V}^*}(x)$ denotes the view generated by $\mathcal{S}$ given input x and transcript of $\mathcal{V}^*$, and $\approx$ denotes two distributions perfect indistinguishable. This definition is commonly used in existing transparent zero knowledge proof schemes [AHIV17; BBBPWM; WTSTW18; BSCRSVW19].

5.2.2 Our Zero-knowledge Argument Backend

With the recent progress on efficient zero knowledge proofs(ZKP) protocols, there are several general purpose ZKP systems with different trade-offs on the prover time, the proof size and the verification time. In our construction and implementation, we aim to optimize for fast prover time and to scale to large decision trees. Therefore, after careful comparisons among all existing ZKP systems, we choose the scheme named Aurora proposed in [BSCRSVW19] as the ZKP backend in our zero knowledge decision tree construction. We state its properties in the following theorem. Note that our construction is also compatible with other ZKP systems.

Theorem 5.2.2. *[BSCRSVW19]. Let λ be the security parameter, for a finite field $\mathbb{F}$ and a family of layered arithmetic circuit $\mathcal{C}_{\mathbb{F}}$ over $\mathbb{F}$, there exists a zero knowledge argument of knowledge for the relation*

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

as defined in Definition 6.2.2, where x is the public input and w is the auxiliary input(private to the prover) to the circuit C.

Moreover, for every $(C, x; w) \in \mathcal{R}$, the running time of $\mathcal{P}$ is $O(|C| \log |C|)$ field operations. The running time of $\mathcal{V}$ is $O(|C|)$ and the proof size is $O(\log^2 |C|)$, where $|C|$ is the number of arithmetic gates in the circuit C.

Aurora has no trusted setup. Its security is proven in the random oracle model, and is plausibly post-quantum secure. It can be made non-interactive using Fiat-Shamir [FS] in the random oracle model.

In addition, in order to build our zero knowledge decision tree scheme, we require an additional algorithm of the general purpose ZKP protocol to commit the witness. This is formalized as "Commit-and-Prove" in [CFQ], and is naturally supported by Aurora and most of ZKP systems. We denote the algorithm as $\mathsf{comm}_w \leftarrow \mathsf{Commit}(w, \mathsf{pp})$. It is executed after $\mathcal{G}$ and before $\mathcal{P}$, and $\mathcal{V}$ additionally takes comm_w as an input. It satisfies the extractability of commitment. Similar to the extractability in Definition 6.2.2, there exists a PPT extractor $\mathcal{E}$, given any tuple $(\mathsf{pp}, x, \mathsf{comm}^*_w)$ and the executing process of $\mathcal{P}^*$, it could always extract a witness w^* such that $\mathsf{comm}^*_w \leftarrow \mathsf{Commit}(w^*, \mathsf{pp})$ except for the negligible probability in λ. Formally speaking, $\mathsf{comm}^*_w = \mathsf{Commit}(\mathcal{E}^{\mathcal{P}^*}(\mathsf{pp}, x, \mathsf{comm}^*_w), \mathsf{pp})$.

5.3 Zero Knowledge Decision Tree

In this section, we present our main construction of zero knowledge decision tree predictions. We first introduce the background on decision trees. Then we formally define the notion of zero knowledge decision tree predictions, and present our protocol with the security analysis.

5.3.1 Decision Tree

Decision tree is one of the most commonly used machine learning algorithms. It performs particularly well for classification problems, and has good explainability and interpretability. Therefore, it is widely deployed in applications such as product recommendations, fraud detection and automated trading.

For simplicity, we focus on binary decision trees for classification problems in our presentation, but our techniques can be generalized naturally to decision trees with higher degrees, decision trees for regression problems, multivariate decision trees and random forests with small changes. We present these variants in

Section 5.5.2. In a decision tree $\mathbf{T}$, each intermediate node contains an attribute, each branch represents a decision and each leaf node denotes an outcome (categorical or continues value). More formally, each internal node v has an associated attribute index $v.\mathsf{att}$ from the set $[d]$ of d attributes, a threshold $v.\mathsf{thr}$ and two children $v.\mathsf{left}$ and $v.\mathsf{right}$. Each leaf node u stores the classification result $u.\mathsf{class}$. Each data sample is represented as a size-d vector $\mathbf{a}$ of values corresponding to each attribute. The algorithm of decision tree prediction is shown in Algorithm 14. It starts from the root of $\mathbf{T}$. For each node of v in $\mathbf{T}$, it compares $\mathbf{a}[v.\mathsf{att}]$ with $v.\mathsf{thr}$, and moves to $v.\mathsf{left}$ if $\mathbf{a}[v.\mathsf{att}] < \mathsf{v.thr}$, and $v.\mathsf{right}$ otherwise. Eventually, the algorithm reaches a leaf node u and the result of the prediction is $u.\mathsf{class}$.

To train a decision tree, given a training dataset, the decision tree is obtained by splitting the set into subsets from the root to the children. The splitting is based on some splitting rules by maximizing the certain objective function such as the information gain and the splitting process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion will halt when the subset of a node has the same classification, or when splitting can not increase the value of the objective function. In our scheme, we only consider proving predictions given the pretrained decision tree, and the training process is out of the scope.

5.3.2 Zero Knowledge Decision Tree Prediction

Motivated by the applications mentioned in the introduction, in our model, the prover owns a pre-trained decision tree. The prover commits to the decision tree first, and then later the verifier queries for the prediction of a data sample. The prover generates a proof together with the result to convince the verifier its validity.

Formally speaking, let $\mathbb{F}$ be a finite field, $\mathbf{T}$ be a binary decision tree of height h and N nodes ($N \leq 2^h - 1$). Suppose the test dataset is $\mathcal{D} \subseteq \mathbb{F}^d$, in which each data point has d features. So for each data $\mathbf{a}$ in $\mathcal{D}$, $\mathbf{a} \in \mathbb{F}^d$. Let $[M]$ be the set of all target classifications. We treat the decision tree algorithm as a mapping $\mathbf{T} : \mathbb{F}^d \to [M]$. For a data point $\mathbf{a} \in \mathcal{D}$, $\mathbf{T}(\mathbf{a}) \in [M]$ is the prediction for the classification of $\mathbf{a}$ using the decision tree algorithm on $\mathbf{T}$. A zero-knowledge decision tree scheme (zkDT) consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{zkDT.KeyGen}(1^\lambda)$: given the security parameter, generate the public parameter pp.

Algorithm 14 Decision Tree Prediction

Input: Decision tree $\mathbf{T}$, data sample $\mathbf{a}$
Output: classification $y_{\mathbf{a}}$

```
v := T.root
while v is not a leaf node do
    if a[v.att] < v.thr then
        v := v.left
    else
        v := v.right
    end if
end while
return v.class
```

- $\mathsf{comm}_{\mathbf{T}} \leftarrow \mathsf{zkDT.Commit}(\mathbf{T}, \mathsf{pp}, r)$: commit the decision tree $\mathbf{T}$ with a random point r generated by the prover.
- $(y_{\mathbf{a}}, \pi) \leftarrow \mathsf{zkDT.Prove}(\mathbf{T}, \mathbf{a}, \mathsf{pp})$: given a data $\mathbf{a}$, run the decision tree algorithm to get $y_{\mathbf{a}} = \mathbf{T}(\mathbf{a})$ and the corresponding proof π.
- $\{0,1\} \leftarrow \mathsf{zkDT.Verify}(\mathsf{comm}_{\mathbf{T}}, h, \mathbf{a}, y_{\mathbf{a}}, \pi, \mathsf{pp})$: validate the prediction of $\mathbf{a}$ given $y_{\mathbf{a}}$ and π obtained from the prover.

In our scheme, we assume the height of the decision tree (or an upper bound) is known to both parties. $\mathsf{comm}_{\mathbf{T}}$ is the commitment of the decision tree. $y_{\mathbf{a}}$ denotes the class of $\mathbf{a}$ returned by the decision tree. And π denotes the proof generated by the prover. $\{0,1\}$ represents reject or accept output by the verifier after seeing the classification and the proof.

Definition 5.3.1. *We say that a scheme is a zero knowledge decision tree if the following holds:*

- ***Completeness.*** *For any decision tree $\mathbf{T}$ and a data point $\boldsymbol{a} \in \mathbb{F}^d$, $\mathsf{pp} \leftarrow \mathsf{zkDT.KeyGen}(1^\lambda)$, $\mathsf{comm}_{\mathbf{T}} \leftarrow \mathsf{zkDT.Commit}(\mathbf{T}, \mathsf{pp}, r)$, $(y_{\boldsymbol{a}}, \pi) \leftarrow \mathsf{zkDT.Prove}(\mathbf{T}, \boldsymbol{a}, \mathsf{pp})$, it holds that*

$$\Pr\left[\mathsf{zkDT.Verify}(\mathsf{comm}_{\mathbf{T}}, h, \boldsymbol{a}, y_{\boldsymbol{a}}, \pi, \mathsf{pp}) = 1\right] = 1$$

- ***Soundness.*** *For any PPT adversary $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr\begin{bmatrix} \mathsf{pp} \leftarrow \mathsf{zkDT.KeyGen}(1^\lambda) \\ (\mathbf{T}^*, \mathsf{comm}_{\mathbf{T}^*}, \boldsymbol{a}, y^*_{\boldsymbol{a}}, \pi^*) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, r) \\ \mathsf{comm}_{\mathbf{T}^*} = \mathsf{zkDT.Commit}(\mathbf{T}^*, \mathsf{pp}, r) \\ \mathsf{zkDT.Verify}(\mathsf{comm}_{\mathbf{T}^*}, h, \boldsymbol{a}, y^*_{\boldsymbol{a}}, \pi^*, \mathsf{pp}) = 1 \\ \mathbf{T}(\boldsymbol{a}) \neq y^*_{\boldsymbol{a}} \end{bmatrix}$$

- ***Zero Knowledge.*** *For security parameter λ, $\mathsf{pp} \leftarrow \mathsf{zkDT.KeyGen}(1^\lambda)$, for a decision tree $\mathbf{T}$ with h levels, PPT algorithm $\mathcal{A}$, and simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$, consider the following two experiments:*

$\mathsf{Real}_{\mathcal{A},\mathbf{T}}(\mathsf{pp})$:

1. $\mathsf{com}_{\mathbf{T}} \leftarrow \mathsf{zkDT.Commit}\ (\mathbf{T}, \mathsf{pp}, r)$
2. $\boldsymbol{a} \leftarrow \mathcal{A}(h, \mathsf{com}_{\mathbf{T}}, \mathsf{pp})$
3. $(y_{\boldsymbol{a}}, \pi) \leftarrow \mathsf{zkDT.Prove}(\mathbf{T}, \boldsymbol{a}, \mathsf{pp})$
4. $b \leftarrow \mathcal{A}(\mathsf{com}_{\mathbf{T}}, h, \boldsymbol{a}, y_{\boldsymbol{a}}, \pi, \mathsf{pp})$
5. *Output b*

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

1. $\mathsf{com} \leftarrow \mathcal{S}_1(1^\lambda, \mathsf{pp}, h)$
2. $\boldsymbol{a} \leftarrow \mathcal{A}(h, \mathsf{com}, \mathsf{pp})$
3. $(y_{\boldsymbol{a}}, \pi) \leftarrow \mathcal{S}_2^{\mathcal{A}}(\mathsf{com}, h, \boldsymbol{a}, \mathsf{pp})$, *given oracle access to* $y_{\boldsymbol{a}} = \mathbf{T}(\boldsymbol{a})$.
4. $b \leftarrow \mathcal{A}(\mathsf{com}, h, \boldsymbol{a}, y_{\boldsymbol{a}}, \pi, \mathsf{pp})$
5. *Output b*

For any PPT algorithm $\mathcal{A}$ and all decision tree $\mathbf{T}$ with the height of h, there exists simulator $\mathcal{S}$ such that

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

Intuition of the specific construction of zkDT: .given the algorithm of decision tree and the definition of the zero knowledge decision tree protocol(zkDT), we will focus on the specific construction of our zkDT scheme in the subsequent subsections. The general idea of the construction is as follows. In the beginning, the prover sends the committment of a decision tree $\mathbf{T}$, $\mathsf{comm}_{\mathbf{T}}$, to the verifier. After receiving $\mathbf{a}$ from the verifier, the prover computes $y_{\mathbf{a}}$ and the corresponding witness w for proving $y_{\mathbf{a}} = \mathbf{T}(\mathbf{a})$, then sends $y_{\mathbf{a}}$ to the verifier. We treat it as some relationship $\mathcal{R} = ((y_{\mathbf{a}}, \mathbf{a}, \mathsf{comm}_{\mathbf{T}}); w)$ in Definition 6.2.2. Then the verifier and the prover invoke the backend zero-knowledge proofs protocol in subsection 5.2.2 to verify the relationship $\mathcal{R}$ without leaking any information of $\mathbf{T}$ except for $y_{\mathbf{a}}$.

5.3.3 Authenticated Decision Tree

We start with the scheme to commit to a decision tree. A naive approach to do so is to simply compute the cryptographic hash of the whole decision tree. However, later when the prover wants to prove the prediction of a data sample using the committed decision tree, the whole decision tree has to be included as the witness of the zero knowledge proof, while only the path from the root to the leaf node of the output is relevant for the prediction. This would introduce a high overhead to recompute the hash of the whole tree.

One could also build a Merkle hash tree [Mer] on top of all the nodes in the decision tree (with a predefined order). Then later in zkDT.Prove, the prover associates each node in the prediction path with a valid Merkle tree proof. The overhead of this approach will be $O(h \log N)$ hashes in the zero knowledge proof backend to validate the all nodes in the prediction path. Instead, we propose to leverage the rich literature of authenticated data structures [Tam03]. We propose to build authenticated decision trees (ADT) directly on the data structure of decision trees so that proving a prediction path only relies on the information stored in the nodes.

Construction of ADT: . Figure 5.1 illustrates our construction of ADT. The construction is very similar to Merkle hash tree, with the difference that the data stored in an intermediate node of the decision tree is also included in the computation of its hash, together with the hashes of the two children. In particular, each node v contains the attribute $v.\mathsf{att}$, the threshold $v.\mathsf{thr}$, the pointers to the children $v.\mathsf{left}$ and $v.\mathsf{right}$, and the hashes of its children. In the real implementation, we use different identities in $[N]$ to represent the nodes in $\mathbf{T}$. Therefore, $v.\mathsf{left}$ and $v.\mathsf{right}$ are identities of the left child and right child of v respectively.

The verification algorithm is also similar to that of the Merkle hash tree. To validate the prediction for a data sample, the proof includes the prediction path from the root to the leaf node that outputs the prediction result. In addition, the proof also includes the hashes of the siblings of the nodes along the prediction path.

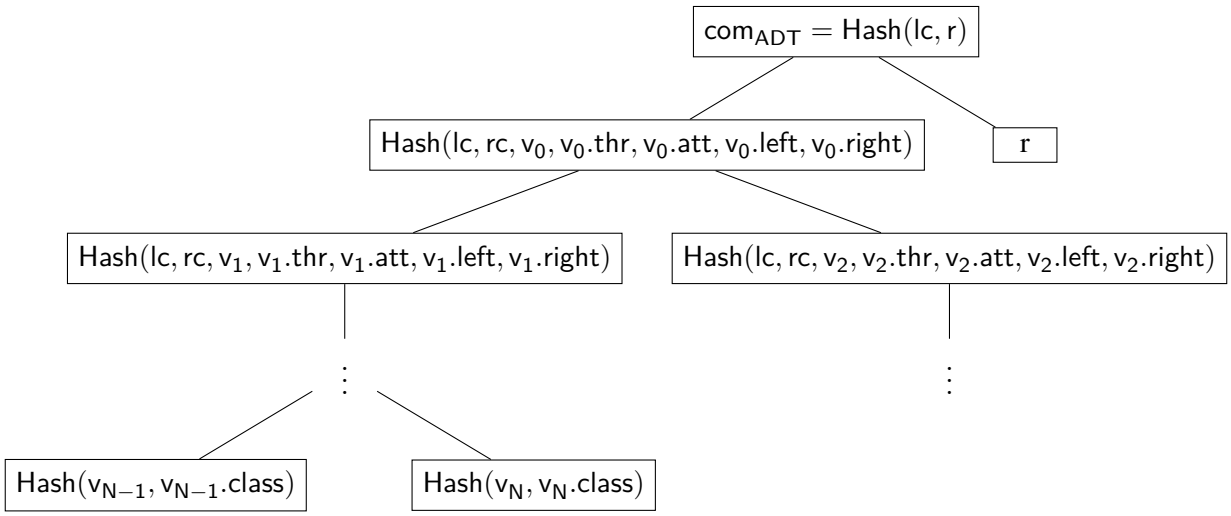


Figure 5.1: Committing algorithm of ADT scheme, lc and rc represent the left child value and the right child value respectively.

With the proof, the verifier can recompute the root hash and compare it with the commitment. In this way, to prove the validity of a prediction, the verification only computes $O(h)$ hashes. Note that in our construction of zkDT, the verification of ADT is never executed by the verifier directly. As we will show in the next section, the prover further proves that there exists such a valid prediction path through a general purpose zero knowledge proof. Because of this design, the verification of ADT does not have to be zero knowledge.

The algorithms of our ADT are in the following. Note that in order to prove the zero knowledge property of the scheme later, the commitment has to be randomized and we add a random point r to the root of the decision tree and use the hash of the root concatenated with r as the final commitment, as shown in Figure 5.1. Moreover, for the purpose of our application, the ADT does not have to support dynamic insertions and deletions, which simplifies the construction significantly.

- $\mathsf{pp} \leftarrow \mathsf{ADT.KeyGen}(1^\lambda)$: Sample a collision resistant hash function from the family of hash functions.
- $\mathsf{comm}_{\mathsf{ADT}} \leftarrow \mathsf{ADT.Commit}(\mathbf{T}, \mathsf{pp}, r)$: compute hashes from leaf nodes to the root of $\mathbf{T}$ with the random point r as shown in Figure 5.1.
- $\pi_{\mathsf{ADT}} \leftarrow \mathsf{ADT.Prove}(\mathbf{T}, \mathsf{Path}, \mathsf{pp})$: given a path in $\mathbf{T}$, π_{ADT} contains all siblings of the nodes along the path Path and the randomness r in Figure 5.1.
- $\{0, 1\} \leftarrow \mathsf{ADT.Verify}(\mathsf{comm}_{\mathsf{ADT}}, \mathsf{Path}, \pi_{\mathsf{ADT}}, \mathsf{pp})$: given Path and π_{ADT}, recompute hashes along Path with π_{ADT} as the same progress in Figure 5.1 and compare the root hash with $\mathsf{comm}_{\mathsf{ADT}}$. Output 1 if they are the same, otherwise output 0.

Given these algorithms and the construction, we have the following theorem:

Theorem 5.3.2. *Let* $\mathbf{T}$ *be a decision tree with* h *levels and* N *nodes, our* ADT *scheme satisfies the following properties.*

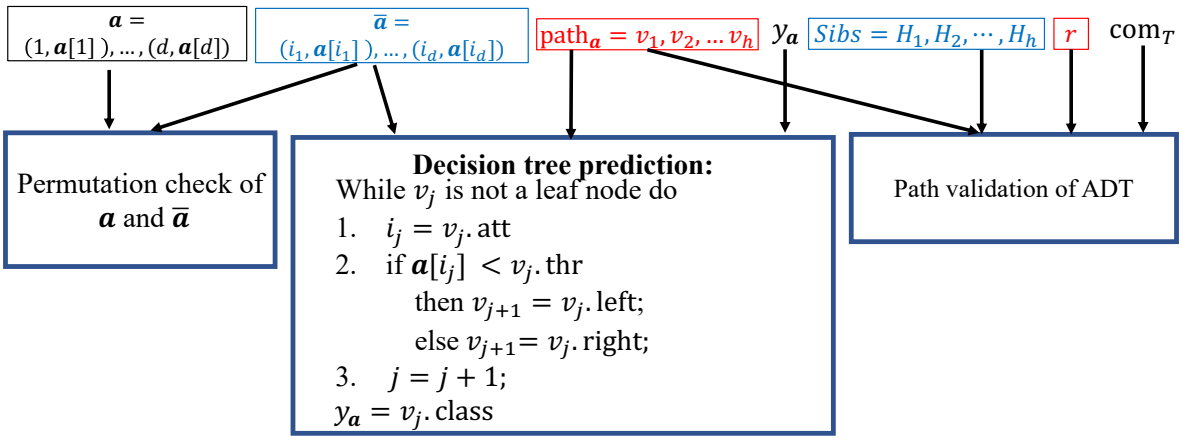


Figure 5.2: Zero knowledge decision tree prediction. Public inputs are in black, secret witness is in red, and extended witness for efficiency is in blue.

- ***Completeness***: *if* $\mathsf{pp} \leftarrow \mathsf{ADT.KeyGen}(1^\lambda)$, $\mathsf{Path} \in \mathbf{T}$, $\mathsf{comm}_{\mathsf{ADT}} \leftarrow \mathsf{ADT.Commit}(\mathbf{T}, \mathsf{pp}, r)$ *and* $\pi_{\mathsf{ADT}} \leftarrow \mathsf{ADT.Prove}(\mathbf{T}, \mathsf{Path}, \mathsf{pp})$, *then*

$$\Pr[\mathsf{ADT.Verify}(\mathsf{comm}_{\mathsf{ADT}}, \mathsf{Path}, \pi_{\mathsf{ADT}}, \mathsf{pp}) = 1] = 1$$

- ***Soundness***: *for any PPT adversary* $\mathcal{A}$, *if* $\mathsf{pp} \leftarrow \mathsf{ADT.KeyGen}(1^\lambda)$, $\mathsf{comm}_{\mathsf{ADT}} \leftarrow \mathsf{ADT.Commit}(\mathbf{T}, \mathsf{pp}, r)$, $\pi^*_{\mathsf{ADT}} \leftarrow \mathcal{A}(\mathbf{T}, \mathsf{Path}, \mathsf{pp})$ *but* $\mathsf{Path} \notin \mathbf{T}$, *then*

$$\Pr[\mathsf{ADT.Verify}(\mathsf{comm}_{\mathsf{ADT}}, \mathsf{Path}, \pi^*_{\mathsf{ADT}}, \mathsf{pp}) = 1] \leq \mathsf{negl}(\lambda)$$

- ***Hiding***: $\mathsf{pp} \leftarrow \mathsf{ADT.KeyGen}(1^\lambda)$, *for any decision tree* $\mathbf{T}$ *with* h *levels, any PPT algorithm* $\mathcal{A}$, *there exists a simulator* $\mathcal{S}_{\mathsf{ADT}}$: *let* $\mathsf{comm}_{\mathsf{ADT}} = \mathsf{ADT.Commit}(\mathbf{T}, \mathsf{pp}, r)$ *and* $\mathsf{comm}'_{\mathsf{ADT}} = \mathcal{S}_{\mathsf{ADT}}(\mathsf{pp}, h, r)$,

$$|\Pr[\mathcal{A}(\mathsf{comm}_{\mathsf{ADT}}, \mathsf{pp}) = 1] - \Pr[\mathcal{A}(\mathsf{comm}'_{\mathsf{ADT}}, \mathsf{pp}) = 1]| \leq \mathsf{negl}(\lambda)$$

In addition, the time of $\mathsf{ADT.Commit}$ *is* $O(N)$, *and the prover time, verification time and the proof size are all* $O(h)$.

Proof Sketch: The completeness of our ADT scheme is straight forward. The soundness holds because of the collision-resistance of the hash function. To prove the hiding property, we can construct a simulator $\mathcal{S}_{\mathsf{ADT}}(\mathsf{pp}, h, r) = \mathsf{ADT.Commit}(\vec{0}_h, \mathsf{pp}, r)$, where $\vec{0}_h$ represents a decision tree with h levels and all nodes containing only 0 strings. It is indistinguishable from the real algorithm because the verifier does not know r, which is uniformly random. We omit the formal proofs here.

5.3.4 Proving the validity of the prediction

Following the algorithm to commit to a decision tree, we further present our protocol to prove the correctness of the prediction. A natural idea is to invoke $\mathsf{ADT}.\mathcal{P}$ and $\mathsf{ADT}.\mathcal{V}$ directly to obtain the valid prediction

path, and check the prediction in Algorithm 14. However, this is not zero knowledge as the verifier would learn a path in the decision tree. We propose to apply an additional zero knowledge proof protocol on top of this validation. As mentioned in the previous section, the prover instead proves that there exists a valid prediction path such that $\mathsf{ADT}.\mathcal{V}$ would accept, and the prediction algorithm is correctly executed. By applying generic zero knowledge proofs on this relationship, the prediction path and the hashes of the siblings remain confidential as the witness, and the output is merely 1 or 0, i.e. all the checks are satisfied or not. In this way, the protocol is both sound and zero knowledge.

However, efficiently designing zero knowledge proof protocols for such an relationship turns out to be non-trivial. This is because every node v on the prediction path for a data sample $\mathbf{a}$ will access one attribute from $\mathbf{a}$ indexed by $v.\mathsf{att}$ for comparison, which is a classical random access operation. Most generic zero knowledge proof protocols represent computations as arithmetic circuits. Implementing the decision tree prediction algorithm as an arithmetic circuit introduces a high overhead on the prover time. In particular, each comparison in an internal node leads to an overhead of $O(d)$, and the overall size of the circuit is $O(dh)$. There are a few RAM-based generic zero knowledge proof protocols [BSCGTV; BSCTV; ZGKPP18; BCGJM18] in the literature that represent computations as RAM programs. However, though asymptotically the prover time only depends on the running time of the RAM program, the concrete overhead is very high (thousands of arithmetic gates per step of the program). Instead, in our construction, we apply some of the ideas in these work in a non-black-box way to design a specific and efficient zero knowledge proof protocol for the validation of decision tree predictions.

Reducing decision tree prediction circuit efficiently.. Figure 5.2 illustrates our design to efficiently reduce the validity of the prediction using a committed decision tree to an arithmetic circuit. As shown in the figure, the public input (in black) of the circuit consists of the data sample $\mathbf{a}$, the commitment of the decision tree $\mathsf{com}_{\mathbf{T}}$ and the prediction result $y_{\mathbf{a}}$. The secret witness (in red) from the prover includes the prediction path $\mathsf{path}_{\mathbf{a}}$, and the randomness r used in the commitment of ADT (for technical reasons to prove zero knowledge). In order to improve the efficiency, the prover further inputs the siblings of nodes on the prediction path, and the permutation $\bar{\mathbf{a}}$ of the data sample $\mathbf{a}$ ordered by $v.\mathsf{att}$ of the nodes on the prediction path as part of the witness (in blue). The purpose of these "extended" witness will be explained below. The whole circuit consists of three parts: (1) validating the prediction algorithm of the decision tree, (2) checking the permutation between $\mathbf{a}$ and $\bar{\mathbf{a}}$, and (3) checking the validity of the prediction path in the committed decision tree. Finally, the output of the circuit is either 1 or 0, denoting either all the conditions are satisfied or some check fails.

Decision tree prediction.. The first component of the circuit is to validate the prediction algorithm. With the help of $\bar{\mathbf{a}}$, this can be efficiently implemented using an arithmetic circuit. In particular, we slightly modify the representation of $\mathbf{a}$ and $\bar{\mathbf{a}}$ to be index-value pairs, i.e., $\mathbf{a} = (1, \mathbf{a}[1]), \ldots, (d, \mathbf{a}[d])$ and $\bar{\mathbf{a}} = (i_1, \mathbf{a}[i_1]), \ldots, (i_d, \mathbf{a}[i_d])$. Under this representation, the circuit simply checks that for every internal node v_j on the prediction path ($j = 1, \ldots, h-1$), (1) $v_j.\mathsf{att} = i_j$, and (2) if $\mathbf{a}[i_j] < v_j.\mathsf{thr}$, $v_{j+1} = v_j.\mathsf{left}$, otherwise $v_{j+1} = v_j.\mathsf{right}$. As we explained in the previous subsection, $v, v.\mathsf{left}, v.\mathsf{right} \in [N]$. The equality tests and comparisons are computed using standard techniques in the literature of circuit-based zero knowledge proof with the help of auxiliary input [PHGR13]. Finally, the circuit checks if $y_{\mathbf{a}} = v_h.\mathsf{class}$. The circuit outputs 1 if all the checks pass, and outputs 0 otherwise. The total number of gates in this part is $O(d + h)$, which is asymptotically the same as the plain decision tree prediction in Algorithm 14.

Note that if $h < d$, which is usually true in practice, the circuit only checks the indices of the first $h - 1$ pairs in $\bar{\mathbf{a}}$. The rest of the indices are arbitrary, as long as $\bar{\mathbf{a}}$ is a permutation of $\mathbf{a}$. It does not affect the correctness or the soundness of the scheme, as those attributes are not used for prediction anyway. In addition, concrete decision trees are usually not balanced. The prover and the verifier can either agree on the length of

the prediction path and construct a separate circuit for every data sample, or use the height of the tree as an upper-bound to construct the same circuit for all data samples. The former is more efficient, but leaks the length of the prediction paths. Both options are supported by our scheme and the asymptotic complexity are the same. For simplicity, we abuse the notation and use h both for the height of the tree and for (an upper bound of) the length of the prediction path.

Permutation test.. The second component is to check that $\bar{\mathbf{a}}$ is indeed a permutation of $\mathbf{a}$. Together with the first component, it ensures that $y_{\mathbf{a}}$ is the correct prediction result of $\mathbf{a}$ using the prediction path $\mathsf{path}_{\mathbf{a}}$. The construction is inspired by the RAM-based zero knowledge proof systems and we also apply the techniques of characteristic polynomials proposed in [ZGKPP17c; ZGKPP18; BCGJM18] to check permutations. In particular, the characteristic polynomial of a vector $\mathbf{c} = (\mathbf{c}[1], \cdots, \mathbf{c}[d]) \in \mathbb{F}^d$ is $\chi_{\mathbf{c}}(x) = \Pi_{i=1}^{d}(x - \mathbf{c}[i])$, the polynomial with roots $\mathbf{c}[i]$. To prove permutations between $\mathbf{c}$ and $\bar{\mathbf{c}}$, it suffices to show that their characteristic polynomial evaluates to the same value at a random point $r \in \mathbb{F}$ chosen by the verifier:

$$\Pi_{i=1}^{d}(r - \mathbf{c}[i]) = \Pi_{i=1}^{d}(r - \bar{\mathbf{c}}[i])$$

The soundness error is $\frac{d}{|\mathbb{F}|}$ by Schwartz-Zippel Lemma [Sch79; Zip79].

In our construction, however, we need to prove that two vectors of pairs $\mathbf{a}$ and $\bar{\mathbf{a}}$ are permutations of each other. To this end, we use the approach proposed in [BCGJM18] to pack each pair to a single value by a random linear combination. The verifier chooses a random point $z \in \mathbb{F}$. For each pair $(j, \mathbf{a}[j])$ in $\mathbf{a}$ and $(i_j, \bar{\mathbf{a}}[j])$ in $\bar{\mathbf{a}}$, the circuit computes $\mathbf{c}[j] = \mathbf{a}[j] + z \times j$ and $\bar{\mathbf{c}}[j] = \bar{\mathbf{a}}[j] + z \times i_j$. We then invoke the characteristic polynomial evaluation directly on $\mathbf{c}$ and $\bar{\mathbf{c}}$.

The completeness is straight forward. To prove the soundness, suppose $\bar{\mathbf{a}}$ is not a permutation of $\mathbf{a}$, then there must exist a pair $(i, \mathbf{a}[i])$ not appearing in $\bar{\mathbf{a}}$. After the packing process, for each j, $\Pr[\mathbf{a}[i] + z \times i = \bar{\mathbf{a}}[j] + z \times i_j | (i, \mathbf{a}[i]) \neq (i_j, \bar{\mathbf{a}}[j])] \leq \frac{1}{|\mathbb{F}|}$ by Schwartz-Zippel Lemma. Therefore, although $\mathbf{a}[i] + z \times i$ is one root of the characteristic polynomial of $\mathbf{c}$, the probability of that it is also one root of the characteristic polynomial for $\bar{\mathbf{c}}$ is at most $\frac{d}{|\mathbb{F}|}$ by the union bound. Combining with the soundness of the characteristic polynomial checking, we obtain that our method has the soundness error at most $\frac{2d}{|\mathbb{F}|}$, which is also negligible of λ. The technique can be extended to verify the permutation of vectors of more than two elements by packing them with a polynomial of z.

The circuit outputs 1 if and only if the above check passes. The number of gates is $O(d)$. Here we assume that every attribute is only used at most once in any prediction path. When an attribute can be used multiple times, which is also very common in practice, the circuit instead checks that $\bar{\mathbf{a}}$ is a *multiset* of $\mathbf{a}$, i.e., every pair in $\mathbf{a}$ appears in $\bar{\mathbf{a}}$ with cardinality greater than or equal to 0. We will present a technique to check the multiset relationship in Section 5.4. The sub-circuits for decision tree prediction and path validation remain unchanged, and the total number of gates in the circuit is the same asymptotically, as the size of $\bar{\mathbf{a}}$ in bounded by h in this case.

Path validation.. Finally, the only missing component is to ensure that the prediction path is indeed valid as committed before. The third sub-circuit implements the $\mathsf{ADT.Verify}$ algorithm with input $\mathsf{path}_{\mathbf{a}}$, all sibling hash values on $\mathsf{path}_{\mathbf{a}}$, random point r and $\mathsf{com}_{\mathbf{T}}$. The circuit recomputes the hashes from the leaf to the root, and compares the root hash with $\mathsf{com}_{\mathbf{T}}$. In total, the circuit computes $O(h)$ hashes, which justifies the design of our ADT scheme.

Finally, the circuit aggregates all three checks and output 1 if and only if all checks pass. The size of the whole circuit is $O(d + h)$, which is also asymptotically optimal. The prover and the verifier then execute a generic circuit-based zero knowledge proof protocol on the circuit in Figure 5.2.

Protocol 13 (Zero Knowledge Decision Tree (zkDT)). Let λ be the security parameter, $\mathbb{F}$ be a prime field, $\mathbf{T}$ be a decision with h levels, C be the arithmetic circuit in Figure 5.2. Let $\mathcal{P}$ and $\mathcal{V}$ be the prover and the verifier respectively. We use $\mathsf{ZKP.KeyGen}$, $\mathsf{ZKP.Commit}$, $\mathsf{ZKP}.\mathcal{P}$, $\mathsf{ZKP}.\mathcal{V}$ to represent the algorithms of the backend ZKP protocol.

- $\mathsf{pp} \leftarrow \mathsf{zkDT.KeyGen}(1^\lambda)$: let $\mathsf{pp}_1 \leftarrow \mathsf{ADT.KeyGen}(1^\lambda)$, $\mathsf{pp}_2 \leftarrow \mathsf{ZKP.KeyGen}(1^\lambda)$ and $\mathsf{pp} = (\mathsf{pp}_1, \mathsf{pp}_2)$.
- $\mathsf{comm}_{\mathbf{T}} \leftarrow \mathsf{zkDT.Commit}(\mathbf{T}, \mathsf{pp}, r)$: $\mathsf{comm}_{\mathbf{T}} \leftarrow \mathsf{ADT.Commit}(\mathbf{T}, \mathsf{pp}_1, r)$, where r is the randomness generated by $\mathcal{P}$.
- $(y_{\mathbf{a}}, \pi) \leftarrow \mathsf{zkDT.Prove}(\mathbf{T}, \mathbf{a}, \mathsf{pp})$:
  1. $\mathcal{P}$ runs the algorithm 14 with input $\mathbf{T}$ and $\mathbf{a}$ to get $y_{\mathbf{a}} = \mathbf{T}(\mathbf{a})$. Then generates the witness $w = (\bar{\mathbf{a}}, path_{\mathbf{a}}, \mathsf{aux}, r)$ for the circuit C in accordance with the procedure of the decision tree algorithm. aux represents the extended witness in Figure 5.2. Let $\mathsf{comm}_w \leftarrow \mathsf{ZKP.Commit}(w, \mathsf{pp}_2)$. $\mathcal{P}$ sends comm_w and $y_{\mathbf{a}}$ to $\mathcal{V}$.
  2. After receiving the randomness $\mathbf{r}'$ for checking the permutation of $\mathbf{a}$ and $\bar{\mathbf{a}}$ from $\mathcal{V}$, $\mathcal{P}$ invokes $\mathsf{ZKP.Prove}(C, (\mathsf{comm}_T, \mathbf{a}, y_{\mathbf{a}}, \mathbf{r}'), w, pp_2)$ to get π. Sends π to $\mathcal{V}$.
- $\{0,1\} \leftarrow \mathsf{zkDT.Verify}(\mathsf{com}_{\mathbf{T}}, h, \mathbf{a}, y_{\mathbf{a}}, \pi, \mathsf{pp})$: $\mathcal{V}$ outputs 1 if $\mathsf{ZKP.Verify}(C, (\mathsf{comm}_T, \mathbf{a}, y_{\mathbf{a}}, \mathbf{r}'), \pi, \mathsf{comm}_w, pp_2) = 1$, otherwise it outputs 0.

5.3.5 Putting Everything Together

In this section, we combine everything together and formally present our zero knowledge decision tree prediction scheme in Protocol 13. Our scheme has a transparent setup phase where $\mathsf{zkDT}.\mathcal{G}$ does not have a trapdoor. It merely samples a collision-resistant hash function for ADT, and executes the algorithm $\mathcal{G}$ in the generic zero knowledge proof. Using Aurora as our backend, it also samples a hash function modeled as a random oracle.

We have the following theorem:

Theorem 5.3.3. *Protocol 13 is a zero knowledge decision tree scheme by Definition 5.3.1.*

Proof. **Completeness.**. As explained in Section 5.3.3 and 5.3.4, the circuit in $\mathsf{zkDT}.\mathcal{P}$ outputs 1 if $y_{\mathbf{a}}$ is the correct prediction of $\mathbf{a}$ output by Algorithm 14 on $\mathbf{T}$. Therefore, the correctness of Protocol 13 follows the correctness of the ADT and the zero knowledge proof protocol by Theorem 5.3.2 and 5.2.2.

Soundness.. By the extractability of commitment of in Theorem 5.2.2, with overwhelming probability, there exists a PPT extractor $\mathcal{E}$ such that given comm_w, it extracts a witness w^* such that $\mathsf{comm}_w = \mathsf{ZKP.Commit}(w^*, \mathsf{pp}_2)$. By the soundness of zkDT in Definition 5.3.1, if $\mathsf{comm}_{\mathbf{T}} = \mathsf{zkDT.Commit}(\mathbf{T}, \mathsf{pp}, r)$ and $\mathsf{zkDT.Verify}(\mathsf{com}_{\mathbf{T}}, h, \mathbf{a}, y_{\mathbf{a}}, \pi, \mathsf{pp}) = 1$ but $y_{\mathbf{a}} \neq \mathbf{T}(\mathbf{a})$, let $\mathsf{comm}_w = \mathsf{ZKP.Commit}(w^*, \mathsf{pp}_2)$ during the interactive process in Protocol 13, then there are two cases.

[Simulator for Protocol 13] Let λ be the security parameter, $\mathbb{F}$ be a prime field, $\mathbf{T}$ be a decision with h levels, C be the arithmetic circuit in Figure 5.2. $(\mathsf{pp}_1, \mathsf{pp}_2) \leftarrow \mathsf{zkDT.KeyGen}(1^\lambda)$.

- $\mathsf{comm} \leftarrow \mathcal{S}_1(1^\lambda, \mathsf{pp}, h)$: S_1 invokes $\mathcal{S}_{\mathsf{ADT}}$ to generate $\mathsf{comm} = \mathcal{S}_{\mathsf{ADT}}(\mathsf{pp}_1, h, r)$, where r is the randomness generated by S_{ADT}.
- $(y_{\mathbf{a}}, \pi) \leftarrow \mathcal{S}_2^{\mathcal{A}}(h, \mathbf{a}, \mathsf{pp})$:
  1. $\mathcal{S}_2$ asks the oracle of $\mathbf{T}$ to get $y_{\mathbf{a}} = \mathbf{T}(\mathbf{a})$. Then $\mathcal{S}_2$ shares all public input of C to $\mathcal{S}_{\mathsf{ZKP}}$ and invokes $\mathcal{S}_{\mathsf{ZKP}}.\mathsf{Commit}(\mathsf{pp}_2)$ to get comm_w.
  2. After receiving the randomness $\mathbf{r}'$ for the permutation check from $\mathcal{A}$, it invokes $\mathcal{S}_{\mathsf{ZKP}}.\mathsf{Prove}(C, (\mathsf{comm}, \mathbf{a}, y_{\mathbf{a}}, \mathbf{r}'), \mathsf{pp}_2)$ to get π. Then S_2 sends π to $\mathcal{A}$.
- $\{0,1\} \leftarrow \mathcal{A}(\mathsf{com}, h, \mathbf{a}, y_{\mathbf{a}}, \pi, \mathsf{pp})$: wait $\mathcal{A}$ for validation.

- Case 1: $w^* = (\overline{\mathbf{a}}^*, path^*_{\mathbf{a}}, aux^*, r)$ satisfying to $C((\mathsf{comm}_{\mathbf{T}}, \mathbf{a}, y_{\mathbf{a}}, \mathbf{r}'); w^*) = 1$. Then we could know either $\mathsf{path}^*_{\mathbf{a}}$ is not a path in $\mathbf{T}$ but passing the verification for $\mathsf{comm}_{\mathbf{T}}$, or $\overline{\mathbf{a}}^*$ is not a permutation of $\mathbf{a}$ but passing the permutation test. The probability of both events are $\mathsf{negl}(\lambda)$ as claimed by the soundness the ADT scheme and the soundness of the characteristic polynomial check respectively. Hence, the probability that $\mathcal{P}$ could generate such w^* is also $\mathsf{negl}(\lambda)$ by the union bound.
- Case 2: $w^* = (\overline{\mathbf{a}}^*, path^*_{\mathbf{a}}, aux^*, r)$ but $C((\mathsf{comm}_{\mathbf{T}}, \mathbf{a}, y_{\mathbf{a}}, \mathbf{r}'); w^*) = 0$. Then according to the soundness of Aurora, given the commitment comm^*_w, the adversary could generate a proof π_w making $\mathcal{V}$ accept the incorrect witness and output 1 with probability $\mathsf{negl}(\lambda)$.

Combining these two cases, the soundness of the zkDT scheme is also $\mathsf{negl}(\lambda)$.

Zero-knowledge.. In order to prove the zero-knowledge property, we construct a PPT simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ in Figure 5.3. Let $\mathcal{S}_{\mathsf{ADT}}$ and $\mathcal{S}_{\mathsf{ZKP}}$ represent the simulator for ADT protocol and the backend ZKP protocol respectively. The proof follows by a standard hybrid argument.
Hybrid H_0.: H_0 behaves in exactly the same way as the honest prover in Protocol 13.
Hybrid H_1.: H_1 uses the real $\mathsf{zkDT.Commit}(\mathbf{T}, \mathsf{pp}, r)$ in Protocol 13 for the commitment phase, it invokes $\mathcal{S}_{\mathsf{ZKP}}$ to simulate the interactive proof phase.
Hybrid H_2.: H_2 behaves in exactly the same way as the simulator of Protocol 13.

Given the same commitment $\mathsf{comm}_{\mathbf{T}}$, the verifier cannot distinguish H_0 and H_1, because of the zero knowledge property of the backend ZKP protocol given the same circuit C and the same public input. If the verifier could distinguish H_1 from H_2, then we can find a PPT adversary to distinguish whether a commitment is of an empty decision tree with only zero strings or not. This is contradictory to the hiding property of our ADT scheme. Therefore, the verifier cannot distinguish H_0 from H_2 by the hybrid, which completes the proof of zero knowledge.

Efficiency.. The prover's computation consists of two parts: committing to the decision tree $\mathbf{T}$ and generating the proof for circuit C. For the committing phase, the provers needs to do $O(N)$ hashes. For the

proof generation phase, the total computation is $O(|C| \log |C|) = O((d+h)\log(d+h)) = O(d \log d)$ in accordance with Theorem 5.2.2 and $d > h$. The verifier's computation only contains the verification for the circuit C with size $O(d+h)$, so it is $O(|C|) = O(d)$ due to Theorem 5.2.2. The proof size is one digest plus the Aurora proof for the circuit C, which is only $O(\log^2 |C|) = O(\log^2 d)$. Finally, we can apply the Fiat-Shamir heuristic [FS] to remove the interactions in our zkDT protocol in the random oracle model.

5.4 Zero Knowledge Decision Tree Accuracy

In this section, we present our scheme for zero knowledge decision tree accuracy. As motivated in the introduction, in this scenario, the prover owns and commits to a decision tree, receives a testing dataset from the verifier, and then proves the accurary of the committed decision tree model on this dataset. The verifier learns nothing about the model except its accuracy.

Formally speaking, similar to Section 5.3.2, suppose the decision tree model is $\mathbf{T}$ with h levels and N nodes, where h and N are known to both parties, the testing dataset is $\mathcal{D} = \{\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_n\}$ with n data samples and their matching labels are $\mathbb{L} = \{\ell_1, \ell_2, \cdots, \ell_n\}$. A zero knowledge decision tree accuracy (zkDTA) scheme consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{zkDTA.KeyGen}(1^\lambda)$: given the security parameter, generate the public parameter pp.
- $\mathsf{comm}_{\mathbf{T}} \leftarrow \mathsf{zkDTA.Commit}(\mathbf{T}, \mathsf{pp}, r)$: commit the decision tree $\mathbf{T}$ with a random point r generated by the prover.
- $(\mathsf{accu}, \pi) \leftarrow \mathsf{zkDTA.Prove}(\mathbf{T}, \mathcal{D}, \mathbb{L}, \mathsf{pp})$: given a dataset $\mathcal{D}$ and their labels $\mathbb{L}$, run the decision tree algorithm for each data sample in $\mathcal{D}$, compare the predictions with $\mathbb{L}$, and then output accu denoting the accuracy of the decision tree, i.e., the total number of correct predictions. The algorithm also generates the corresponding proof π.
- $\{0,1\} \leftarrow \mathsf{zkDTA.Verify}(\mathsf{comm}_{\mathbf{T}}, h, N, \mathcal{D}, \mathbb{L}, \mathsf{accu}, \pi, \mathsf{pp})$: validate the number of correct predictions, accu, given π obtained from the prover.

Definition 5.4.1. *We say that a scheme is a zero knowledge decision tree accuracy if the following holds:*

- ***Completeness.*** *For any decision tree $\mathbf{T}$ with h levels and N nodes, a test dataset $\mathcal{D}$ with corresponding labels $\mathbb{L}$, $\mathsf{comm}_{\mathbf{T}} \leftarrow \mathsf{zkDTA.Commit}(\mathbf{T}, \mathsf{pp}, r)$, $(\mathsf{accu}, \pi) \leftarrow \mathsf{zkDTA.Prove}(\mathbf{T}, \mathcal{D}, \mathbb{L}, \mathsf{pp})$, it holds that*

$$\Pr\left[\mathsf{zkDTA.Verify}(\mathsf{comm}_{\mathbf{T}}, h, N, \mathcal{D}, \mathbb{L}, \mathsf{accu}, \pi, \mathsf{pp}) = 1\right] = 1$$

- ***Soundness.*** *For any PPT adversary $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{zkDTA.KeyGen}(1^\lambda) \\ (\mathbf{T}^*, \mathsf{comm}_{\mathbf{T}^*}, \mathcal{D}, \mathbb{L}, \mathsf{accu}^*, \pi^*) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{r}) \\ \mathsf{comm}_{\mathbf{T}^*} = \mathsf{zkDTA.Commit}(\mathbf{T}^*, \mathsf{pp}, r) \\ \mathsf{zkDTA.Verify}(\mathsf{comm}_{\mathbf{T}^*}, h, N, \mathcal{D}, \mathbb{L}, \mathsf{accu}^*, \pi^*, \mathsf{pp}) = 1 \\ \sum_{i=1}^{n} I(\mathbf{T}^*(\boldsymbol{a}_i) = \ell_i) \neq \mathsf{accu}^* \end{array}\right]$$

$I(\mathbf{T}^*(\boldsymbol{a}_i) = \ell_i) = 1$ *if* $\mathbf{T}^*(\boldsymbol{a}_i) = \ell_i$, *otherwise* $I(\mathbf{T}^*(\boldsymbol{a}_i) = \ell_i) = 0$.

- ***Zero Knowledge.*** *For security parameter* λ, $\mathsf{pp} \leftarrow \mathsf{zkDT.KeyGen}(1^\lambda)$*, for any decision tree* $\mathbf{T}$ *with* h *levels and* N *nodes, PPT algorithm* $\mathcal{A}$*, and simulator* $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$*, consider the following two experiments:*

$\mathsf{Real}_{\mathcal{A},\mathbf{T}}(\mathsf{pp})$*:*
1. $\mathsf{com}_{\mathbf{T}} \leftarrow \mathsf{zkDTA.Commit}\ (\mathbf{T}, \mathsf{pp}, r)$
2. $\mathcal{D}, \mathbb{L} \leftarrow \mathcal{A}(h, N, \mathsf{com}_{\mathbf{T}}, \mathsf{pp})$
3. $(\mathsf{accu}, \pi) \leftarrow \mathsf{zkDTA.Prove}(\mathbf{T}, \mathcal{D}, \mathbb{L}, \mathsf{pp})$
4. $b \leftarrow \mathcal{A}(\mathsf{com}_{\mathbf{T}}, h, N, \mathcal{D}, \mathbb{L}, \mathsf{accu}, \pi, \mathsf{pp})$
5. *Output b*

$\mathsf{Ideal}_{\mathcal{A},\mathcal{S}^{\mathcal{A}}}(\mathsf{pp}, h, N)$*:*
1. $\mathsf{com} \leftarrow \mathcal{S}_1(1^\lambda, \mathsf{pp}, h, N)$
2. $\mathcal{D}, \mathbb{L} \leftarrow \mathcal{A}(h, N, \mathsf{com}, \mathsf{pp})$
3. $(\mathsf{accu}, \pi) \leftarrow \mathcal{S}_2^{\mathcal{A}}(\mathsf{com}, h, N, \mathcal{D}, \mathbb{L}, \mathsf{pp})$*, given oracle access to* $\mathsf{accu} = \sum_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{I}(\mathbf{T}(\boldsymbol{a}_{\mathsf{i}}) = \ell_{\mathsf{i}})$.
4. $b \leftarrow \mathcal{A}(\mathsf{com}, h, N, \mathcal{D}, \mathbb{L}, \mathsf{accu}, \pi, \mathsf{pp})$
5. *Output b*

For any PPT algorithm $\mathcal{A}$ *and all decision tree* $\mathbf{T}$ *with the height of* h *and* N *nodes, there exists simulator* $\mathcal{S}$ *such that*

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

5.4.1 Checking Multiset

Intuitively, when designing the zkDTA scheme, one can repeat the construction in Section 2 described in Protocol 13 and Figure 5.2 multiple times for every data sample in the testing dataset, followed by an aggregation circuit testing the accuracy. The prover time in this case grows roughly linearly with the size of the testing dataset. However, the prediction paths share many common nodes and their total size may exceed the number of nodes in the decision tree on a large testing dataset. I.e., $N < nh$.

We introduce an optimization for this case. Instead of validating each prediction path one by one, the idea is to validate all N nodes of the decision tree in one shot. Then the circuit checks that the nodes of each prediction path are drawn from these N nodes of the decision tree and they form a path with the correct parent-children relationships. On top of these checks, the circuit tests the correctness of Algorithm 14 in the same way as the zkDT scheme and computes the accuracy of the model. To test that each node in the prediction paths is included in the original decision tree, it suffices to check that all nodes of the prediction paths form a *multiset* of the set of N nodes of the decision tree. We can again validate such a multiset relationship using the characteristic polynomials.

Multiset check.. Suppose $Q = (q_1, q_2, \cdots, q_m)$ is an array of m elements with possible duplicates, and $S = \{s_1, \cdots, s_n\}$ is a set of size n. The prover needs to show that Q is a multiset of S, i.e., $\forall i \in [m], q_i \in S$.

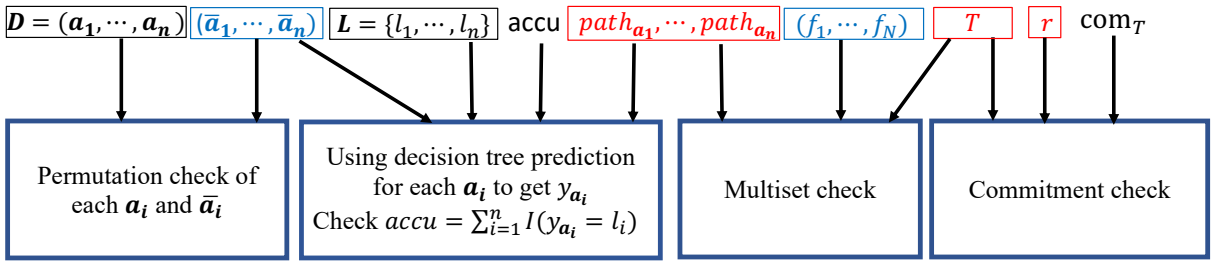


Figure 5.4: Zero knowledge decision tree accuracy.

We apply the technique proposed in [ZGKPP18]. The characteristic polynomial of a multiset Q is defined as $\Pi_{i=1}^{m}(x - q_i)$ with possible duplicated elements in Q. It can also be computed as $\Pi_{i=1}^{n}(x - s_i)^{f_i}$, where f_i is the multiplicity of each element s_i in array Q. Therefore, to check the multiset relationship, the prover provides the multiplicity f_i, the verifier picks a random number r and the circuit tests $\Pi_{i=1}^{m}(r - q_i) = \Pi_{i=1}^{n}(r - s_i)^{f_i}$. We call polynomial $g(x) = \Pi_{i=1}^{m}(x - q_i) - \Pi_{i=1}^{n}(x - s_i)^{f_i}$ the multiset polynomial. The soundness of the test also follows the Schwartz-Zippel Lemma [Sch79; Zip79]. If there exists $q_i \notin S$, then the multiset polynomial $g(x)$ is a non-zero polynomial with degree at most m, if we additionally force $\sum_{i=1}^{n} f_i = m$. Then $\Pr[g(r) = 0 | r \xleftarrow{\$} \mathbb{F}] \leq \frac{m}{|\mathbb{F}|}$ by the Schwartz-Zippel Lemma. So the soundness error is at most $\frac{m}{|\mathbb{F}|} = \mathsf{negl}(\lambda)$.

To implement this test in an arithmetic circuit, as there is no exponentiation gate, we ask the prover to provide f_i in binary. Then the circuit checks that the inputs are indeed binary, and uses the multiplication tree to compute the exponentiation. As $\sum_{i=1}^{n} f_i = m$ and $f_i \leq m$, the prover only provides $\log m$ bits for each f_i.

With the multiset check, in our zkDTA scheme, the prover provides all N nodes in $\mathbf{T}$ and proves that all the nodes in the prediction paths form a multiset of the N nodes. In addition, the circuit reconstructs the ADT using the N nodes of $\mathbf{T}$ and checks that it is consistent with $\mathsf{comm}_{\mathbf{T}}$. In this way, the total number of hashes is bounded by N if $N = 2^h$. The number of gates for the multiset check is at most $O(nh + N\log(n))$. In the real implementation, computing hashes is usually the bottleneck of the efficiency. For example, SHA-2 takes around $270,000$ multiplication gates and algebraic hash functions[Ajt96] take hundreds to thousands of gates to implement. Our optimization reduces the number of hashes from $O(nh)$ to $O(N)$, which greatly improves the performance of the ZKDTA scheme in practice.

Furthermore, the method of the multiset check can also be used in our zkDT protocol in Protocol 13 to support decision trees with repeated attributes on the prediction paths, which is very common in practice. We instead test that $\bar{\mathbf{a}}$ is a multiset rather than a permutation of $\mathbf{a}$ in this case.

5.4.2 Validating Decision Tree

In the previous optimization, the circuit checks that N nodes provided by the prover form a valid decision tree. Using ADT, we can validate it by reconstructing the decision tree in the circuit 2^h hashes. However, in practice, we notice that most decision trees are not balanced. For example, in our largest decision tree model, there are total 23 levels and 1029 nodes. In this case, $N \ll 2^h$. Therefore, in this section, we present an approach to validate a decision tree with a circuit of size linear to N rather than 2^h.

We first replace the commitment by a hash of all N nodes concatenated by a random value r, instead of the root of the ADT. In addition, each node contains a unique id(id) in $[N]$, the id of its parent (pid), left child

Protocol 14 (Zero Knowledge Decision Tree Accuracy). Let λ be the security parameter, $\mathbb{F}$ be a prime field, $\mathbf{T}$ be a binary decision with h levels and N nodes, C_A be the arithmetic circuit for model accuracy test, $\mathcal{D} = \{\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_n\}$ of size n be the test set, $\mathbb{L} = \{\ell_1, \ell_2, \cdots, \ell_n\}$ be the corresponding labels of test data. Let $\mathcal{P}_A$ and $\mathcal{V}_A$ be the prover and the verifier respectively. ZKP.KeyGen, ZKP.Commit, ZKP.$\mathcal{P}$, ZKP.$\mathcal{V}$ represent the algorithms of the backend ZKP protocol.

- $\mathsf{pp} \leftarrow \mathsf{zkDTA.KeyGen}(1^\lambda)$: $\mathsf{pp} \leftarrow \mathsf{ZKP.KeyGen}(1^\lambda)$.
- $\mathsf{comm}_\mathbf{T} \leftarrow \mathsf{zkDTA.Commit}(\mathbf{T}, \mathsf{pp}, r)$: $\mathsf{comm}_\mathbf{T} = \mathsf{Hash}(\mathbf{T}, \mathsf{r})$, it hashes all nodes in $\mathbf{T}$ with r, the randomness generated by $\mathcal{P}_A$.
- $(\mathsf{accu}, \pi) \leftarrow \mathsf{zkDTA.Prove}(\mathbf{T}, \mathcal{D}, \mathbb{L}, \mathsf{pp})$: we could use Fiat-Shamir heuristic transformation to make the following process non-interactive.
  1. $\mathcal{P}_A$ runs the algorithm 14 with input $\mathbf{T}$ and the data set $\mathcal{D}$ to get $y_{\mathbf{a}_i} = \mathbf{T}(\mathbf{a}_i)$ for all i. Let $\mathsf{accu} = \sum_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{I}(\mathsf{y}_{\mathbf{a}_\mathsf{i}} = \ell_\mathsf{i})$. According the procedure of the algorithm, the prover could generate the witness $\overline{\mathbf{a}_i}$ and $path_{\mathbf{a}_i}$ for each data point $\mathbf{a}_i$, together with all nodes in $\mathbf{T}$, their multiplicity in $\{path_{\mathbf{a}_i}\}_{i=1}^n$ and the randomness r used in $\mathsf{comm}_\mathbf{T}$ as the witness w_A of the circuit C_A. Let $\mathsf{comm}_{w_A} \leftarrow \mathsf{ZKP.Commit}(w_A, \mathsf{pp}_2)$. $\mathcal{P}_A$ sends comm_{w_A} and accu to $\mathcal{V}_A$.
  2. After receiving the randomness $\mathbf{r}'$ for checking characteristic polynomials and the multiset polynomial from $\mathcal{V}_A$. $\mathcal{P}_A$ invokes $\mathsf{ZKP.Prove}(C_A, (\mathsf{comm}_T, \mathcal{D}, \mathbb{L}, \mathsf{accu}, \mathbf{r}'), \mathsf{w_A}, \mathsf{pp_2})$ to get π. Send π to $\mathcal{V}_A$.
- $\{1, 0\} \leftarrow \mathsf{zkDTA.Verify}(\mathsf{com}_\mathbf{T}, h, N, \mathcal{D}, \mathbb{L}, \mathsf{accu}, \pi, \mathsf{pp})$: $\mathcal{V}_A$ outputs 1 if $\mathsf{ZKP.Verify}(C_A, (\mathsf{comm}_T, \{\mathbf{a}\}_{i=1}^n, \mathsf{accu}, \mathbf{r}'), \pi, \mathsf{comm}_{\mathsf{w_A}}, \mathsf{pp}) = 1$, outputs 0 otherwise.

(lid), right child (rid) and its depth (depth) in $[h]$ (the id is 0 means the parent or the child is empty). To verify that all N nodes of $T_1, T_2, \cdots, T_N$ form a binary decision tree, it suffices to check the following conditions:

- Only the first node is the root. (i.e, $T_1.\mathsf{pid} = 0$ but $T_i.\mathsf{pid} \neq 0$ when $i \neq 1$.)
- All parent pointers are consistent with the child pointers. (i.e., $T_i.\mathsf{rid} = T_j.\mathsf{id}$ or $T_i.\mathsf{lid} = T_j.\mathsf{id}$ if and only if $T_j.\mathsf{pid} = T_i.\mathsf{id}$.)
- The depth of the parent is smaller than the depth of the child. (i.e., if $T_i.\mathsf{pid} = T_j.\mathsf{id}$ then $T_i.\mathsf{depth} = T_j.\mathsf{depth} + 1$.)
- No repeated child pointers. (i.e., if $T_i.\mathsf{lid} \neq 0$ then $T_i.\mathsf{lid} \neq T_i.\mathsf{rid}$.)

With this idea in the mind, the formal algorithm is presented in Algorithm 15. The main difference is that the checks in Algorithm 15 can be efficiently implemented by arithmetic circuits.

Theorem 5.4.2. *Algorithm 15 outputs 1 if and only if the input nodes form a valid binary tree except for negligible probability. The total number of arithmetic gates to implement the algorithm is $O(N)$.*

Proof. On the one hand, if the N nodes of $T_1, T_2, \cdots, T_N$ construct a valid binary tree, it is easy to check they will pass all checks and Algorithm 15 always outputs 1.

Algorithm 15 Linear Check for Valid Decision Tree

Input: N nodes of $T_1, T_2, \cdots, T_N$.

1: T_1 is the root: $T_1.\mathsf{id} = 1 \wedge T_1.\mathsf{depth} = 1 \wedge T_1.\mathsf{pid} = 0$.
2: **for** $i = 1$ to $N - 1$ **do**
3: $\quad T_{i+1}.\mathsf{id} = T_i.\mathsf{id} + 1$
4: **end for**
5: **for** $i = 1$ to $N - 1$ **do**
6: $\quad T_{i+1}.\mathsf{depth} = T_i.\mathsf{depth} \vee T_{i+1}.\mathsf{depth} = T_i.\mathsf{depth} + 1$
7: **end for**
8: $T_N.\mathsf{depth} \leq h$.
9: Check all pids except T_1 of $\{T_2.\mathsf{pid}, \cdots, T_N.\mathsf{pid}\}$ are a multiset of $\{1, 2, 3, \cdots, N\}$ with individual multiplicity at most 2.
10: Define two vectors of tuples, S_1 and S_2.
11: **for** $i = 1$ to N **do**
12: $\quad$ **if** $T_i.\mathsf{lid} \neq 0$ **then**
13: $\quad\quad S_1 = S_1.\mathsf{append}((T_i.\mathsf{depth}, T_i.\mathsf{id}, T_i.\mathsf{lid}))$
14: $\quad$ **end if**
15: $\quad$ **if** $T_i.\mathsf{rid} \neq 0$ **then**
16: $\quad\quad S_1 = S_1.\mathsf{append}((T_i.\mathsf{depth}, T_i.\mathsf{id}, T_i.\mathsf{rid}))$
17: $\quad$ **end if**
18: $\quad$ **if** $T_i.\mathsf{pid} \neq 0$ **then**
19: $\quad\quad S_2 = S_2.\mathsf{append}((T_i.\mathsf{depth} - 1, T_i.\mathsf{pid}, T_i.\mathsf{id}))$
20: $\quad$ **end if**
21: **end for**
22: Check S_1 is a permutation of S_2.
23: **return** 1 if all check pass.

On the other hand, suppose Algorithm 15 outputs 1. With overwhelming probability, (1) $T_1.\mathsf{depth} = 1$ and $T_1.\mathsf{pid} = 0$; (2) $T_i.\mathsf{id} = i$ for all i; (3) all depths is a multiset of $\{1, 2, \cdots, h\}$; (4) all pids except $T_1.pid$ are in $[N]$; (5) if $T_i.\mathsf{rid} = T_j.\mathsf{id}$ or $T_i.\mathsf{lid} = T_j.\mathsf{id}$ then $T_j.\mathsf{pid} = T_i.\mathsf{id}$ and $T_j.\mathsf{depth} = T_i.\mathsf{depth} + 1$; (6) if $T_i.\mathsf{pid} = T_j.\mathsf{id}$ then $T_j.\mathsf{lid} = T_i.\mathsf{id}$ or $T_j.\mathsf{rid} = T_i.\mathsf{id}$ and $T_j.\mathsf{depth} = T_i.\mathsf{depth} - 1$; (7) if $T_i.lid \neq T_i.rid$ unless $T_i.lid = T_i.rid = 0$. If one of (1), (2), (3), (4) does not hold, it will not pass the checks from line 1 to line 7. If (5) does not hold but S_1 is a permutation of S_2, we know that $(T_i.\mathsf{depth}, T_i.\mathsf{id}, T_i.\mathsf{lid}) = (T_i.\mathsf{depth}, T_i.\mathsf{id}, T_j.\mathsf{id}) \in S_1$ or $(T_i.\mathsf{depth}, T_i.\mathsf{id}, T_i.\mathsf{rid}) = (T_i.\mathsf{depth}, T_i.\mathsf{id}, T_j.\mathsf{id}) \in S_1$. There must exist an index $k \in [N]$ such that $(T_k.\mathsf{depth} - 1, T_k.\mathsf{pid}, T_k.\mathsf{id}) = (T_i.\mathsf{depth}, T_i.\mathsf{id}, T_j.\mathsf{id})$. Then we have $k = j$, $T_j.\mathsf{pid} = T_i.\mathsf{id}$ and $T_i.\mathsf{depth} = T_j.\mathsf{depth} - 1$, which is contradictory. Therefore, if (5) does not hold, S_1 is not a permutation of S_2. With very similar argument, if (6) does not hold, S_1 is not a permutation of S_2. If (7) does not hold, S_1 is not a permutation of S_2 as S_1 has duplicate element while S_2 does not. Therefore, if (5) or (6) or (7) does not hold, then it will not pass the check in line 16 with high probability.

When (1)-(7) hold, we can construct a directed graph $G = (T_{[N]}, E)$ as follows. $(T_i, T_j) \in E$ if and

only if $T_i.\mathsf{pid} = T_j.\mathsf{id}$. The outdegree of T_1 is 0 while all others are 1. To prove G is a tree, we only need to show T_i connects to T_1 for each i, or equivalently, there is no circle in the graph. That is because $T_i.\mathsf{depth} = T_j.\mathsf{depth} - 1$ if $(T_i, T_j) \in E$. So G does not have a circle and it must be a tree. It is a binary tree as the indegree of each nodes are at most 2.

Complexity.. Step 1-6 and Step 8-15 can be computed by a circuit doing a linear scan of all the N nodes. In addition,as the the individual multiplicity is at most 2, both the multiset test in Step 7 and the permutation test in Step 16 are $O(N)$ as we explained in previous sections. Furthermore, we force the N nodes sorted by the id and the depth. It consumes only $O(N)$ to check $T_i.\mathsf{id} = i$ and the depth ranging in $[h]$. So the total number of the gates is $O(N)$.

An alternative approach.. Alternatively, because of our application of decision trees, we can simplify the checks above. Recall that in other parts of the circuit, it is proven that the nodes of the prediction paths are drawn from the set of N nodes, and each prediction path follows the prediction function of a decision tree. Therefore, the graph formed by these nodes already satisfies the second and the third conditions of being a binary decision tree. Because of this, we simplify the test to (1) the N nodes are sorted by the id, $T_i.\mathsf{id} = i$, where T_1 is the root and (2) $T_i.lid \neq T_i.rid$ unless they are empty for all $i \in [N]$. These checks ensure that the subgraph formed by all nodes in the prediction paths is part of a binary decision tree. These nodes are also included in the N nodes because of the multiset check. There is no guarantee on other nodes of the N nodes provided by the prover, but they are not used in the prediction paths anyway, which is good enough for our purposes.

This alternative approach is simpler than the check described in Algorithm 15. It is a trade-off between the efficiency and the security. In practice, as computing hashes is the bottleneck of our system, the difference between these two approaches on the prover time is actually not significant. We use the latter in our implementation.

5.4.3 Our construction of zkDT accuracy

With the optimizations presented in the previous sections, we show the circuit C_A to validate decision tree accuracy in Figure 5.4, and present the formal protocol of zkDTA in Protocol 14. Compared with the circuit C in Figure 5.2 for the zkDT scheme, C_A has extra extended witness with all nodes of $\mathbf{T}$ and an auxiliary list $F = (f_1, f_2, \cdots, f_N)$ representing the multiplicity of N nodes appearing in all prediction paths of the test data. C_A also has one more part for multiset check. Besides, C_A does not need to do path validation for each path, it only recomputes the hash of all nodes in $\mathbf{T}$ with r and compares the final value with the commitment. We call it commitment check. Furthermore, in this new scheme, all $y_{\mathbf{a}_i}$ are not public to the verifier. The circuit compares $y_{\mathbf{a}_i}$ with ℓ_i and computes the number of correct predictions. Finally, the circuit compares the number of correct predictions to the claimed accuracy. We have the following theorem:

Theorem 5.4.3. *Protocol 14 is a zero knowledge decision tree accuracy scheme defined by Definition 5.4.1.*

The completeness, soundness and zero-knowledgeness of Protocol 14 are extensions of these properties in Protocol 13. We omit the proof because of the space limitation.

Efficiency.. Consider the circuit C_A, the circuit size is $O(nh + nd + N \log n + N) = O(nd)$ when $N \ll nd$. Therefore, the prover time is $O(nd \log(nd))$ with $O(N)$ hashes in the committing phase, the verification time is $O(nd)$ and proof size is $O(\log^2(nd))$ according to Theorem 5.2.2.

5.5 Implementation and Evaluations

We fully implement our zero knowledge decision tree schemes and we present their performance in this section.

Software.. The schemes are implemented in C++. There are around 2000 lines of code for our frontend to compile the decision tree predictions and accuracy to arithmetic circuits[1], as shown in Figure 5.2 and 5.4. We use the open-source compiler of libsnark[Libb] to generate arithmetic circuits in our frontend, and we implement the zero knowledge argument scheme Aurora [BSCRSVW19] ourselves as the ZKP backend. We use the extension field of a Mersenne prime $\mathbb{F}_{p^2}$, where $p = 2^{61} - 1$. This is the same as the field used in [ZXZS].

We download the datasets from the UCI repository datasets[DG17] and train the decision tree models using the sklearn package in Python. Then we use these pre-trained decision trees and the testing datasets in our experiments. The attributes are scaled to 32-bit integers in the field for our ZKP backend. As the attributes are only used for comparisons, the scaling does not affect the prediction and the accuracy of the decision trees.

Hardware.. We run all of the experiments on Amazon EC2 c5n.2xlarge instances with 64GB 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.

Hash function.. As we will show in the experiments, computing hashes in the arithmetic circuits is the bottleneck of our system. For example, it takes 27,000 multiplication gates to compute one SHA-256 hash. Therefore, in order to improve the performance of our system, we use the SWIFFT[LMPR] hash function, which is an algebraic hash function that is collision-resistant and is friendly to arithmetic circuits. With the optimizations proposed in jsnark[Jsna], one SWIFFT hash can be implemented with around 3,000 multiplication gates.

Datasets.. We use three datasets from the UCI machine learning repository [DG17]. The small dataset we use is named Breast-Cancer-Wisconsin(Original). It is used for breast cancer diagnosis. Each data has 10 attributes and the prediction is either 0 or 1. We train the model on a training set of 600 data points. The pre-trained decision tree has 61 nodes and 10 levels. The second dataset we use is Spambase. It is used to recognize the spam emails. Each data has 57 attributes and the prediction is either 1 or 0. We train the model on training set of $4,000$ data points. The pretrained decision tree has in total 441 nodes and 26 levels. The largest dataset is Forest Covertype. It is used to predict forest cover type from cartographic variables. Each data has 54 attributes and the total number of class is 7. We train the model on a training set of $5,000$ data points. The pre-trained decision tree has in total $1,029$ nodes and 23 levels.

5.5.1 Performance of ZKDT

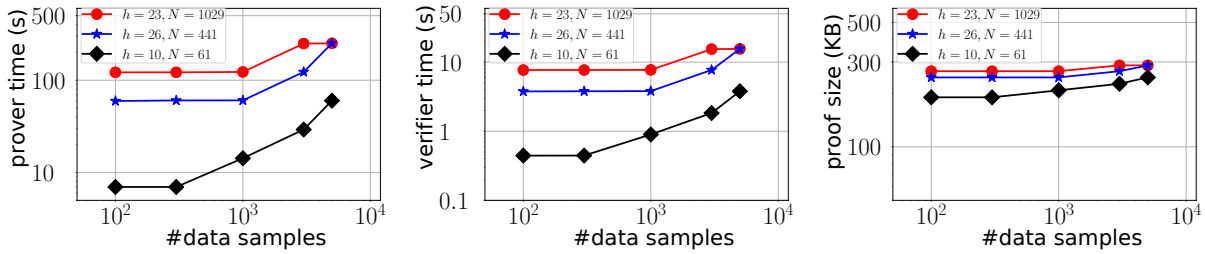
We first present the performance of our zero knowledge decision tree prediction protocol. We vary the length of the prediction paths from 6 to 48. The prediction paths of the first 3 columns are obtained from the decision trees of the three real-world datasets described above, while the last one is obtained from synthetic data. Table 5.1 shows the performance of our scheme.

As we can see in the Table 5.1, the efficiency of our zkDT scheme is reasonable in practice. Though linear to the size of the whole decision trees, the time to commit the decision trees is only on the order of

[1]We actually use the rank-1-constraint-system (R1CS) to be compatible with the backend.

Length h	6	12	24	48
#Attributes d	10	54	57	1000
Commit Time (ms)	0.38	6.3	2.8	13.3
Prover Time (s)	0.754	1.577	3.433	7.024
Verifier Time (s)	0.050	0.104	0.221	0.445
Proof size (KB)	140.736	155.936	172.224	189.632

Table 5.1: Performance of zero knowledge decision tree predictions.



(a) Prover time (b) Verification time (c) Proof size

Figure 5.5: Performance of the zero knowledge decision tree accuracy scheme.

milliseconds. This is because the Commit in our ADT only involves computing hashes, which is very fast in practice. For the prover time and the verification time, it takes 7.02 seconds to generate the proof for a prediction path of length 48, and 0.445 seconds to validate the proof. The proof size is 189KB. Note that we choose our ZKP backend of Aurora [BSCRSVW19] to optimize for the prover time. Our zkDT scheme works on all backends and we could use schemes such as Bulletproof [BBBPWM] and SNARK [PHGR13] to reduce the proof size to several KBs or less, with a sacrifice on the prover time.

Moreover, the prover time and the verification time scale roughly linearly with the length of the prediction paths, while the number of attributes d does not affect the performance by much. This is because the bottleneck of the efficiency is computing the hashes in the circuit of the ZKP backend for the path validation, and the number of hashes is linear to the length of the path. Besides, the performance of our scheme fully depends on the parameters of the decision trees and the size of the data samples, but not on the values. Hence, we do not observe any major difference on the performance between the real datasets and the synthetic dataset.

5.5.2 Other Variants of Zero Knowledge Decision Trees.

Multivariate decision trees. . In univariate decision trees, each decision node checks only one attribute for axis-aligned splits. In a linear multivariate decision tree, each decision node divides the input space into two with an arbitrary hyperplane leading to slanting splits. More formally, In a linear multivariate decision tree $\mathbf{T}$, each internal node v has a size-d vector $v.\mathbf{w}$ of weights for d attributes, a threshold $v.\mathsf{thr}$ and two children $v.\mathsf{left}$ and $v.\mathsf{right}$. Similar to the univariate decision tree, each leaf node u stores the classification result $u.\mathsf{class}$. Each data sample is represented as a size-d vector $\mathbf{a}$ of values corresponding to each attribute. The algorithm of multivariate decision tree prediction is shown in Algorithm 16. It starts from the root of $\mathbf{T}$.

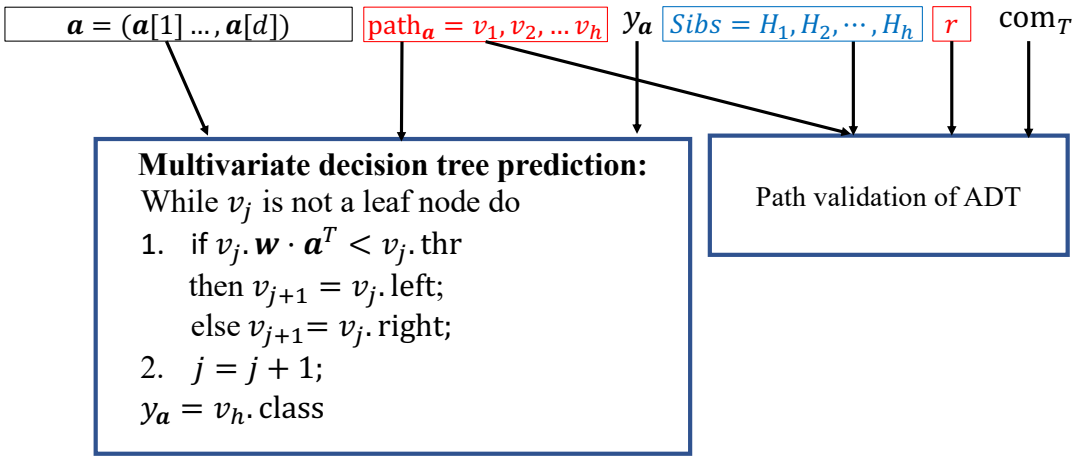


Figure 5.6: Zero knowledge multivariate decision tree prediction. Public inputs are in black, secret witness is in red, and extended witness for efficiency is in blue.

Algorithm 16 Multivariate Decision Tree Prediction

Input: Decision tree $\mathbf{T}$, data sample $\mathbf{a}$
Output: classification $y_{\mathbf{a}}$

```
1: v := T.root
2: while v is not a leaf node do
3:     if v.w · a^T < v.thr then
4:         v := v.left
5:     else
6:         v := v.right
7:     end if
8: end while
9: return v.class
```

For each node of v in $\mathbf{T}$, it compares $v.\mathbf{w} \cdot \mathbf{a}^T$ with $v.\text{thr}$, and moves to $v.\text{left}$ if $v.\mathbf{w} \cdot \mathbf{a}^T < v.\text{thr}$, and $v.\text{right}$ otherwise. Eventually, the algorithm reaches a leaf node u and the result of the prediction is $u.\text{class}$.

Surprisingly, the hyperplane split in each node of multivariate decision trees makes the design of zero knowledge multivariate decision trees simpler. The prediction of multivariate decision trees is not a RAM based program anymore because it uses the linear combination of d attributes in each node. The circuit for zero knowledge multivariate decision tree predictions is given in Figure 5.6. Note that comparing to Figure 5.2, the circuit does not need to perform random access to the attributes using a permutation check.

Regression decision trees.. Decision trees where the target variable can take continuous values (typically real numbers), e.g., the price of a house, are called regression decision trees. For regression decision tree predictions, we retain Algorithm 14 except for changing $y_{\mathbf{a}}$ to a real number. The circuit for verifying the

regression decision tree prediction is also the same as in Figure 5.2 but $y_{\mathbf{a}}$ could be a real number.

Random forests. . Random forest consists of many individual decision trees. It is an ensembling learning method that can help reduce the variance and aviod the overfitting of a single decision tree model. For the classification problem, each individual decision tree in the random forest outputs a class prediction and the class with the most votes becomes the prediction of the random forest. More formally, suppose a random forest contains m decision trees $\mathbf{T}_1, \mathbf{T}_2, \cdots, \mathbf{T}_m$ and $[M]$ is the set of all target classifications, for a data sample $\mathbf{a}$ with d features, we run Algarithm 14 with each decion tree and get the predictions $\mathbf{T}_1(a), \mathbf{T}_2(a), \cdots, \mathbf{T}_m(a) \in [M]$. Then the model outputs the index in $[M]$ with most occurrences in $\mathbf{T}_1(a), \mathbf{T}_2(a), \cdots, \mathbf{T}_m(a)$. The specific algorithm is given in Algorithm 17.

For constructing the circuit to verify the random forest prediction, besides verifying each decision tree's prediction, we add an argmax function on all predictions before outputting the final classification, which is easy to be implemented by the arithmetic circuit.

For the regression problem, random forests report the mean of all values output by individual decision trees, which can be also easily implemented by the arithmetic circuit. We omit the formal construction.

5.5.3 Comparison to Generic Zero Knowledge Proof Schemes

We conducted the experiments on the comparison between our zkDT scheme with the baseline of using RAM-based and circuit-based generic ZKP schemes and present the experimental results in this section.

For the RAM-based ZKP scheme, we write the decision tree predictions algorithm in TinyRAM [BSCGTV], a language similar to assembly language with a simple instruction set. Each iteration in Algorithm 14 takes 20 TinyRAM instructions and there are in total h iterations. We then translate the program to a circuit using the RAM-to-circuit reduction in [BSCTV]. Each instruction takes around 4,000 multiplications gates. Finally, we estimate the prover time using the same ZKP backend of Aurora [BSCRSVW19] on the circuit. For the naive circuit-based ZKP scheme, we hardcode the whole decision tree into the circuit and run the prediction algorithm directly using the arithmetic circuit. Then we apply the same ZKP backend. We also extend the implementation to zero knowledge random forests trained on the same dataset. Table 5.2 shows the performance of random forests consisting of different number of decision trees.

Algorithm 17 Random Forest Prediction

Input: Random forest with $\mathbf{T}_1, \mathbf{T}_2, \cdots, \mathbf{T}_m$, data sample $\mathbf{a}$
Output: classification $y_{\mathbf{a}}$

1: **for** $i = 1$ to M **do**
2: $\quad Occ[i] = 0$
3: **end for**
4: **for** $i = 1$ to m **do**
5: $\quad$ Run Algorithm 14 with input of $\mathbf{T}_i$ and $\mathbf{a}$
6: $\quad Occ[\mathbf{T}_i(\mathbf{a})]$++
7: **end for**
8: **return** $\arg\max_i Occ[i]$

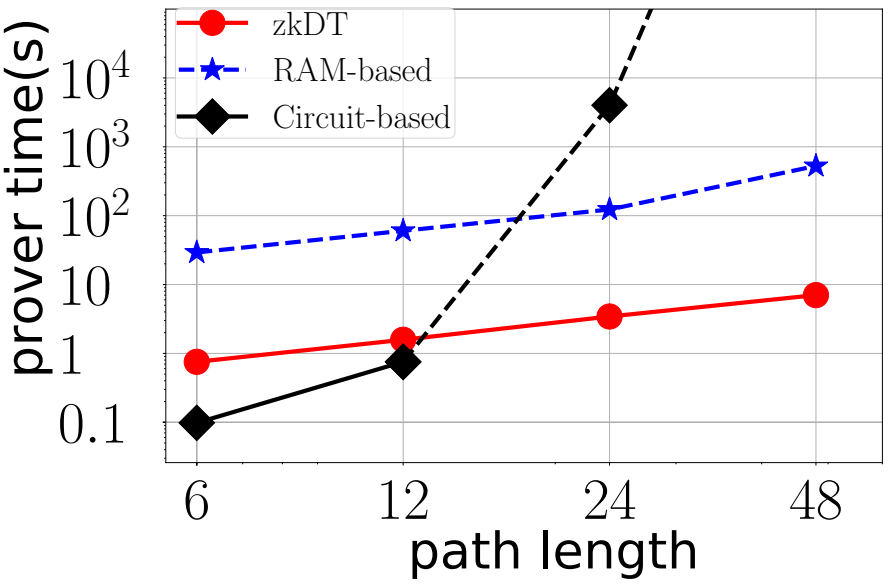


Figure 5.7: Comparison between zkDT, RAM-based and circuit-based generic ZKP schemes.

# of Trees	2	8	32	128
Length h	24	24	12	12
Commit Time (ms)	13.49	54.95	115.14	468.56
Prover Time (s)	6.992	29.317	60.532	253.41
Verifier Time (s)	0.447	1.842	3.83	15.86
Proof size (KB)	189.728	225984	245632	286592

Table 5.2: Performance of zero knowledge random forest predictions.

5.5.4 Performance of ZKDTA

In this section, we further evaluate the performance of our zero knowledge decision tree accuracy scheme. We implement Protocol 14 and test it on decision trees trained on the three datasets described before. We vary the number of data samples in the testing dataset as 100, 300, 1000, 3000 and 5,000. We present the prover time, proof size and verification time in Figure 5.5.

As shown in Figure 5.5, our zkDTA scheme achieves very good efficiency in practice. On the largest instance with a decision tree of 1029 nodes and 23 levels, it only takes 250 seconds to generate a zero knowledge proof for its accuracy on a large testing dataset of 5,000 data samples. We believe this overhead is close to practical in many applications. The verification time is around 15.6 seconds. In this case, the proof size is 287KB, which is actually smaller than the size of the decision tree and the size of the testing dataset. This means our scheme not only provides soundness and zero knowledge, but also reduces the communicating comparing to the naïve solution of posting the model and the dataset. The gap will further increase because of the succinctness of the proof size.

In addition, Figure 5.5 shows that the prover time and the verification time for all three models remain mostly unchanged until the size of the testing dataset becomes larger. This is because the bottleneck of our scheme is computing hashes in the circuit of the ZKP backend. The number of hashes is proportional to the total number of nodes in the decision tree, and does not depend on the size of the testing dataset. For example, for the large decision tree model with $N = 1029$ nodes, when the size of the testing dataset is less than 1000, the sub-circuit checking the commitment using hashes consists of around 2^{21} multiplication gates and contributes to more than 75% of the whole circuit. Thus, the performance of the scheme remains mostly the same for dataset with less than 1000 samples. As the size of the testing dataset becomes larger than 1000, other components of the circuit to check decision tree predictions, multisets and permutations start to impact the performance. There are 2^{23} multiplication gates in our largest data point in the figure.

The observation above also justifies the importance of our two optimizations proposed in Section 5.4. If we simply repeat the zero knowledge predictions multiple times, the number of hashes will increase with the size of the testing dataset. In the largest data point, there are 5,000 samples and each prediction path is of length around 20. The performance would be $100\times$ slower compared to our optimized scheme. Similarly, as shown by the three real-world datasets, the decision trees are usually not balanced. Without our second optimization in Section 5.4, the number of hashes would be 2^h, which is much larger than N. For the largest decision tree with $N = 1029$ and $h = 23$, our optimization improves the prover time and proof size by around $8000\times$.

Finally, the accuracy of the three decision trees (from small to large) are 94.89%, 92% and 64.25% respectively on their largest testing datasets with 5,000 samples. They are exactly the same as the original decision trees trained from the datasets, as our zkDTA scheme does not use any approximations and does not introduce any accuracy loss.

5.5.5 Applications of Our Schemes

Besides ensuring the integrity of decision tree predictions and accuracy in the scenarios motivated in the introduction, our zero knowledge decision tree schemes can also be applied to build a fair and secure trading platform for machine learning models on blokchains. Machine learning models are valuable assets now, and people want to monetize their expertise on machine learning by selling high-quality models. In this scenario, the buyer prefers to test the quality of the machine learning model before making the payment, while the seller does not want to reveal the model first, as the buyer could disappear without any payment after seeing the model. This is the classical problem of fair exchange. One could rely on a trusted party to address this problem. However, it often introduces a heavy burden on the trusted party to validate the quality of the models, enforce the payments and resolve the disputes. Some applications may lack the existence of such a trusted party.

Blockchain is a promising technique to replace the role of the trusted party in this scenario. In a blockchain, all the users validate the data posted on the blocks such that as long as more than 50% of the users are honest, the data on the blockchain is valid and cannot be altered. *Zero knowledge contingent payments* [Zkc; CGGN17] provide a framework for users to trade secret data fairly and securely on blockchains. Instead of posting the data directly on the blockchain, which reveals the data to all users of the blockchain, the seller posts a short zero knowledge proof about the properties of the data. Subsequent protocols enforces the payment and the delivery of the data simultaneously using smart contracts, given that the zero knowledge proof is valid. In order to build a trading platform for machine learning models on blockchains, efficient ZKP protocols for machine learning accuracy are the only missing piece in the framework of zero knowledge contingent

payments. Our zero knowledge decision tree schemes in this paper fill in this gap and can be used to build such a fair and secure trading platform. Realizing the system of the trading platform is left as future work.

Chapter 6

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+20].

6.1 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 [**hamlin2022**; **wintermeyer2021**] 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 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; XZZPS19a; Zha+20; BSBHR19; BSCTV; AHIV17; BSCRSVW19; COS19; CHMMVW20; ZGKPP17c; ZGKPP18; BBBPWM; GWC19; Set20b]. 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 [Cir]. 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 [ZXZS]. 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 [Gro16b], 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

[1] see the ranking at `https://rekt.news/leaderboard`

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 6.5. 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.

6.1.1 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.

6.2 Background

In this section we cover the preliminaries, essential background on blockchains, and zero-knowledge proofs.

6.2.1 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:

- $\mathsf{rt} \leftarrow \mathsf{MT.Commit}(\mathbf{x})$
- $(\mathbf{x}[i], \pi_i) \leftarrow \mathsf{MT.Open}(\mathbf{x}, i)$
- $\{1, 0\} \leftarrow \mathsf{MT.Verify}(\pi_i, \mathbf{x}[i], \mathsf{rt})$.

6.2.2 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.

6.2.3 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

$$\mathsf{LightCC}(\mathsf{LCS}_{r-1}, \mathsf{blkH}_{r-1}, \mathsf{blkH}_r) \rightarrow \{\mathsf{true}, \mathsf{false}\}$$

to denote the block validation rule of a light client: given a new block header blkH_r, 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 6.2.1 (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.*

6.2.4 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 6.2.2. *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 $\pi \leftarrow \mathcal{P}(\mathbf{x}, \mathbf{w}, \mathsf{pp})$,*

$$\Pr[\mathcal{V}(\mathbf{x}, \pi, \mathsf{pp}) = 1] = 1$$

- ***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*

$$\Pr[(\mathbf{x}; \mathbf{w}) \notin \mathcal{R} \wedge \mathcal{V}(\mathbf{x}, \pi^*, \mathsf{pp}) = 1] \leq \mathsf{negl}(\lambda),$$

  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.*

6.3 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.

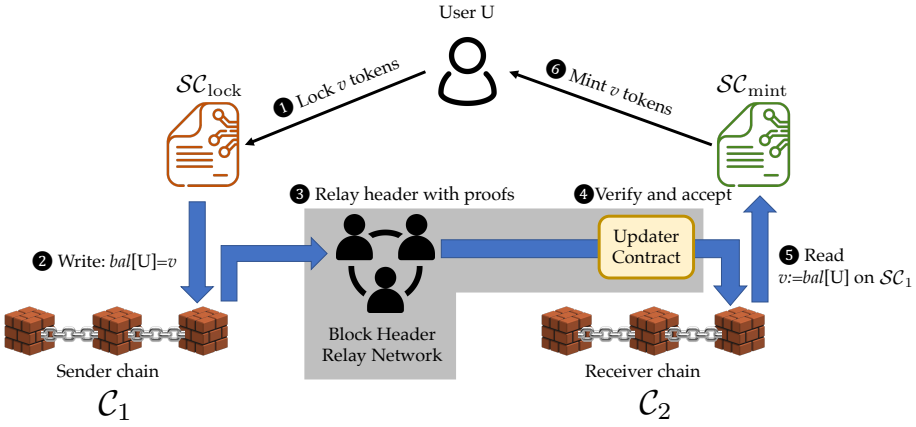


Figure 6.1: The design of zkBridge illustrated with the example of cross-chain token transfer. The components in shade belongs to zkBridge. For clarity we only show one direction of the bridge and the opposite direction is symmetric.

6.3.1 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 6.1 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

[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.

relay nodes; it exposes a simple and application-agnostic API, from which application smart 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. 6.1 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 \$v tokens in $\mathcal{SC}_{\text{lock}}$ (step ① in Fig. 6.1) and then requests \$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 \$v (step ⑤), and only then mint \$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 6.3.3.

6.3.2 Protocol detail

Having presented the overview, in this section, we specify the protocol in more detail.

6.3.2.1 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 6.2), 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.

6.3.2.2 Construction of zkBridge

As described in Section 6.3, a bridge $\mathcal{BR}[\mathcal{SC}_1 \rightarrow \mathcal{SC}_2]$ 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 18.

Protocol 18 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 blkH_{r-1}, namely blkH_r.

Generate a ZKP π proving

$$\mathsf{LightCC}(\mathsf{LCS}_{r-1}, \mathsf{blkH}_{r-1}, \mathsf{blkH}_r) \rightarrow \mathsf{true}.$$

Send $(\pi, \mathsf{blkH}_r, \mathsf{blkH}_{r-1})$ to the updater contract.

end procedure

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 6.4.

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 [FS].

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 19.

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.

Protocol 19 The updater contract

```
headerDAG := ∅                                              // DAG of headers
LCS := ⊥                                                    // light client state
procedure HeaderUpdate(π, blkH_r, blkH_{r−1})
    if blkH_{r−1} ∉ headerDAG then
        return False                                        // skip if parent block is not in the DAG
    end if
    if π verifies against LCS, blkH_{r−1}, blkH_r then
        Update LCS according to the light client protocol.
        Insert blkH_r into headerDAG.
    end if
end procedure
procedure GetHeader(t)                                      // t is a unique identifier to a block header
    if t ∉ headerDAG then
        return ⊥                                            // tell the caller to wait
    else
        return headerDAG[t], LCS                            // The LCS will help users to determine if t is on a fork.
    end if
end procedure
```

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 information 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 6.3.1. *The bridge $\mathcal{BR}[\mathcal{SC}_1 \to \mathcal{SC}_2]$ implemented by protocols 18 and 19 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. 6.2.1; 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 $Longest : \mathsf{DAG} \rightarrow \mathsf{List}$ 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 $\mathsf{MainChain} := \mathsf{L}[: -K]$. 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. ∎

6.3.3 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 6.3.1. 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.

6.3.4 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 6.4 and 6.5. 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 [ZXZS] 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 6.4.

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 6.5.

6.4 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 [ZXZS; XZZPS19a; Set20b; WTSTW18; BSCTV; BSCRSVW19; AHIV17; BSBHR19; Zha+20; GWC19; COS19] 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 6.4.1, the detail of the distributed sumcheck protocol in Section 6.4.2 and the detail of the distributed PC protocol in Section 6.4.3. We combine them all together to build deVirgo in Section 6.4.4.

6.4.1 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:

$$\tilde{V}(\mathbf{x}) = \sum_{\mathbf{b}\in\{0,1\}^\ell} \prod_{i=1}^{\ell} ((1-x_i)(1-b_i) + x_i b_i)) \cdot V(\mathbf{b}),$$

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 $\tilde{\beta}$ is the multilinear extension of β. Then $\tilde{\beta}$ can be expressed as: $\tilde{\beta}(\mathbf{x},\mathbf{y}) = \prod_{i=1}^{\ell}((1-x_i)(1-y_i) + x_i y_i)$.

6.4.2 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

Protocol 17 (**Sumcheck**). The protocol proceeds in ℓ rounds.

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

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

  $\mathcal{V}$ 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

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

  $\mathcal{V}$ 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

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

  $\mathcal{V}$ 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.

the verifier $\mathcal{V}$ that the summation is H via a sequence of interactions, and the formal protocol is presented in Protocol 24.

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; ZGKPP17c; XZZPS19a; ZXZS]. 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 $\mathbf{b} \in \{0,1\}^{s_i}$ 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 $add_i, mult_i : \{0,1\}^{s_{i-1}+2s_i} \to \{0,1\}$, 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:

$$V_i(\mathbf{z}) = \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} (add_{i+1}(\mathbf{z},\mathbf{x},\mathbf{y})(V_{i+1}(\mathbf{x}) + V_{i+1}(\mathbf{y})) \\ + mult_{i+1}(\mathbf{z},\mathbf{x},\mathbf{y})V_{i+1}(\mathbf{x})V_{i+1}(\mathbf{y})) \tag{6.1}$$

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:

$$\begin{aligned}\tilde{V}_i(\mathbf{g}) =& \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} h_i(\mathbf{g},\mathbf{x},\mathbf{y}) \\ =& \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} (\tilde{add}_{i+1}(\mathbf{g},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &+ \tilde{mult}_{i+1}(\mathbf{g},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}))\,,\end{aligned} \tag{6.2}$$

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

With Equation 6.2, 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 6.2 with $i = 0$. As described in Protocol 24, 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 6.2, this random linear combination can be written as

$$\begin{aligned}&\alpha_{i,1}\tilde{V}_i(\mathbf{u}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v}) \\ =&\alpha_{i,1} \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &+\tilde{mult}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}) \\ &+\alpha_{i,2} \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y})(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &+\tilde{mult}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}) \\ =& \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} (\alpha_{i,1}\tilde{add}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &+ (\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u},\mathbf{x},\mathbf{y}) + \alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v},\mathbf{x},\mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y})\end{aligned} \tag{6.3}$$

$\mathcal{V}$ and $\mathcal{P}$ then execute the sumcheck protocol on Equation 6.3 instead of Equation 6.2. 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 18.

Protocol 18 (**GKR**). 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 $C(\mathbf{x}) = \mathbf{1}$ 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`.

Complexity of the sumcheck protocol in GKR protocol. For simplicity in the complexity analysis, we

define the sumcheck equation in GKR protocol as

$$\tilde{V}_i(\mathbf{g}) = \sum_{\mathbf{x}\in\{0,1\}^\ell} f(\mathbf{x}, \tilde{V}_{i+1}(\mathbf{x})), \tag{6.4}$$

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 6.4, the prover time in Protocol 24 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 6.4.

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 6.4 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 24, 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 [XZZPS19a] 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 19.

1. From round 1 to round $(\ell - n)$ (step 1 in Protocol 19), $\mathcal{P}_i$ 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.

Protocol 19 (**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 $1 \leq j \leq \ell - n$, each $\mathcal{P}_i$ sends P_0 a univariate polynomial

$$f_j^{(i)}(x_j) \stackrel{def}{=} \sum_{\mathbf{b} \in \{0,1\}^{\ell-n-j}} f^{(i)}(\mathbf{r}[1:j-1], x_j, \mathbf{b}),$$

   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 $0 \leq i < N$.

3. In the j-th round, where $\ell - n < j \leq \ell$, $\mathcal{P}_0$ and $\mathcal{V}$ run Protocol 24 on the statement:

$$H' = \sum_{\mathbf{b} \in \{0,1\}^n} f'(\mathbf{b}),$$

   where $H' = \sum_{i=0}^{N-1} f^{(i)}(\mathbf{r}[1:\ell-n])$.

2. In round $(\ell - n)$ (step 2 in Protocol 19), 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 19), $\mathcal{P}_0$ 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 machines, which is optimal for distributed algorithms both in asymptotic complexity and in practice. We give the complexity of Protocol 19 in the following.

[3]The approach can extend to the product of two multi-linear polynomials, which matches the case in Virgo.

Complexity of the distributed sumcheck protocol. For the multivariate polynomial of f defined in Equation 6.4, 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 $O(\ell)$.

6.4.3 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.
- $\mathsf{comm}_f \leftarrow \mathsf{PC.Commit}(f, \mathsf{pp})$: 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 $\mathsf{comm}_f = \mathsf{MT.Commit}(\mathbf{f}_\mathbb{L})$.
- $(y, \pi_f) \leftarrow \mathsf{PC.Open}(f, \mathbf{r}, \mathsf{pp})$: 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 $\pi_f = (\mathbf{f}_\mathbb{L}[k_1], \pi_{k_1}, \cdots, \mathbf{f}_\mathbb{L}[k_c], \pi_{k_c})$.[4]
- $\{1, 0\} \leftarrow \mathsf{PC.Verify}(\mathsf{comm}_f, \mathbf{r}, y, \pi_f, \mathsf{pp})$: 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 comm_f by $\mathsf{MT.Verify}$ $(\pi_{k_1}, \mathbf{f}_\mathbb{L}[k_1], \mathsf{comm}_f), \cdots,$ $\mathsf{MT.Verify}$ $(\pi_{k_c}, \mathbf{f}_\mathbb{L}[k_c], \mathsf{comm}_f)$. If all checks pass, the verifier outputs 1, otherwise the verifier outputs 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 $\tilde{\beta}$ function, we have

$$f(\mathbf{r}) = \sum_{i=0}^{N-1} \tilde{\beta}(\mathbf{r}[\ell - n + 1 : \ell], \mathbf{i}) f^{(i)}(\mathbf{r}[1 : \ell - n]). \tag{6.5}$$

[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.

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 PC.Commit to commit $f^{(i)}$ in the beginning of the sumcheck protocol. In the last round, $\mathcal{P}_i$ runs 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 PC.Verify to validate N polynomial commitments separately. Then $\mathcal{V}$ computes $\tilde{\beta}(\mathbf{r}[\ell-n+1:\ell],\mathbf{i})$ for each i. Finally, $\mathcal{V}$ checks $f(\mathbf{r})=\sum_{i=0}^{N-1}\tilde{\beta}(\mathbf{r}[\ell-n+1:\ell],\mathbf{i})f^{(i)}(\mathbf{r}[1:\ell-n])$.

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 20.

Protocol 20 (**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 $f(\mathbf{x})=\tilde{\beta}(\mathbf{x}[\ell-n+1:\ell],\mathbf{i})f^{(i)}(x[1:\ell-n])$. 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 $\mathsf{comm}_{h^{(i)}}=\mathsf{MT.Commit}(\mathbf{h}_{\mathbb{L}}^{(i)})$ and sends $\mathsf{comm}_{h^{(i)}}$ to $\mathcal{P}_0$.
4. Suppose $\mathbf{h}=(\mathsf{comm}_{h^{(0)}},\cdots,\mathsf{comm}_{h^{(N-1)}})$, $\mathcal{P}_0$ computes $\mathsf{comm}=\mathsf{MT.Commit}(\mathbf{h})$ and sends comm 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 $f(\mathbf{r})=\sum_{i=0}^{N-1}\tilde{\beta}(\mathbf{r}[\ell-n+1:\ell],\mathbf{i})f^{(i)}(\mathbf{r}[1:\ell-n])$. $\mathcal{V}$ 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 $\mathsf{MT.Verify}(\pi_{k_1},\mathbf{h}[k_1],\mathsf{comm})$, $\cdots$, $\mathsf{MT.Verify}(\pi_{k_c},\mathbf{h}[k_c],,\mathsf{comm})$. $\mathcal{V}$ 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 PC.Verify. If all checks pass, $\mathcal{V}$ outputs 1, otherwise $\mathcal{V}$ outputs 0.

The idea of our scheme is that each $\mathcal{P}_i$ exchanges data with other machines immediately after computing

[6]In Virgo, $\rho=32$ for security requirements. Our scheme can extend to $\rho=32$ easily.

$\mathbf{f}_{\mathbb{L}}^{(i)}$ instead of invoking 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 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 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 MT.Commit on $com_{h^{(0)}}, \cdots, com_{h^{(N-1)}}$ to compute the aggregated commitment, comm, and $\mathcal{P}_0$ sends comm to $\mathcal{V}$ (step 4). In the 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 $(\mathsf{comm}_{h^{(k_j-1)}}, \pi_{k_j})$ = MT.Open (comm, k_j), and sends these messages to $\mathcal{V}$ (step 5 and step 6). $\mathcal{V}$ can validate N evaluations by invoking 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 20 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 $O(\lambda(N+\ell^2))$.

6.4.4 Combining everything together

In this section, we combine the distributed sumcheck and the distributed PC altogether to build deVirgo.

Background: The Virgo protocol. By combining the GKR protocol and the polynomial commitment in Section 6.4.3 We present the formal protocol of Virgo in Protocol 21 and the the complexity of Protocol 21 in the following[7].

Protocol 21 (**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 $\mathsf{comm}_{\tilde{V}_d}$ and sends $\mathsf{comm}_{\tilde{V}_d}$ to $\mathcal{V}$.
2. $\mathcal{P}$ and $\mathcal{V}$ run step 1-3 in Protocol 18.
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 PC.Open and PC.Verify on $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ with $\mathsf{comm}_{\tilde{V}_d}$ 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 1, otherwise $\mathcal{V}$ outputs 0.

[7] Protocol 21 is a knowledge argument system rather than a zero-knowledge proof protocol as we actually use the knowledge argument system in our construction.

Complexity of Virgo [ZXZS]. Given a layered arithmetic circuit C with d layers and m inputs, Protocol 21 is a zero-knowledge proof protocol as defined in Definition 6.2.2 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 $O(d \log |C| + \lambda \log^2 m)$.

deVirgo. For a data-parallel layered arithmetic circuit C with N copies and d layers, following the workflow of Virgo in Protocol 21, 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 22. And we have the theorem as follows.

Theorem 6.4.1. *Protocol 22 is an argument of knowledge satisfying the completeness and knowledge soundness in Definition 6.2.2 for the relation $C(\mathbf{x}, \mathbf{w}) = \mathbf{1}$, where C consists of N identical copies of $C_0, \cdots, C_{N-1}$.*

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 6.5. 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 20, 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 [ZXZS; 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 22 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 22 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 22 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 22 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 $O(d \log |C| + \lambda(N + \log^2 m))$.

Protocol 22 (**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 20 on $\tilde{V}_d$ to get $\mathsf{comm}_{\tilde{V}_d}$ and sends $\mathsf{comm}_{\tilde{V}_d}$ 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 19, 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 19, the distributed sumcheck protocol, on

$$\alpha_{i,1}\tilde{V}_i(\mathbf{u}^{(i)}) + \alpha_{i,2}\tilde{V}_i(\mathbf{v}^{(i)}) =$$
$$\sum_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} ((\alpha_{i,1}\tilde{add}_{i+1}\mathbf{u}^{(i)}, \mathbf{x}, \mathbf{y}) + \alpha_{i,2}\tilde{add}_{i+1}(\mathbf{v}^{(i)}, \mathbf{x}, \mathbf{y}))(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y}))$$
$$+(\alpha_{i,1}\tilde{mult}_{i+1}(\mathbf{u}^{(i)}, \mathbf{x}, \mathbf{y}) + \alpha_{i,2}\tilde{mult}_{i+1}(\mathbf{v}^{(i)}, \mathbf{x}, \mathbf{y}))\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}))$$

   - 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.

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 20 to open $\tilde{V}_d(\mathbf{u}^{(d)})$ and $\tilde{V}_d(\mathbf{v}^{(d)})$ while $\mathcal{V}$ invokes step 7 in Protocol 20 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.

6.5 Reducing proof size and verifier time

Although 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 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 6.2.2, 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 [COS19].

# 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 6.1: 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 6.1. 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 6.1, 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.

6.6 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 6.2.1. 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.

6.6.1 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.

6.6.1.1 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 super-majority 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 6.4) and recursive verification (Section 6.5) 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.

6.6.1.2 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 6.6.2. 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.

6.6.2 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.

6.6.2.1 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 6.2: Evaluation results. RV is the shorthand for recursive verification.

6.6.2.2 Proof generation time of deVirgo.

We first evaluate the main cryptographic building block—deVirgo—and compare its performance with the original Virgo [ZXZS]. 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.

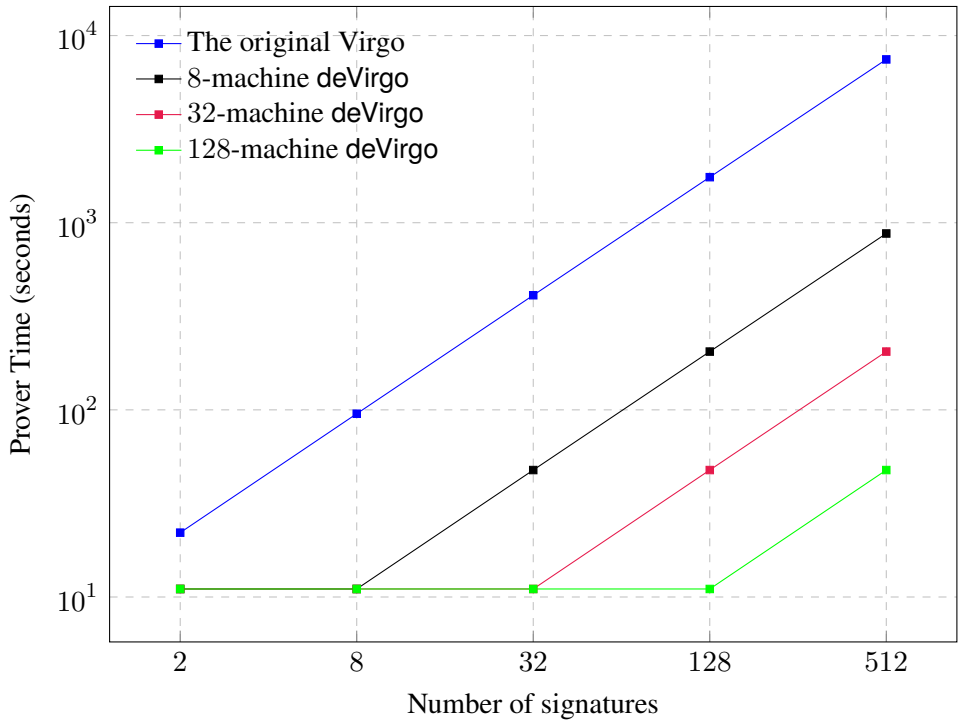


Figure 6.2: Prover time of deVirgo and the original Virgo for Cosmos block header verification.

Figure 6.2 shows the prover time (in seconds) against different N. For deVirgo, we repeat the experiment with 8, 32, 128 distributed machines. According to Fig. 6.2, 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 6.2 reports the communication cost among parallel machines. The total communication cost is linear in the number of machines, consistent with the analysis in Section 6.4.4, 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 6.4, deVirgo accelerates the proof generation on data-parallel circuits with N copies by a factor of almost N, which is optimal for distributed algorithms.

6.6.2.3 Proof size and verification time.

To reduce on-chain verification cost, we use the recursive verification technique presented in Section 6.5. 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 recursive proofs (the column marked RV), and the sum, in Table 6.2, 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 6.2 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 6.2, 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).

6.6.2.4 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.

6.6.3 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 6.3 reports the configuration and each machine costs around $4.5k. The total setup cost is thus around $\$4.5\text{k} \times 32 = \$144\text{k}$. 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 consumption	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 6.3: 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 6.6.1.2, we can further reduce the on-chain cost and offload the cost to users if the number of users is large.

6.6.4 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 6.3 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 6.6.3, since the updater contract only verifies Groth16 proofs in all cases.

6.7 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.

Chapter 7

Polynomial Commitment with a One-to-Many Prover and Applications

Verifiable Secret Sharing (VSS) is a foundational cryptographic primitive that serves as an essential building block in multi-party computation and decentralized blockchain applications. One of the most practical ways to construct VSS is through a polynomial commitment, where the dealer commits to a random polynomial whose 0-th coefficient encodes the secret to be shared, and proves the evaluation of the committed polynomial at a different point to each of N verifiers, i.e., the polynomial commitment is used in a "one-to-many" fashion.

The recent work of Tomescu et al. (IEEE S&P 2020) was the first to consider polynomial commitment with "one-to-many prover batching", such that the prover can prove evaluations at N different points at the cost of $\widetilde{O}(1)$ proofs. However, their scheme is not optimal and requires a trusted setup.

In this paper, we asymptotically improve polynomial commitment with one-to-many prover batching. We propose two novel schemes. First, we propose a scheme with optimal asymptotics in all dimensions in the trusted setup setting. Second, we are the first to consider one-to-many prover batching for *transparent* polynomial commitments, and we propose a transparent scheme whose performance approximately matches the best-known scheme in the trusted setup setting.

We implement our schemes and evaluate their performance. Our scheme in the trusted setup setting improves the proof size by $20\times$ and the verifier time by $7.8\times$ for 2^{21} parties, with a small overhead on the prover time. Our transparent polynomial commitment removes the trusted setup and further improves the prover time by $2.3\times$.

This work was previously published in [ZXHSZ22].

7.1 Introduction

In an $(N, t+1)$ Verifiable Secret Sharing (VSS) protocol [CGMA85; BOGW88; CCD88; RBO89; KZG], roughly speaking, there is a *dealer* and N receivers. The dealer has a secret s, and it wants to split s into N shares, and gives out one share to each receiver. The secret s can be reconstructed if at least $t+1$ receivers combine their shares. However, any coalition of t or fewer receivers cannot learn any information about s (assuming that the dealer is honest). The scheme is "verifiable" if honest receivers can reliably detect a cheating dealer who deals internally inconsistent shares to different receivers. VSS is a foundational building block and widely used in multi-party computation [BOGW88; CCD88; RBO89], threshold cryptosystems [RBO89; Tom+20], and distributed key generation (DKG) [Tom+20; KG09; Kat10]. Recently, VSS has received increasing attention since decentralized blockchains provide a large-scale playground for threshold cryptosystems [Tom+20; GKMPS20].

Several recent works [KZG; Tom+20] showed that *round-efficient* VSS can be constructed from *polynomial commitment* schemes — this is one of the most practical approaches for constructing VSS. In a polynomial commitment scheme, a dealer (also called a prover) can produce a commitment c of a polynomial f whose coefficients are assumed to be in some finite field. Later, during an opening phase, the dealer can claim that the committed polynomial evaluates to y at a given point x, and it can prove to a verifier that this is indeed the correct evaluation result by producing an ideally succinct proof π. Given a polynomial commitment scheme, it is relatively straightforward to construct a VSS scheme [KZG; Tom+20]. Specifically, the dealer chooses a random degree-t polynomial f whose 0-th coefficient encodes the secret s. The dealer now commits to the polynomial and broadcasts the commitment c to all receivers. Next, it chooses N distinct points $x_1, x_2, \ldots, x_N$, and gives $y_i = f(x_i)$ to receiver $i \in [N]$ respectively, and proves to the receiver that the purported outcome y_i is correct with respect to the commitment c. If the dealer is honest, then optimistically the protocol can end here. If the dealer is dishonest and deals incorrect shares to many receivers, the receivers can resort to some complaint mechanism to disqualify the dealer (assuming a synchronous network).

To construct VSS from polynomial commitments, the underlying polynomial commitment scheme is used in a *one-to-many* fashion, i.e., for the same committed polynomial, the dealer needs to prove N evaluations on different points to N different receivers. To produce these N proofs, a naïve approach is for the prover to repeat N times the proving algorithm of the underlying polynomial commitment scheme, thus incurring N times the computational overhead. The recent work of Tomescu et al. [Tom+20] showed an elegant *one-to-many prover batching* technique for the well-known KZG polynomial commitment [KZG], such that the dealer can produce N proofs with only $\widetilde{O}(1)$ slowdown (relative to computing a single proof). Tomescu et al.'s work, however, does not achieve prove batching directly for the KZG scheme, but rather, a more involved variant of KZG. It breaks down the proof generation of the KZG scheme into $\log N$ steps and constructs an authenticated multipoint evaluation tree (AMT) to store redundant computations and improve the efficiency of multiple proofs. Consequently, the verification time and proof size become a logarithmic factor more costly than the original KZG protocol. Another limitation of Tomescu et al.'s work is that it relies on a trusted setup. If the trusted setup is compromised, then the soundness of the scheme can be broken. In decentralized blockchain applications, such a trusted setup is undesirable.

In this paper, we revisit the interesting direction suggested by Tomescu et al. [Tom+20]. We ask the following two questions:

1. Can we achieve one-to-many prover batching directly for the KZG polynomial commitment? If so, can we preserve the optimal verification time and proof size of KZG, while computing N proofs for the cost of one (or for the cost of $\widetilde{O}(1)$ proofs)?

2. Can we approximately match the asymptotic overhead of Tomescu et al. [Tom+20] in all dimensions, but remove the trusted setup?

7.1.1 Our Results and Contributions

We answer these questions with two novel constructions of "polynomial commitment with one-to-many prover batching".

- **Prover batching for the KZG polynomial commitment.** The first contribution is a new algorithm for computing N KZG proofs for the same committed polynomial, paying the cost of only $\widetilde{O}(1)$ proofs. Since our algorithm does not modify the underlying KZG polynomial commitment, we inherit the constant verification time and constant proof size of KZG. Our scheme achieves asymptotic optimality in all dimensions: the proof size and verification time are optimal; the prover time for generating N proofs is $O(N \log N)$ which is also optimal, since simply evaluating the polynomial at N different points would incur $N \log N$ time using the Fast Fourier Transformation (FFT), assuming that $t = \Theta(N)$. Therefore, our scheme also subsumes the results of the original Kate et al.'s paper [KZG] and Tomescu et al.'s paper [Tom+20][1].
- **Transparent polynomial commitment with prover batching.** Our second contribution is a transparent polynomial commitment scheme where a dealer can produce N proofs in $O(N \log N)$ time, and the verification time and proof size are both $O(\log^2 N)$. Here, the prover time is optimal for the same reason as mentioned earlier. Both the proof size and the verification time are succinct and only a logarithmic factor worse than Tomescu et al. [Tom+20].
- **Implementation and evaluation.** We fully implemented both our schemes and evaluated their performance. We then used our new "polynomial commitment schemes with prover batching" to implement VSS and DKG protocols. We compared the efficiency of the resulting schemes with prior work in the same setting. With $N = 2^{21}$ parties, our KZG-based polynomial commitment and the corresponding VSS scheme reduced the proof size of the AMT scheme [Tom+20] by $20\times$, reduced the verifier time by $7.8\times$, while introducing a small overhead of $3\times$ on the prover time. These led to $3.3\times$ better computation time and $20\times$ smaller communication in the DKG scheme. Our transparent scheme not only removes the trusted setup but also improves the prover time by an order of magnitude. However, it does introduce a large proof size. Our code is open source (the code is available at `https://github.com/sunblaze-ucb/eVSS`).
- **Techniques: "one-to-many zero-knowledge proof".** To construct our transparent polynomial commitment with prover batching, we come up with a more general technique which can be of independent interest and lead to other interesting applications. Basically, consider a circuit C with N outputs, wherein the prover wants to prove one output to each verifier respectively. We give a "one-to-many zero-knowledge proof" construction where the prover's computation is only $\widetilde{O}(|C|)$ where $|C|$ denotes the size of C, whereas a straightforward application of existing techniques where the prover produces a separate proof for each verifier would have incurred at least $N \cdot |C|$ prover time.

Table 7.2 shows how our "polynomial commitment with one-to-many prover batching" compares with prior schemes. Given such a polynomial commitment scheme, one can directly construct (synchronous) VSS

[1] After submitting our paper, we found that the same algorithm was also proposed independently by Dankrad Feist and Dmitry Khovratovich at `https://github.com/khovratovich/Kate`. We thank Alin Tomescu for pointing it out.

Table 7.1: Polynomial commitment with a one-to-many prover: comparison with prior works. We assume $t = \Theta(N)$.

Scheme	Trans.	$\mathcal{P}$ time	$\mathcal{V}$ time	Proof size
KZG [KZG]	no	$O(N^2)$	$O(1)$	$O(1)$
AMT [Tom+20]	no	$O(N \log N)$	$O(\log N)$	$O(\log N)$
hbACSS [YLFKM21]	yes	$O(N^2)$	$O(N)$	$O(\log N)$

≪TomR 7.1.1: ≫ • **Trans.** means without a trusted setup. $\mathcal{P}$ **time** represents the dealer time for producing *all* N *proofs*. $\mathcal{V}$ **time** represents the verification time per verifier.

Table 7.2: Polynomial commitment with a one-to-many prover: comparison with prior works. We assume $t = \Theta(N)$.

Scheme	Trans.	$\mathcal{P}$ time	$\mathcal{V}$ time	Proof size
KZG [KZG]	no	$O(N^2)$	$O(1)$	$O(1)$
AMT [Tom+20]	no	$O(N \log N)$	$O(\log N)$	$O(\log N)$
hbACSS [YLFKM21]	yes	$O(N^2)$	$O(N)$	$O(\log N)$
Our KZG-based	no	$O(N \log N)$	$O(1)$	$O(1)$
Our Transparent	yes	$O(N \log N)$	$O(\log^2 N)$	$O(\log^2 N)$

≪TomR 7.1.2: ≫ • **Trans.** means without a trusted setup. $\mathcal{P}$ **time** represents the dealer time for producing *all* N *proofs*. $\mathcal{V}$ **time** represents the verification time per verifier.

and DKG using existing techniques described by Tomescu et al. [Tom+20]. Table 7.3 shows the asymptotic overhead of the resulting VSS and DKG schemes and how our work improves over prior work.

7.1.2 Technical Highlights

7.1.2.1 Transparent Polynomial Commitment with Prover Batching

A naïve idea is to commit to the coefficients of the polynomial f using a vector commitment, resulting in a concise commitment c. Now, the dealer can use a non-interactive zero-knowledge proof system to produce a proof that vouches for the evaluation at a specific point. Since the dealer needs to produce a proof for each of the N verifiers, the naïve approach is to repeat the zero-knowledge proof N times — however, this approach would result in at least N^2 prover time for producing all N proofs (since evaluating the polynomial at each point requires at least N amount of computation).

Note that if the dealer only had to evaluate the polynomial f at N different points without having to produce proofs, this could be accomplished through the Fast Fourier Transform (FFT) in $O(N \log N)$ time. The intriguing question is whether we can evaluate the polynomial at all N points and produce all N proofs in $O(N \log N)$ time as well.

A more general problem: one-to-many zero-knowledge proof. To answer this question, we in fact turn our attention to a more general problem. Suppose that there is some circuit C with N different outputs. There are N verifiers, and each of them cares about receiving and verifying one of the outputs of C. Can the prover

produce all N proofs in time $\widetilde{O}(|C|)$ where $|C|$ denotes the size of the circuit[2]? Note that in comparison, the naïve approach of repeating the ZKP independently N times would result in at least $N \cdot |C|$ prover time.

Table 7.3: Comparison of our schemes and prior works in VSS and DKG settings. We assume $t = \Theta(N)$.

Scheme	Trans.	Broad-cast	Optimistic case			Worst-case‡		
			$\mathcal{P}$ time	$\mathcal{V}$ time	Communication	$\mathcal{P}$ time	$\mathcal{V}$ time	Communication
Feldman-VSS [Fel87]	yes	$O(N)$	$O(N \log N)$	$O(N)$	$O(N)$	$O(N \log N)$	$O(N^2)$	$O(N)$
eVSS [KZG]	no	$O(1)$	$O(N^2)$	$O(1)$	$O(1)$	$O(N^2)$	$O(N)$	$O(N)$
AMT-VSS [Tom+20]	no	$O(1)$	$O(N \log N)$	$O(\log N)$	$O(\log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
hbACSS-VSS [YLFKM21]	yes	$O(1)$	$O(N^2)$	$O(N)$	$O(\log N)$	$O(N^2)$	$O(N \log N)$	$O(N \log N)$
Our KZG-based-VSS	no	$O(1)$	$O(N \log N)$	$O(1)$	$O(1)$	$O(N \log N)$	$O(N)$	$O(N)$
Our Transparent-VSS	yes	$O(1)$	$O(N \log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(N \log N)$	$O(N \log^2 N)$	$O(N \log^2 N)$

Table 7.4: VSS schemes

Scheme	Trans.	Broadcast	Optimistic case		Worst-case‡	
			Computation	Communication	Computation	Communication
JF-DKG [GJKR99]	yes	$O(N)$	$O(N^2)$	$O(N)$	$O(N^3)$	$O(N^2)$
eJF-DKG [Kat10]	no	$O(1)$	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N^2)$
AMT-DKG [Tom+20]	no	$O(1)$	$O(N \log N)$	$O(N \log N)$	$O(N^2 \log N)$	$O(N^2 \log N)$
hbACSS-DKG [YLFKM21]	yes	$O(1)$	$O(N^2)$	$O(N \log N)$	$O(N^2 \log N)$	$O(N^2 \log N)$
Our KZG-based-DKG	no	$O(1)$	$O(N \log N)$	$O(N)$	$O(N^2)$	$O(N^2)$
Our Transparent-DKG	yes	$O(1)$	$O(N \log^2 N)$	$O(N \log^2 N)$	$O(N^2 \log^2 N)$	$O(N^2 \log^2 N)$

Table 7.5: DKG schemes (per party overhead)

≪**TomR 7.1.3:** ≫ In this table we only compare VSS/DKG schemes that are in the same synchronous model and incur a constant number of rounds. We discuss other schemes in different settings (e.g., asynchronous, gossip model with $O(\log N)$ rounds) in Section 7.1.3. **Trans.** means without a trusted setup. $\mathcal{P}$ **time** represents the dealer's computation for producing N *proofs* for N receivers. $\mathcal{V}$ **time** represents the verification time per receiver. **Communication** represents the proof size for each receiver in VSS setting and the total communication for each party in DKG setting. ‡ **Worst-case** represents $\Theta(N)$ bad shares, which results in the complaint round.

Brief background on GKR. To achieve this, we will base our scheme on Virgo [ZXZS], which is in turn based on the famous GKR protocol [GKR15]. For simplicity, we will explain our intuition without worrying about zero-knowledge, and therefore we can think of the original GKR protocol. It helps to first consider the interactive version, and then we will describe a new Fiat-Shamir-style transformation to make our interactive protocol non-interactive in the random oracle model.

In the GKR protocol, the prover starts with the output layer (henceforth called the last layer). Proving the output layer boils down to proving a sumcheck statement for a special polynomial that encodes the wiring structure of the output layer of the circuit. This sumcheck would then be reduced to proving two sumchecks for the last but one layer, which can be coalesced into a single sumcheck proof (for the last but one layer) by taking random linear combinations. This goes on recursively layer by layer. If the prover simply ran GKR with each verifier separately, the prover would have to prove a different statement to each verifier at every layer.

Idea 1: using an extra sumcheck protocol to unify statements for all layers. Our idea is to introduce a clever sumcheck protocol after the output layer, such that after the sumcheck protocol with each verifier, the prover would be proving the same statement to all N verifiers for all other layers, *as long as all verifiers use the same random challenges in every round of the protocol*. This way, except for proof component

[2]We use the notation $\widetilde{O}$ to hide polylogarithmic factors.

corresponding to the extra sumcheck, computing the proof components for all other layers is a shared effort among all verifiers. Moreover, we propose a new algorithm for the prover to run all sumchecks with N verifiers in $O(N \log N)$ time. We defer the details of the construction to Section 7.3.

Idea 2: a new Fiat-Shamir-style transformation to make it non-interactive. With the first idea, we could achieve prover batching as long as all verifiers use the same random challenges in every round of the interactive protocol. Our final construction is non-interactive, and to achieve this we describe a new Fiat-Shamir-style transformation such that the prover can emulate the verifiers' challenges non-interactively by making queries to a random oracle.

Note that applying the standard Fiat-Shamir transformation does not work for us since we additionally require that the random challenges are shared among all verifiers in each round. Recall that the standard Fiat-Shamir transformation queries the random oracle on the transcript with the verifier so far. In our case, the transcript with each verifier differs in the output layer. If we simply hashed the entire transcript, it would result in different challenges for different verifiers.

A conceptually simple but somewhat inefficient approach to overcome this discrepancy among verifiers is to use a Merkle tree to hash all verifiers' transcripts and use the root as a unified random challenge among all verifiers. The prover also needs to send the corresponding Merkle branch to each individual verifier, such that a verifier can ascertain that its view of the transcript so far has been incorporated in generating the random challenge. A similar idea was suggested by Yurek et al. [YLFKM21] but for a somewhat different purpose. The drawback with this approach is that it incurs a logarithmic blowup in proof size and verifier time for every round of the protocol.

Our final approach is a hybrid one. We apply the Merkle tree only to the first logarithmically many rounds, i.e., rounds for the extra sumcheck protocol after the output layer. Recall that for every other layer in the circuit, the prover would be proving the same statement to all verifiers, and therefore the transcripts among the verifiers would converge after the extra sumcheck protocol. Thus for all other layers, we can rely on the standard Fiat-Shamir heuristic. This hybrid approach would save us a logarithmic factor in comparison with applying Merkle hash tree to every round.

Proving soundness of our new Fiat-Shamir-style transform. In the most general setting, the standard Fiat-Shamir transformation is known to work only for constant-round interactive proofs if we want the soundness loss in the reduction to be polynomially bounded. In our case, the original interactive protocol is $O(d \cdot \log N)$ rounds where d denotes the depth of the circuit C. Nonetheless, we can still prove the soundness of our new Fiat-Shamir-style transformation with only polynomial loss in soundness in our reduction. To prove this, we suggest a different way to view our transformation. We focus on the perspective of a single verifier and consider a variant (denoted T_{du}) of our original interactive protocol. In T_{du}, we introduce some dummy rounds and dummy messages which correspond to hash computations in the Merkle tree. We then view our final non-interactive proof as applying an alternative heuristic transformation to the modified protocol T_{du}. Using this alternative view, we are able to use techniques from Ben-Sasson et al. [BSCS16] to prove soundness. First, we show that the protocol T_{du} satisfies a stronger notion of soundness called *state restoration soundness*. Given the stronger soundness property, we can show that applying the aforementioned heuristic transformation to T_{du} gives a sound non-interactive protocol in the random oracle model.

Putting everything together. So far, we have described our ideas neglecting the zero knowledge requirement. It is relatively easy to augment the protocol with zero knowledge using techniques proposed in [CFS17; XZZPS19b; ZXZS]. The modifications to the protocol do not fundamentally alter the soundness proof of our Fiat-Shamir-style transformation. We refer to the details of how to achieve zero knowledge in the full version.

Summarizing the above, we now have a non-interactive, one-to-many zero-knowledge proof system with a batched prover. One-to-many polynomial commitment is a special case of this more general problem where the circuit C is an FFT circuit that evaluates the polynomial at N different points. We emphasize that in solving the one-to-many polynomial commitment problem, we actually come up with a "one-to-many zero-knowledge proof" technique that is much more general, and can be of independent interest and will likely lead to broader applications.

7.1.2.2 New Prover Batching

We propose a new prover batching technique for the KZG polynomial commitment. We review the KZG polynomial commitment scheme in Section 7.4.1. Our novel technique is the following. We show that if the dealer needs to open the polynomial f at the points $\omega, \omega^2, \ldots, \omega^N$ where ω is the N-th root of unity, then the N proof terms can be computed efficiently using a constant number of FFT and inverse FFT invocations. Moreover, the FFT computation can be directly applied to the public parameters of the KZG commitment scheme in the base group of a bilinear map, without knowing the trapdoor, as FFT only involves additions and scalar multiplications. Observing this requires some more involved algebraic manipulations which we defer to Section 7.4.2.

7.1.3 Related work

VSS. Chor et al. [CGMA85] were the first to introduce the notion of VSS. Feldman [Fel87] constructed the first efficient Feldman-VSS scheme with homomorphic encryption schemes. Feldman-VSS is computational hiding and information-theoretic binding. The following work of Pedersen [Ped91] presented a counterpart protocol with information-theoretic hiding and computational binding. However, both schemes broadcast $O(N)$ messages during the dealing phase and cost $O(N)$ time for each verifier to check the correctness of the share. Kate et al. [KZG] reduced the broadcast message and the verification time to $O(1)$ in eVSS by the constant-sized KZG polynomial commitment. Their polynomial commitment needed a trusted setup and increased the dealer's computation to $O(N^2)$. Tomescu et al. [Tom+20] achieved a quasi-linear dealing time at the cost of the $O(\log N)$ verification time. The communication for each verifier also increased to $O(\log N)$.

DKG. VSS plays an essential role in constructing DKG protocols. Ingemarsson and Simmons [IS90] first proposed DKG. Pederson [Ped91] improved their scheme for discrete log-based cryptosystems. Gennaro et al. [GJKR99] showed that the secret generated by Pederson's scheme was biased and fixed the problem in their JF-DKG schemes. Neji et al. [NBBR16] debiased the secret by a more efficient method. Moreover, JF-DKG scheme was converted into an adaptively secure DKG by Canetti et al. [CGJKR99]. All DKG protocols mentioned above need $O(N)$ broadcast messages. Later on, Kate's eJF-DKG [Kat10] tamed the broadcasting cost to $O(1)$ on top of eVSS. Tomescu et al. [Tom+20] built AMT-DKG based on their AMT-VSS scheme to achieve a space-time trade-off for eJF-DKG. In this work, our KZG-based-DKG applies our KZG-based-VSS directly to DKG to remove the overhead on time and space in eJF-DKG and AMT-DKG respectively. Recently, Gurkan et al. [GJMMST21] presented an aggregatable VUF-DKG without a trusted setup. It achieves $O(N \log^2 N)$ computation and communication complexity, which is asymptotically the same as our Transparent-DKG. However, the scheme is in the "gossip" model where each party sends messages to her neighbors and has $\log N$ rounds.

Disambiguation. In our experiments, we focus on VSS and DKG in the *synchronous* setting. Earlier works have also shown that polynomial commitment schemes give rise to *asynchronous* VSS and DKG schemes [KG09; KKMS20; YLFKM21; GLLTXZ; DXR]. In the asynchronous setting, the resulting VSS and DKG schemes would also benefit from one-to-many prover batching. An interesting future direction is to apply our prover batching technique and improve VSS and DKG in the asynchronous setting.

7.2 Preliminary

We use $\mathsf{negl}(\cdot) : \mathbb{N} \to \mathbb{R}$ to denote the negligible function, where for each positive polynomial $f(\cdot)$, $\mathsf{negl}(k) < \frac{1}{f(k)}$ for sufficiently large integer k. We use $\mathsf{nonegl}(\cdot) : \mathbb{N} \to \mathbb{R}$ to denote the complement of $\mathsf{negl}(\cdot)$. Let λ denote the security parameter. "PPT" stands for Probabilistic Polynomial Time. We use $f(), h()$ for polynomials, x, y, z for single variable, $\mathbf{x}, \mathbf{y}, \mathbf{z}$ for vectors of variables and $\mathbf{g}, \mathbf{u}, \mathbf{v}$ for vectors of values. x_i denotes the i-th element in $\mathbf{x}$. We use capital letters such as A to represent arrays in algorithms, and $A[i]$ denotes the i-th element in the array. For a multivariate polynomial f, its "variable-degree" is the maximum degree of f in any of its variables. Let $[k]$ denote the set of $\{0, 1, \ldots, k-1\}$.

Let $\mathbb{F}$ be a prime field. We use $\mathbb{w}^0, \ldots, \mathbb{w}^{N-1}$ to denote N roots of unity on $\mathbb{F}$ such that $\mathbb{w}^N = 1$ in $\mathbb{F}$. Let $\mathbb{H} = \{\mathbb{w}^0, \ldots, \mathbb{w}^{N-1}\}$ be a subset of $\mathbb{F}$. We often rely on polynomial arithmetics, which can be efficiently performed via fast Fourier transforms (FFT) and their inverses (IFFT). In particular, polynomial evaluations and interpolations over $\mathbb{H}$ can be performed in $O(N \log N)$ field operations via the standard FFT and IFFT algorithms [CLRS09].

Convolution of two vectors: Let A, B be two arrays of length n. Their convolution, denoted as $C = A * B$, is defined as: $C[j] = \sum_{i=0}^{j} A[i]B[j-i]$, for $j \in [2n]$, assuming the values of vectors A and B are zeros when the index is out of range (i.e., $\geq n$). It is known that the convolution is equivalent to the multiplication of two polynomials, which can be computed efficiently using FFT and inverse FFT. In particular, $C = \mathsf{IFFT}(\mathsf{FFT}(A) \odot \mathsf{FFT}(B))$, where $\odot$ denotes the Hadamard (element-wise) product, and the two FFTs evaluate A and B on $2n$ points.

Merkle Tree. Merkle tree [Mer87] has been widely used for the vector commitment 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 $\mathbf{r} = (r_0, \ldots, r_{N-1})$, it consists of three algorithms:$\mathsf{rt} \leftarrow \mathsf{MT.Commit}(\mathbf{r})$, $(r_i, \mathsf{path}_i) \leftarrow \mathsf{MT.Open}(i, \mathbf{r})$ and $\{1, 0\} \leftarrow \mathsf{MT.Verify}(\mathsf{rt}, i, r_i, \mathsf{path}_i)$.

We require not only the root to be hiding, but also opening $\mathbf{r}$ at the index of i does not leak any information about $\mathbf{r}$ other than r_i, which is treated as the privacy property of Merkle tree. Formally speaking, for any vector $\mathbf{r}$ of size N, any PPT algorithm $\mathcal{A}$, there exists a simulator $\mathcal{S}_{\mathsf{MT}}$ such that $\mathsf{rt} \leftarrow \mathsf{MT.Commit}(\mathbf{r})$, $(r_i, \mathsf{path}_i) \leftarrow \mathsf{MT.Open}(i, \mathbf{r})$, $\mathsf{rt}', \mathsf{path}_i' \leftarrow \mathcal{S}_{\mathsf{MT}}(r_i, N)$,

$$|\Pr[\mathcal{A}(\mathsf{rt}, r_i, \mathsf{path}_i) = 1] - \Pr[\mathcal{A}(\mathsf{rt}', r_i, \mathsf{path}_i') = 1]| \leq \mathsf{negl}(\lambda).$$

The privacy property can be achieved by concatenating a random number on each leaf of Merkle tree when committing.

Interactive proofs. An interactive proof allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ the validity of some statement through several rounds of interaction. We say that an interactive proof is public coin if $\mathcal{V}$'s challenge in each round is independent of $\mathcal{P}$'s messages in the previous rounds. The proof system is interesting when the running time of $\mathcal{V}$ is less than the time of directly computing the function F. We formalize the interactive proofs in the following:

Definition 7.2.1. *Let F be a function. A pair of interactive machines $\langle \mathcal{P}, \mathcal{V} \rangle$ is an interactive proof for $F(x) = y$ with negligible soundness if the following holds:*

- **Completeness.** *For $F(x) = y$ it holds that* $\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x, y) = 1] = 1$.
- **Soundness.** *For $F(x) \neq y$ and any $\mathcal{P}^*$ it holds that* $\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle(x, y) = 1] \leq \mathsf{negl}(\lambda)$.

Verifiable Secret Sharing. An $(N, t+1)$ secret sharing scheme [Sha79; Bla79] allows a dealer to split up a secret s among N verifiers in such a way that only the subset of $t+1$ or more participants can recover the secret s, and the subset of t or fewer participants can not. An $(N, t+1)$ VSS scheme can be instantiated with a polynomial commitment scheme. It consists of two phases: the sharing (Sh) phase and the reconstruction (Rec) phase. In the sharing phase, the dealer $\mathcal{P}$ picks a random polynomial $f(x)$ of degree t such that the secret $s = f(0)$, and then commits to f. Then $\mathcal{P}$ sends the shared secret $s_j = f(u_j)$ and the corresponding proof π_j to the verifier $\mathcal{V}_j$ for all $j \in [N]$, where each u_j is a unique value. $\mathcal{V}_j$ accepts s_j by checking π_j. In the reconstruction phase, each verifier $\mathcal{V}_j$ reveals s_j and π_j to the reconstructor. The reconstructor uses Lagrange interpolation to recover s after receiving $t+1$ valid shares. The formal VSS protocol instantiated with the polynomial commitment is presented in Protocol 23.

Protocol 23. $(N, t+1)$ **verifiable secret sharing scheme**
Suppose $\mathcal{P}$ is the dealer with a secret $s \in \mathbb{F}$ and $\mathcal{V}_0, \ldots, \mathcal{V}_{N-1}$ are N verifiers. Let $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, t)$.

- **Sh phase:**
  1. $\mathcal{P}$ picks $f \in_R \mathbb{F}[X]$ of degree t such that $s = f(0)$, computes $s_j = f(u_j)$ for all $j \in [N]$.
  2. $\mathcal{P}$ runs $\mathsf{com}_f = \mathsf{Commit}(f, \mathsf{pp})$ and broadcasts com_f to all verifiers.
  3. $\mathcal{P}$ runs $(f(u_j), \pi_j) = \mathsf{Open}(f, u_j, \mathsf{pp})$ and sends $(f(u_j), \pi_j)$ to $\mathcal{V}_j$ for all $j \in [N]$.
  4. For each $j \in [N]$, $\mathcal{V}_j$ invokes $b \leftarrow \mathsf{Verify}(\mathsf{com}_f, u_j, f(u_j), \pi_j, \mathsf{pp})$. If $b = 0$, $\mathcal{V}_j$ broadcasts a complaint against the dealer.
  5. If the size of the set S of complaining players is larger than t, the dealer is disqualified. Otherwise, the dealer reveals the correct shares with proofs by broadcasting $\{f(u_j, \pi_j)\}_{j \in S}$. If any one proof does not verify (or dealer did not broadcast), the dealer is disqualified. Otherwise, each V_j now has her correct share $f(u_j)$.
- **Rec phase:** Given com_f and shares $(f(u_j), \pi_i)_{i \in T \subseteq [N]}$ such that $|T| > t$, the reconstructor runs $b_j \leftarrow \mathsf{Verify}(\mathsf{com}_f, u_j, f(u_j), \pi_j, \mathsf{pp})$ for all $j \in T$. If $b_j = 1$ for all $j \in T$, the reconstructor recovers f with $\{f(u_j)\}_{j \in T}$ by Lagrange interpolation and obtains $s = f(0)$.

7.2.1 Polynomial commitment

Univariate polynomial commitment. Let $\mathbb{F}$ be a finite field and f be a polynomial on $\mathbb{F}$ with degree D. A univariate polynomial commitment (PC) for $f \in \mathbb{F}^D[X]$ and $a \in \mathbb{F}$ consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D)$: Given the security parameter and a bound on the degree of the polynomial, the algorithm generates public parameter pp.
- $\mathsf{com}_f \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$: Given a polynomial $f(x) = \sum_{i=0}^{D} c_i x^i$, the prover commits f with the private randomness r_f and the public parameter pp. r_f can be none.
- $(y, \pi) \leftarrow \mathsf{Open}(f, r_f, a, \mathsf{pp})$: For an evaluation point a, the prover computes $y = f(a)$ and the proof π.
- $\{1, 0\} \leftarrow \mathsf{Verify}(\mathsf{com}_f, a, y, \pi, \mathsf{pp})$. Given the commitment com_f, the evaluation point a, the answer y and the proof π, the verifier checks the correctness of the evaluation.

Definition 7.2.2. *A* PC *scheme satisfies the following properties:*

- ***Completeness.*** *For any polynomial $f \in \mathbb{F}^D[X]$ and $a \in \mathbb{F}$, the following probability is 1.*

$$\Pr\left[\begin{array}{rl} & \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D) \\ \mathsf{com}_f \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp}) : & \mathsf{Verify}(\mathsf{com}_f, a, y, \pi, \mathsf{pp}) = 1 \\ (y, \pi) \leftarrow \mathsf{Open}(f, r_f, a, \mathsf{pp}) & \end{array}\right]$$

- ***Proof of Knowledge.*** *For any polynomial-sized circuit $\mathcal{A}$, there exists a PPT extractor $\mathcal{E}$ and a negligible function $\mathsf{negl}(\cdot)$, such that for any auxiliary string z and any $\lambda \in \mathbb{N}$, the following probability is $\mathsf{negl}(\lambda)$.*

$$\Pr\left[\begin{array}{rl} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D) & \\ (\pi^*, \mathsf{com}^*, y^*, a^*) \leftarrow \mathcal{A}(1^\lambda, z, \mathsf{pp}) : & \mathsf{Verify}(\mathsf{com}^*, a^*, y^*, \pi^*, \mathsf{pp}) = 1 \\ f^* \leftarrow \mathcal{E}(1^\lambda, z, \mathsf{pp}) & \wedge f^*(a^*) \neq y^* \end{array}\right]$$

If a PC *scheme satisfies an additional property of zero knowledge, then it is a zero-knowledge univariate polynomial commitment scheme (*zkPC*).*

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

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

- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D)$
- $\mathsf{com}_f \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$
- $a \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, \mathsf{pp})$
- $(y, \pi) \leftarrow \mathsf{Open}(f, r_f, a, \mathsf{pp})$
- $b \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, a, y, \pi, \mathsf{pp})$
- *Output b*

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

- $(\mathsf{com}_f, \mathsf{pp}, \mathsf{trap}) \leftarrow \mathcal{S}(1^\lambda, D)$
- $a \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, \mathsf{pp})$
- $\pi \leftarrow \mathcal{S}(\mathsf{com}_f, a, f(a), \mathsf{pp}, \mathsf{trap})$.
- $b \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, a, f(a), \pi, \mathsf{pp})$
- *Output b*

For any non-uniform polynomial-time adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for all polynomial $f \in \mathbb{F}^D[X]$,

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

Protocol 24 (**Sumcheck**). It proceeds in ℓ rounds.

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

$$h_1(x_1) \stackrel{def}{=} \sum_{b_2,\ldots,b_\ell \in \{0,1\}} h(x_1, b_2, \ldots, b_\ell)\,,$$

  $\mathcal{V}$ checks $\mu = h_1(0) + h_1(1)$. 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

$$h_i(x_i) \stackrel{def}{=} \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} h(r_1, \ldots, r_{i-1}, x_i, b_{i+1}, \ldots, b_\ell)\,,$$

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

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

$$h_\ell(x_\ell) \stackrel{def}{=} h(r_1, r_2, \ldots, r_{\ell-1}, x_\ell)\,,$$

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

A polynomial commitment scheme is said to be transparent if the public parameter pp *is simply a uniform random string, i.e., there is no secret state to generate* pp.

Multivariate polynomial commitment. The polynomial commitment could be extended for multivariate polynomials $f \in \mathcal{F} : \mathbb{F}^\ell \to \mathbb{F}$. The algorithms and definitions are similar to those of the univariate polynomial, and we use MVPC and zkMVPC to denote the multivariate schemes.

7.2.2 Interactive Proofs for Layered Circuits

We present the GKR protocol, an efficient interactive proof for layered arithmetic circuits by Goldwasser et al. [GKR15].

7.2.2.1 Sumcheck Protocol

The GKR protocol uses the sumcheck protocol as a major building block. The problem is to sum a multivariate polynomial $h : \mathbb{F}^\ell \to \mathbb{F}$ on the Boolean hypercube: $\sum_{b_1,b_2,\ldots,b_\ell \in \{0,1\}} h(b_1, b_2, ..., b_\ell)$. Directly computing the sum requires an 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 $\mathcal{P}$.We describe the sumcheck protocol in Protocol 24. The proof size of the protocol

is $O(D\ell)$, where D is the variable-degree of h, as in each round, $\mathcal{P}$ sends a univariate polynomial of one variable in h, which can be uniquely defined by $D+1$ points. The verifier time is $O(D\ell)$. The prover time depends on the degree and the sparsity of h, and we will give the complexity later in our scheme. The sumcheck protocol is complete and sound with $\epsilon = \frac{D\ell}{|\mathbb{F}|}$.

Definition 7.2.3 (Multilinear Extension). *Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. The multilinear extension of V is the unique polynomial $\tilde{V} : \mathbb{F}^\ell \to \mathbb{F}$ s.t. $\tilde{V}(x_1, x_2, ..., x_\ell) = V(x_1, x_2, ..., x_\ell)$ for all $x_1, x_2, \ldots, x_\ell \in \{0,1\}$. $\tilde{V}$ can be expressed as:*

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

where b_i is the i-th bit of $\mathbf{b}$.

Definition 7.2.4 (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 $\tilde{\beta}$ is the multilinear extension of β. Then $\tilde{\beta}$ can be expressed as: $\tilde{\beta}(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{\ell}((1-x_i)(1-y_i) + x_i y_i)$.*

7.2.2.2 GKR Protocol

With the sumcheck protocol as a building block, Goldwasser et al. [GKR15] proposed an interactive proof for the evaluation of layered arithmetic circuits. 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 values in layer i of the circuit can be written as a sumcheck equation of the values in layer $i+1$. Following the convention in prior works of GKR protocols [CMT12; Tha13b; ZGKPP17c; XZZPS19b; ZXZS], we denote the number of gates in the i-th layer as S_i and let $s_i = \lceil \log S_i \rceil$. We then define a function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ 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. We also define two additional functions $add_i, mult_i : \{0,1\}^{s_{i-1}+2s_i} \to \{0,1\}$, 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 gates $\mathbf{x}, \mathbf{y}$ as input. By taking their multilinear extensions, for any $\mathbf{g}^{(i)} \in \mathbb{F}^{s_i}$, $\tilde{V}_i$ can be written as:

$$\begin{aligned}\tilde{V}_i(\mathbf{g}^{(i)}) &= \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} f_i(\mathbf{g}^{(i)}, \mathbf{x}, \mathbf{y}) \\ &= \sum_{\mathbf{x},\mathbf{y}\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(\mathbf{g}^{(i)}, \mathbf{x}, \mathbf{y})(\tilde{V}_{i+1}(\mathbf{x}) + \tilde{V}_{i+1}(\mathbf{y})) \\ &\quad + \tilde{mult}_{i+1}(\mathbf{g}^{(i)}, \mathbf{x}, \mathbf{y})\tilde{V}_{i+1}(\mathbf{x})\tilde{V}_{i+1}(\mathbf{y}). \end{aligned} \tag{7.1}$$

With Equation 7.1, as $\tilde{add}_{i+1}$ and $\tilde{mult}_{i+1}$ are publicly known, upon receiving the output, the verifier can reduce a claim of $\tilde{V}_0(\mathbf{g}^{(0)})$ to a claim $\tilde{V}_1(\mathbf{g}^{(1)})$ about layer 1, and recursively to $\tilde{V}_d(\mathbf{g}^d)$ through sumcheck protocols layer by layer. With the optimal algorithms for the prover in the GKR protocol proposed in [XZZPS19b], we have the following theorem:

Theorem 7.2.5. *[XZZPS19b]. Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d layered arithmetic circuit. There exists an interactive proof protocol for the function computed by C with soundness $O(d \log |C|/|\mathbb{F}|)$. The total*

communication is $O(d \log |C|)$ and the running time of the prover $\mathcal{P}$ is $O(|C|)$. When C has regular wiring pattern[3], the running time of $\mathcal{V}$ is $O(n + k + d \log |C|)$.

Lifting GKR protocols to argument systems. The GKR protocol is not an argument system supporting witness from $\mathcal{P}$, as in the last round, $\mathcal{V}$ needs to evaluate $\tilde{V}_d$ defined by the input of the circuit at a random point locally. To address this problem, in [ZGKPP17c], Zhang et al. first construct an argument system by combining the polynomial commitments with the GKR protocol. In their scheme, $\mathcal{P}$ first commits to the multilinear extension of $\mathcal{P}$'s witness by MVPC.Commit before the GKR protocol. In the last random of the GKR protocol, instead of evaluating locally, $\mathcal{V}$ queries $\mathcal{P}$ the evaluation on $\mathcal{P}$'s witness. $\mathcal{P}$ invokes MVPC.Open to prove the correctness of the evaluation and $\mathcal{V}$ validates it using MVPC.Verify. Combined with $\mathcal{V}$'s public input, $\mathcal{V}$ is able to verify the last claim about $\tilde{V}_d$ in the GKR protocol. Subsequent works [ZGKPP17a; WTSTW18; XZZPS19b; ZXZS] improve the efficiency and achieve zero-knowledge based on the framework. We follow the same framework in our scheme with a transparent setup, and we present the protocol explicitly in Section 7.3.

7.3 Transparent Polynomial Commitment with Prover Batching

We first present our transparent polynomial commitment scheme with prover batching for multiple evaluations. There are several candidates of transparent polynomial commitment schemes recently [BFS19; ZXZS; WTSTW18; Lee20; VP19] with the prover time of $\tilde{O}(t)$ for a single evaluation. However, in the application of the VSS scheme in Protocol 23, if the dealer naively runs the transparent polynomial commitment scheme on N evaluations separately, the running time will be $\tilde{O}(Nt)$.

In our scheme, we reduce the prover time of generating all N proofs to $O(N \log N)$ field operations, which is asymptotically the same as evaluating the polynomial at N points. We propose the notion of *one-to-many* zero knowledge arguments, where each verifier receives one output out of the entire output of a common computation represented by circuit C. Instead of running a zero knowledge proof protocol with each verifier separately, which may take $\tilde{O}(N|C|)$ time for the prover in the worst case, our scheme reduces the prover time to $\tilde{O}(|C| + N \log N)$.

When applied to the polynomial commitment for VSS, we set the evaluations at powers of the N-th root of unity $\mathbb{w}$ (i.e., $\omega^N = 1 \mod p$). In this way, we realize the polynomial commitment with prover batching by instantiating the circuit C in our one-to-many zero knowledge argument with the classical butterfly circuit [Wei69] for the FFT algorithm. The circuit takes the coefficients of the polynomial f as input, and outputs $f(\mathbb{w}^0), \ldots, f(\mathbb{w}^{N-1})$, where each verifier $\mathcal{V}_j$ receives $f(\mathbb{w}^j)$. With our one-to-many zero knowledge argument, the prover is able to generate all proofs in time $O(N \log N)$.

Below, we will first describe our one-to-many zero knowledge argument scheme assuming that somehow, all verifiers send the same challenge in every round (Protocol 25) — to aid understanding, the reader may assume for the time being that all verifiers query a trusted random oracle in each round to generate a common random challenge. Later, we will describe a new Fiat-Shamir-style transformation (Protocol 26) to make Protocol 25 non-interactive, such that all verifiers would effectively share the same challenges in this non-interactive version. Finally, we will prove the soundness of our new Fiat-Shamir transformation using the techniques inspired by Ben-Sasson, Chiesa, and Spooner [BSCS16]. For simplicity, we first describe

[3] "Regular" circuits is defined in [CMT12, Theorem A.1]. Roughly speaking, it means the mutilinear extension of its wiring predicates can be evaluated at a random point in time $O(\log |C|)$.

a simplified version of our protocol *without zero-knowledge*. In full version, we will describe how to use standard techniques to additionally achieve zero-knowledge.

7.3.1 One-to-Many Argument System Given Shared Random Challenges

Following the notation of the GKR protocol in Section 7.2.2, we denote the entire output of circuit C as $\tilde{V}_0(\mathbf{x})$ for $\mathbf{x} \in \{0,1\}^{\log N}$. Suppose the size of the output is N and each verifier $\mathcal{V}_j$ receives one output $\tilde{V}_0(\mathbf{j})$, where $\mathbf{j}$ is the binary representation of j. Using its multilinear extension, we can write each $\tilde{V}_0(\mathbf{j})$ as a sumcheck of $\tilde{V}_0(\mathbf{x})$ using the identity function $\tilde{\beta}$:

$$\tilde{V}_0(\vec{j}) = \sum_{\mathbf{x} \in \{0,1\}^{\log N}} \tilde{\beta}(\mathbf{j}, \mathbf{x}) \tilde{V}_0(\mathbf{x}).$$

$\mathcal{P}$ and $\mathcal{V}_j$ can run one sumcheck protocol to reduce the claim about $\tilde{V}_0(\mathbf{j})$ to the claim about $\tilde{V}_0(\mathbf{g}^{(0)})$ for $\mathbf{g}^{(0)} \in \mathbb{F}^{\log N}$. This is equivalent to adding an additional layer of "selector" that selects the j-th output for $\mathcal{V}_j$. Assuming all verifiers share the same random challenge in every round, they will share the random vector of $\mathbf{g}^{(0)}$ during the sumcheck protocol and share the last claim of $\tilde{V}_0(\mathbf{g}^{(0)})$. Then $\mathcal{P}$ invokes the GKR-based argument on the circuit C to prove to all verifiers the correctness of $\tilde{V}_0(\mathbf{g}^{(0)})$. Given the common randomness during the invocation, the prover computes the same message for all verifiers in every round. Thus in this step, the total computational cost of $\mathcal{P}$ the same as that of proving to a single verifier, which is $\tilde{O}(|C|)$. We present the formal protocol in Protocol 25.

It remains to show that in Step 4 of Protocol 25, the prover can generate all messages in the sumcheck protocols with all the verifiers in $O(N \log N)$ time. As there are N different sumcheck protocols and the size of the polynomial in each sumcheck is $O(N)$, naively running Step 4 takes $O(N^2)$ time using existing techniques. However, we observe that all the sumcheck protocols with different $\mathcal{V}_j$ share the same polynomial $\tilde{V}_0(\vec{x})$. The only difference is that the identity function $\tilde{\beta}$ takes different $\vec{j}$. By utilizing the special structure of the identity function $\tilde{\beta}$, we are able to come up with a new algorithm to run all sumcheck protocols efficiently. The algorithm initializes and updates a lookup table based on $\tilde{V}_0$ once for all verifiers in every round. Then using the lookup table, the prover is able to generate the message for each sumcheck protocol in every round in a constant time. Thus the prover time is $O(N)$ per round, and thus is $O(N \log N)$ in total. We present the formal algorithm in Algorithm 20. As shown in Step 6 of Algorithm 20, in the i-th round of the sumcheck (see Protocol 24 for the message in each round of sumcheck), the messages defined by $\tilde{V}_0$ are shared among all verifiers and can be computed by the prover in $O(N/2^i)$ time. In Step 7, the lookup table is updated based on the randomness received in this round in $O(N/2^i)$. Then in Step 10, the prover generates the messages for each verifier $\mathcal{V}_j$ utilizing the closed form of the identity function as described in Section 7.2. Since the sumcheck protocol has $\log N$ rounds in total, the total prover time is $O(N + \frac{N}{2} + \ldots + 1 + N \log N) = O(N \log N)$. Using Algorithm 20 for Step 4 in Protocol 25, and the transparent zkMVPC scheme in [ZXZS] with $O(n \log n)$ prover time, $O(\log^2 n)$ verifier time and proof size, where n is the size of the input plus the witness, we have the following theorem.

Theorem 7.3.1. *For each $\mathcal{V}_j$ and $\mathcal{P}$, Protocol 25 is an argument system for the function $[\boldsymbol{out}]_j = [C(\boldsymbol{in})]_j$ such that $\boldsymbol{out} = C(\boldsymbol{in})$ with soundness $O(d \log |C|/|\mathbb{F}|)$. The proof size is $O(d \log |C| + \log^2 n)$ and the verifier time is $O(d \log |C| + \log^2 n)$. If N verifiers have the common random challenge in every round, the total prover time is $O(|C| + N \log N + n \log n)$.*

Protocol 25. **Batching prover computation for N verifiers that share random challenges.**
Let λ be the security parameter. Let C: $\mathbb{F}^n \to \mathbb{F}^N$ be a d-depth layered arithmetic circuit. For any $j \in [N]$, $\mathcal{P}$ needs to convince $\mathcal{V}_j$ that $\mathbf{out}_j = [C(\mathbf{in})]_j$ where $[C(\mathbf{in})]_j$ is the j-th output of the circuit given input $\mathbf{in}$, and $\mathbf{out}_j$ is the the claimed result for $\mathcal{V}_j$. Without loss of generality, assume n and N are both powers of 2 and we can pad them if not.
Here, we assume that all verifiers obtain their random challenge from a common random oracle in each round.

1. Set $\mathsf{pp} \leftarrow \mathsf{MVPC.KeyGen}(1^\lambda)$.
2. $\mathcal{P}$ invokes $\mathsf{MVPC.Commit}(\tilde{V}_d, \mathsf{pp})$ to generate $\mathsf{com}_{\tilde{V}_d}$ and broadcasts $\mathsf{com}_{\tilde{V}_d}$ to all verifiers. $\tilde{V}_d$ is the multilinear extension of input values, as defined in the GKR protocol.
3. For each $j \in [N]$, $\mathcal{P}$ sends $\tilde{V}_0(\vec{j})$ as $\mathbf{out}_j$ to $\mathcal{V}_j$ separately.
4. For each $j \in [N]$, $\mathcal{P}$ and $\mathcal{V}_j$ run a sumcheck protocol on

$$\tilde{V}_0(\vec{j}) = \sum_{\mathbf{x} \in \{0,1\}^{\log N}} \tilde{\beta}(\mathbf{j}, \mathbf{x}) \tilde{V}_0(\mathbf{x}),\ \mathbf{j} \text{ is the binary string of } j$$

   At the end of the protocol, $\mathcal{V}_j$ receives $\tilde{V}_0(\mathbf{g}^{(0)})$ for the common random vector of $\mathbf{g}^{(0)}$. $\mathcal{V}_j$ computes $\tilde{\beta}(\vec{j}, \mathbf{g}^{(0)})$ and checks the last statement of the sumcheck protocol.
5. For all verifiers $\mathcal{P}$ invokes the GKR protocol on the circuit C to generate the proof given $\tilde{V}_0(\mathbf{g}^{(0)})$.
6. For all verifiers, in the last round of the GKR protocol, they have the claim about $\tilde{V}_d(\mathbf{g}^{(d)})$. $\mathcal{P}$ and $\mathcal{V}_j$ invoke $\mathsf{MVPC.Open}$ and $\mathsf{MVPC.Verify}$ on $\tilde{V}_d(\mathbf{g}^{(d)})$ with $\mathsf{com}_{\tilde{V}_d}$ and pp. If it is equal to $\tilde{V}_d(\mathbf{g}^{(d)})$ sent by $\mathcal{P}$, $\mathcal{V}_j$ outputs $\mathtt{1}$, otherwise $\mathcal{V}_j$ outputs $\mathtt{0}$.

Proof. **Completeness.** For each $\mathcal{V}_j$ and $\mathcal{P}$, the completeness is straightforward.

Soundness. For each j, if $\tilde{V}_0(j) \neq [C(\mathbf{in})]_j$, let $\tilde{V}_0^\dagger(\mathbf{g}^{(0)})$ be the correct value corresponding to C. If $\tilde{V}_0(\mathbf{g}^{(0)}) \neq \tilde{V}_0^\dagger(\mathbf{g}^{(0)})$, then $\mathcal{V}_j$ outputs $\mathtt{0}$ in Step 6 with the probability of $O(d \log |C|/|\mathbb{F}|)$ by the soundness of the GKR protocol. If $\tilde{V}_0(\mathbf{g}^{(0)}) = \tilde{V}_0^\dagger(\mathbf{g}^{(0)})$, $\mathcal{V}$ outputs $\mathtt{0}$ in Step 4 with probability of $O(\log N/|\mathbb{F}|)$ by the soundness of the sumcheck protocol. Thus, the total probability is bounded by $O(d \log |C|/|\mathbb{F}|)$ by the union bound.

Efficiency. For each verifier $\mathcal{V}_j$, the proof size and the verification time in Step 4 are $O(\log N)$ while the proof size and the verification time in Step 5 are $O(d \log |C|)$. If Protocol 25 employs the transparent MVPC scheme in [ZXZS], the proof size and the verification time are $O(\log^2 n)$ in Step 6. Thus the verification time and the proof size are $O(d \log |C| + \log^2 n)$ for an individual verifier. The prover runs in $O(N \log N)$ time for N verifiers in Step 4 by Algorithm 20. $\mathcal{P}$ also invokes the GKR protocol on C in Step 5. Given the same

Algorithm 20 $\{a_{1,0}, \ldots, a_{1,N-1}, \ldots, a_{\log N,0}, \ldots, a_{\log N,N-1}\} \leftarrow \mathsf{SumCheck}\ (\tilde{V}(\mathbf{x}),\ g_1^{(0)},\ \ldots,\ g_{\log N}^{(0)})$

Input: $\tilde{V}(\mathbf{x})$ for $\mathbf{x} \in \{0,1\}^{\log N}$, random $g_1^{(0)}, \ldots, g_{\log N}^{(0)}$;

Output: For each $j \in [N]$, $\log N$ sumcheck messages $(a_{1,j}, \ldots, a_{\log N,j})$ for $\tilde{V}(\mathbf{j}) = \sum_{x \in \{0,1\}^{\log N}} \tilde{\beta}(\mathbf{j}, \mathbf{x})\tilde{V}_0(\mathbf{x})$. Each message $a_{i,j}$ consists of 3 elements $(a_{i0,j}, a_{i1,j}, a_{i2,j})$;

1: Initialize $\mathsf{beta}_j = 1$ for all $j \in [N]$.
2: Initialize an array $V[B] = \tilde{V}_0(\vec{b})$ for all $\vec{b} \in \{0,1\}^{\log N}$. *// $\mathbf{b}$ is the binary representation of integer B.*
3: **for** Round $i = 1, \ldots, \log N$ **do**
4: **for** Evaluation point $r = 0, 1, 2$ **do**
5: **for** $\mathbf{b} \in \{0,1\}^{\log N - i}$ **do**
6: $\tilde{V}_0(g_1^{(0)}, \ldots, g_{i-1}^{(0)}, r, \mathbf{b}) = V[B] \cdot (1-r) + V[B + 2^{\log N - i}] \cdot r$
7: $V[B] = V[B] \cdot (1 - g_i^{(0)}) + V[B + 2^{\log N - i}] \cdot g_i^{(0)}$
8: **end for**
9: **for** $j = 0, \ldots, N-1$ **do**
10: $a_{it,j} = \mathsf{beta}_j \cdot [(1-j_i) \cdot (1-r) + j_i \cdot r] \cdot \tilde{V}_0(g_1^{(0)}, \ldots, g_{i-1}^{(0)}, r, j_{i+1}, \ldots, j_{\log N})$ *// $j_1 j_2 \ldots j_{\log N}$ is the binary representation of j.*
11: $\mathsf{beta}_j = \mathsf{beta}_j \cdot [(1-j_i) \cdot (1 - g_i^{(0)}) + j_i \cdot g_i^{(0)}]$
12: **end for**
13: **end for**
14: **end for**
15: **return** $\{a_{1,0}, \ldots, a_{1,N-1}, \ldots, a_{\log N,0}, \ldots, a_{\log N,N-1}\}$;

challenges, $\mathcal{P}$ costs $O(|C|)$ time in Step 5 to generate the common proof by Theorem 7.2.5. The prover time is $O(n \log n)$ for the zkMVPC scheme in [ZXZS]. Therefore, the total prover time is $O(|C| + N \log N + n \log n)$ asymptotically. ∎

It is not hard to see that our one-to-many zero knowledge argument scheme can be extended to support a subset of outputs per verifier in a straightforward way, and we omit the details in this paper. By instantiating the circuit C in Protocol 25 with the FFT circuit of size $|C| = O(N \log N)$ and depth $d = O(\log N)$, and the input **in** with the coefficients of the polynomial f and the N-th root of unity $\mathbb{w}$, we are able to construct a polynomial commitment scheme with prover batching. Each verifier $\mathcal{V}_j$ receives $\tilde{V}_0(\mathbf{j}) = f(\mathbb{w}^j)$, and the prover generates all proofs in $O(N \log N)$ time. Suppose $f(x) = c_0 + c_1 x + \ldots + c_t x^t$ and $t = \Theta(N)$, we have the following corollary:

Corollary 7.3.2. *For prover $\mathcal{P}$ and N verifiers $\mathcal{V}_j$ for $j \in [N]$, there exists an argument system for the function between every $\mathcal{P}$ and $\mathcal{V}_j$ that **out**$_j = f(\mathbb{w}^j)$ and **out** $= \mathsf{FFT}(c_0, \ldots, c_t)$ with soundness $O(\log^2 N/|\mathbb{F}|)$. The proof size is $O(\log^2 N)$ and the verifier time is $O(\log^2 N)$. If N verifiers have the common random challenge in every round, the total prover time is $O(N \log N)$ and communication is $O(N \log^2 N)$.*

7.3.2 A New Fiat-Shamir Transformation for Sharing Random Challenges

We now describe a new Fiat-Shamir-style transformation that makes Protocol 25 non-interactive in the random-oracle model, such that the N verifiers could effectively share the same random challenge in every round. A strawman approach is to use the standard Fiat-Shamir heuristic [FS] for each verifier $\mathcal{V}_j$ and $\mathcal{P}$, separately. In the Fiat-Shamir heuristic [FS], $\mathcal{P}$ generates $\mathcal{V}$'s random challenge by querying a random oracle on the entire transcript of messages with $\mathcal{V}$ so far. If we directly apply the Fiat-Shamir heuristic on Protocol 25 for each verifier separately, the random challenge will not be the same in every round of the protocol since each verifier $\mathcal{V}_j$ has the possibly different output $\tilde{V}_0(\mathbf{j})$ at the beginning of the protocol. Hence the previous transcript for $\mathcal{V}_j$ are divergent in any round, and thus the random oracle will output different challenges except with negligible probability.

Warmup: using a Merkle tree to merge random challenges into one. A better but still slightly inefficient approach is to use a Merkle tree to merge the random challenges into a single one. Precisely, in every round, the prover builds a Merkle tree on N random points generated by Fiat-Shamir-style transformation on N transcripts and uses the root as the unique challenge for all verifiers. $\mathcal{P}$ also attaches the corresponding Merkle path to convince each $\mathcal{V}_j$ of the correctness of the common randomness. Although the procedure guarantees the common random challenges, the Merkle tree approach will result in a multiplicative overhead of $O(\log N)$ on the prover time, proof size, and the verifier time of the whole protocol. Specifically, the $\log N$ blowup stems from the need to build the Merkle tree of size N and send a $\log N$-sized Merkle branch to every verifier in every round.

Our approach. We suggest a more efficient approach that achieves the same prover time as Protocol 25 and incurs only an additive overhead of $O(\log^2 N)$ on the proof size. We have the prover generate the verifier's random challenge by querying the random oracle at *only* the last round's challenge and message instead of the whole transcript. The advantage of this new heuristic approach is that all verifiers share the same challenge after Step 4 in Protocol 25 automatically without the Merkle tree. As described in Protocol 25, as long as the random vector of $\mathbf{g}^{(0)}$ is identical in Step 4, all verifiers receive the same claim about $\tilde{V}_0(\mathbf{g}^{(0)})$ with the same random challenge. Our heuristic transformation assures that the transcript in Steps 5-6 will be the same for each verifier. Therefore, the prover only needs to insert Merkle trees in every round of Step 4 (i.e., the output layer). We provide the formal non-interactive protocol in Protocol 26.

In a general setting, the standard Fiat-Shamir transformation usually applies only to constant-round protocols (assuming only polynomial soundness loss in the security reduction). By contrast, we are applying our new Fiat-Shamir-style transformation to a non-constant-round protocol. Nonetheless, we can still prove standard polynomial soundness loss using techniques from Ben-Sasson, Chiesa, and Spooner [BSCS16].

7.3.3 Proving Soundness of Our New Fiat-Shamir-Style Transformation

Below, we focus on Steps 3-6 of Protocol 26 on the circuit C and formally prove that the protocol is complete and sound. When combined with the MVPC in Steps 1,2 and 7, we are able to obtain a non-interactive argument by Definition of Zero-knowledge proofs.

There are two key differences between our transformation in Protocol 26 and the standard Fiat-Shamir transformation: (1) in every round, the randomness is generated by hashing the random challenge and the message in the previous round, instead of the entire transcript so far; (2) in Step 4 a Merkle tree is constructed on the hash of each verifier and the root is used as the common randomness in this round for all verifiers. However, when viewed from the perspective of a single verifier, e.g., $\mathcal{V}_0$, the second difference is actually very

Protocol 26. **Making Protocol 25 non-interactive with a new Fiat-Shamir-style tranformation** Let λ be the security parameter. Let C: $\mathbb{F}^n \to \mathbb{F}^N$ be a d-depth layered arithmetic circuit. For any $j \in [N]$, $\mathcal{P}$ needs to convince $\mathcal{V}_j$ that $\mathbf{out}_j = [C(\mathbf{in})]_j$ where $[C(\mathbf{in})]_j$ is the j-th output of the circuit given input $\mathbf{in}$, and $\mathbf{out}_j$ is the the claimed result for $\mathcal{V}_j$. Without loss of generality, assume n and N are both powers of 2 and we can pad them if not. Let ρ be a random oracle.

1. Set $\mathsf{pp} \leftarrow \mathsf{MVPC.KeyGen}(1^\lambda)$.
2. $\mathcal{P}$ invokes $\mathsf{MVPC.Commit}(\tilde{V}_d, \mathsf{pp})$ to generate $\mathsf{com}_{\tilde{V}_d}$ and broadcasts $\mathsf{com}_{\tilde{V}_d}$ to all verifiers. $\tilde{V}_d$ is the multilinear extension of $\mathbf{in}$, as defined in the GKR protocol.
3. For each $j \in [N]$, $\mathcal{P}$ sends $V_0(j)$ as $\mathbf{out}_j$ to $\mathcal{V}_j$ separately.
4. For each $j \in [N]$, $\mathcal{P}$ runs the sumcheck protocol on the equation in Step 4 of Protocol 25. For $i = 1, \ldots, \log N$:

   a) Suppose $M_{i,j}$ is the i-th univariate polynomial $\mathcal{P}$ sends to $\mathcal{V}_j$ in the sumcheck. If $i = 1$, set $r_{i,j} = \rho(\mathsf{com}_{\tilde{V}_d} || V_0(j) || M_{1,j})$. If $i > 1$, set $r_{i,j} = \rho(g_{i-1}^{(0)} || M_{i,j})$.

   b) $\mathcal{P}$ builds a Merkle tree on the vector of $\mathbf{r}^{(i)} = (r_{i,0}, \ldots, r_{i,N-1})$. Let $g_i^{(0)} = \mathsf{MT.Commit}(\mathbf{r}^{(i)})$. Then $\mathcal{P}$ assigns $g_i^{(0)}$ as the common random challenge in the i-th round.

   c) $\mathcal{P}$ attaches $(r_{i,j}, \mathsf{path}_{i,j}) \leftarrow \mathsf{MT.Open}(j, g_i^{(0)})$ in the proof.

   In the last round of the sumcheck, $\mathcal{P}$ sends $\tilde{V}_0(\mathbf{g}^{(0)})$ to each $\mathcal{V}_j$ as they share the same random vector of $\mathbf{g}^{(0)}$.
5. $\mathcal{P}$ invokes the GKR protocol with $\tilde{V}_0(\mathbf{g}^{(0)})$. In each round, $\mathcal{P}$ generates the random challenges by querying ρ on the last round's challenge and message. For all V_j, the random challenges and the transcript would be exactly the same because they share the same claim about $\tilde{V}_0(\mathbf{g}^{(0)})$ and the same random vector $\mathbf{g}^{(0)}$ from the first round of this step.
6. In the last round of the GKR protocol, all verifiers have the same claim about $\tilde{V}_d(\mathbf{g}^{(d)})$. $\mathcal{P}$ invokes $\mathsf{zkMVPC.Open}(\tilde{V}_d, \mathbf{g}^{(d)}, \mathsf{pp})$ to generate the proof for the claim.
7. For each $j \in [N]$, $\mathcal{V}_j$ checks the proof with random challenges provided by $\mathcal{P}$ and $\mathsf{zkMVPC.Verify}$. Then $\mathcal{V}_j$ checks all authenticated paths in the Merkle tree proof by $\mathsf{MT.Verify}$. In particular, given $\mathsf{path}_{i,j} = (\nu_1, \ldots, \nu_{\log N})$, for $k = 1, \ldots, \log N$: if $j_i = 0$, $\mathcal{V}_j$ computes $r_{i,j} = \rho(r_{i,j} || \nu_i || (i-1)(logN+1)+k)$; otherwise $\mathcal{V}_j$ computes $r_{i,j} = \rho(\nu_i || r_{i,j} || (i-1)(logN+1)+k)$. $\mathcal{V}_j$ checks that $r_{i,j} = g_i^{(0)}$. Finally, $\mathcal{V}_j$ queries ρ to check the generation process of random challenges.

similar to the first one. This is because in each round, $\mathcal{V}_0$ receives $\log N$ messages from the prover (claimed to be the validation path of a Merkle tree). To perform the Merkle tree verification, as shown in Step 6 of

Protocol 26, suppose $\mathcal{P}$ sends the authentication path of $(\nu_1, \ldots, \nu_{\log N})$ to $\mathcal{V}_0$ and r is $\mathcal{V}_0$'s random challenge in the current round, $\mathcal{V}_0$ computes $\mathsf{rt} = \rho(\ldots \rho(\rho(r||\nu_1)||\nu_2) \ldots ||\nu_{\log N})$ and use rt as her random challenge in the next round. When proving soundness, as the prover is malicious, there is no guarantee that these messages are indeed from the same Merkle tree among all verifiers. Therefore, it is equivalent to extending one round of the interactive version of the sumcheck protocol in Step 4 of Protocol 25 to $\log N$ rounds. In these additional rounds, the prover does nothing but sending a dummy message ν_i to the verifier. The verifier ignores the dummy message and replies with fresh randomness. Finally, the prover and the verifier use the last randomness to proceed to the next round of the original sumcheck protocol. We give this interactive protocol with dummy messages for Step 4 of Protocol 25 in Protocol 27.

Observe that when we apply our Fiat-Shamir transformation of hashing only the previous random challenge and message to Protocol 27, it becomes Protocol 26 from the perspective of each verifier, if we model the hash function in the Merkle tree as the random oracle. Moreover, intuitively Protocol 27 is sound as long as the original interactive proof is sound, as the verifier simply picks some additional randomness and ignores some dummy messages from the prover. Therefore, our strategy to prove the soundness of Protocol 26 is: (1) we first show that as long as an interactive proof protocol is secure against state restoration attacks defined in [BSCS16], the non-interactive protocol by applying the transformation of hashing only the previous message and the random challenge is sound; (2) extending one round of an interactive proof protocol to multiple rounds with dummy messages as in Protocol 27 does not affect the security against state restoration attacks.

Protocol 27. **The interactive proof with dummy messages for Step 4 of Protocol 25**

For round $i = 1 : \ldots, \log N$ of the sumcheck protocol:
For $k = 1, \ldots, \log N$:

- $\mathcal{P}$ sends $\nu_{i,k}$ to $\mathcal{V}$.
- $\mathcal{V}$ responds with random number of $\mathsf{Du}_{i,k} \in \mathbb{F}$.
- If $j_k = 0$, set $r_i^{(0)} = \rho(r_i^{(0)}||\nu_{i,k})$; otherwise, set $r_i^{(0)} = \rho(\nu_{i,k}||r_i^{(0)})$.

$\mathcal{P}$ and $\mathcal{V}_j$ use $r_i^{(0)}$ as the random challenge in round i and continue the sumcheck protocol.

Formally, let T be an interactive proof protocol with η rounds for the statement of $F(x) = y$. Let $(m_1, r_1, \ldots, m_\eta, r_\eta)$ denote a complete transcript for T, where $m_i \in \mathbb{F}^*$ is the prover's message in round i while $r_i \in \mathbb{F}$ is the verifier's randomness in round i. Let $\rho : \mathbb{F}^* \to \mathbb{F}$ denote the random oracle. In our new heuristic algorithm, the prover sets required random points as $r_1 = \rho(x||y||m_1||0)$ and $r_i = \rho(r_{i-1}||m_i||i-1)$ or $r_i = \rho(m_i||r_{i-1}||i-1)$ for $i \in \{2, \ldots, \eta\}$ to make T non-interactive. We modify the transformation by taking the round number in T as the extra input to the random oracle and exchanging the order of r_{i-1} and m_i in certain rounds. We show that if T is sound against the prover with state restoration attacks, then the non-interactive protocol is sound after the heuristic transformation.

Definition 7.3.3. *An interactive protocol* T *for the statement* $F(x') = y'$ *with* η *rounds is secure against state restoration attacks if for every* x *and* y *such that* $F(x) \neq y$*, for every* $\mathcal{P}^*$ *and an honest* $\mathcal{V}$*, Game 1 outputs* 1 *with the probability of* $\mathsf{negl}(\lambda)$.

Game 1. **The game between a state-restoring prover $\mathcal{P}^*(x, y)$ and a verifier $\mathcal{V}(x, y)$.**

1. Given x, y satisfying $F(x) \neq y$, the game initializes the set of SeenStates to be $\{\mathsf{null}\}$.
2. Repeat the following at most T times, where $T = \mathsf{poly}(\lambda)$:
   a) $\mathcal{P}^*$ chooses an element cvs in SeenStates. (cvs is short for complete verifier's state.)
   b) The game sets $\mathcal{V}$'s state to cvs.
   c) If $\mathsf{cvs} = \mathsf{null}$: $\mathcal{P}^*$ sends m_1 to $\mathcal{V}$. Then $\mathcal{V}$ returns a random point r_1 to $\mathcal{P}^*$; $\mathcal{P}^*$ adds $m_1||r_1$ into SeenStates.
   d) If $\mathsf{cvs} = m_1||r_1||\ldots||m_{i-1}||r_{i-1}$ for $1 < i \leq \eta$: $\mathcal{P}^*$ sets $\mathcal{V}$'s state to cvs, generates m_i according to cvs and sends it to $\mathcal{V}$. After receiving r_i randomly sampled by $\mathcal{V}$. $\mathcal{P}^*$ adds $\mathsf{cvs}||m_i||r_i$ into SeenStates.
   e) If $\mathsf{cvs} = m_1||r_1||\ldots||m_\eta||r_\eta$, the prover can choose to set $\mathcal{V}$'s state to cvs. $\mathcal{V}$ computes his decision b given $(x, y, m_1, r_1, \ldots, m_\eta, r_\eta)$. Then, the game halts and outputs b.
3. The game halts and outputs 0.

Theorem 7.3.4. *If T is an interactive proof for $F(x) = y$ with η rounds and it is secure against state restoration attacks in Definition 7.3.3, then after the transformation, the non-interactive protocol T' satisfies the soundness in interactive proofs.*

Proof. We use $\mathbb{P}^\rho$ and $\mathbb{V}$ to represent the prover and the verifier in the non-interactive protocol separately. Suppose $\mathbb{P}$ can query a random oracle ρ at most Δ times. Given $\Delta = \mathsf{poly}(\lambda)$, we construct a prover $\mathcal{P}^*$ with state restoration attack ability against verifier $\mathcal{V}$ in the original interactive protocol.

Construction of $\mathcal{P}^*$. We use $\mathcal{P}^*$ to simulate the random oracle for $\mathbb{P}^\rho$ and $\mathcal{P}^*$ works as follows.

1. Let ρ be a table mapping $\mathbb{F}^* \to \mathbb{F}$ and let δ be a table mapping a random point in $\mathbb{F}$ to the verifier's state. Both tables are empty in the beginning and are filled with elements as $\mathcal{P}^*$ runs the protocol. Intuitively, we use ρ to simulate $\mathbb{P}^\rho$ access to a random oracle while we use δ to keep track of $\mathcal{V}$'s states that $\mathcal{P}^*$ has "seen in his mind". Given a verifier's state cvs, let $L(\mathsf{cvs})$ be the number of rounds contained in the state, which can be simply treated as the number of $||$ in cvs by the format of cvs. Suppose the vector $\vec{e} = (e_1, \ldots, e_{\eta-1}) \in \{0, 1\}^{\eta-1}$ is public.
2. Begin simulating $\mathbb{P}^\rho$ and, for $i = 1, \ldots, \Delta$:
   a) Let θ_i denote the i-th query by $\mathbb{P}^\rho$.
   b) If θ_i has been inserted into the table ρ, $\mathcal{P}^*$ responds with $\rho(\theta_i)$. Go to next iteration for i.
   c) If $i < j$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.

d) If $i = j$, $\mathcal{P}^*$ splits θ_i to $x||y||m_1||0$ ($\mathcal{P}^*$ aborts if he cannot split θ_i to $x||y||m_1||0$). $\mathcal{P}^*$ starts the game with $\mathcal{V}$ on $f(x) = y$. $\mathcal{P}^*$ sets $\mathcal{V}$'s state to (null), sends m_1 to $\mathcal{V}$, receives the first randomness of r_1 from $\mathcal{V}$. $\mathcal{P}$ sets $\rho(x||y||m_1||0) := r_1$ and $\delta(r_1) := m_1||r_1$. Go to next iteration for i.

e) If $i > j$, suppose the last element of θ_i is k.

i. If $\theta_i = k$ or $k \geq \eta$ or $k = 0$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.

ii. If $e_k = 0$, let $r_k \in \mathbb{F}$ be the first element of θ_i. $\mathcal{P}^*$ splits θ_i to $r_k||m_{k+1}||k$. If $\delta(r_k)$ is defined and $L(\delta(r_k)) = k - 1$, $\mathcal{P}^*$ sets $\mathcal{V}$' state to $\mathsf{cvs} = \delta(r_k)$. Then $\mathcal{P}^*$ sends m_{k+1} to $\mathcal{V}$. After receiving r_{k+1} from $\mathcal{V}$, $\mathcal{P}^*$ answers $\mathbb{P}^\rho$ with r_{k+1}, sets $\rho(\theta_i) := r_{k+1}$, and sets $\delta(r_{k+1}) := \mathsf{cvs}||m_k||r_{k+1}$. If $\delta(r_k)$ is note defined or $L(\delta(r_k)) \neq k - 1$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.

iii. If $e_k = 1$, let $r_k \in \mathbb{F}$ be the last element ahead of k. $\mathcal{P}^*$ splits θ_i to $m_{k+1}||r_k||k$. If $\delta(r_k)$ is defined and $L(\delta(r_k)) = k - 1$, $\mathcal{P}^*$ sets $\mathcal{V}$' state to $\mathsf{cvs} = \delta(r_k)$. Then $\mathcal{P}^*$ sends m_{k+1} to $\mathcal{V}$. After receiving r_{k+1} from $\mathcal{V}$, $\mathcal{P}^*$ answers $\mathbb{P}^\rho$ with r_{k+1}, sets $\rho(\theta_i) := r_{k+1}$, and sets $\delta(r_{k+1}) := \mathsf{cvs}||m_k||r_{k+1}$. If $\delta(r_k)$ is not defined or $L(\delta(r_k)) \neq k - 1$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.

Our construction has two major differences from the construction in [BP]. In the construction above, $\mathcal{P}^*$ guesses the statement of $f(x) = y$ that $\mathbb{P}$ would use in the proof by assuming that the j-th query to the random oracle is $x||y||m_1||0$, instead of knowing it in advance. This only introduces a polynomial loss on the probability. Moreover, the query to the random oracle contains the round number. This is because in our non-interactive argument in Protocol 26, to verify the Merkle tree path, the prover's message is sometimes on the left and sometimes on the right of the input of the hash. Our construction of $\mathcal{P}^*$ tracks this information by the round number in order to determine the ordering of the queries to the random oracle. In particular, $\mathcal{P}^*$ use k to detect which round is relevant to the query. For each k, with the public indicator e_k, $\mathcal{P}^*$ knows that $\mathbb{P}$'s message is in the head or the tail of the string.

Analysis of $\mathcal{P}^*$. We now analyze $\mathcal{P}^*$'s ability to cheat given $\mathbb{P}$'s ability to cheat.

Let $\mathcal{U}(\lambda)$ denote the uniform distribution over all functions on $\rho : \mathbb{F}^* \to \mathbb{F}$. If ρ is uniformly sampled from $\mathcal{U}(\lambda)$, then we write $\rho \leftarrow \mathcal{U}(\lambda)$ and say that ρ is a random oracle. We claim that $\mathcal{P}^*$ simulates a $\rho \in \mathcal{U}(\lambda)$ uniformly at random. That is because, given any new input, $\mathcal{P}^*$ responds either with a uniformly random point generated by himself, or a uniformly random point provided by $\mathcal{V}$. It is equivalent to draw ρ uniformly at random in the beginning of the non-interactive protocol.

We claim that if $\mathbb{P}^\rho$ outputs the proof of $(x, y, m_1, r_1, \ldots, m_\eta, r_\eta)$ that makes $\mathbb{V}$ accept with probability $\mathsf{nonegl}(\lambda)$, then $\mathcal{P}^*$ will have $\mathsf{cvs} = m_1||r_1||\ldots||m_\eta||r_\eta$ for $F(x) = y$ to win the game with probability $\mathsf{nonegl}(\lambda)$. The formal proof is provided in the following.

Without loss of generality, we suppose $e_k = 0$ for $1 \leq k < \eta$. We define some events as follows.

1. E_1 represents that $\mathbb{P}^\rho$ outputs $(x, y, m_1, r_1, \ldots, m_\eta, r_\eta)$ that makes $\mathbb{V}$ accept. Then it satisfies $r_1 = \rho(x||y||m_1||0)$ and $r_i = \rho(r_{i-1}||m_i||i-1)$ for $1 < i \leq \eta$.

2. E_2 represents that $\mathbb{P}^\rho$ queries $\mathcal{P}^*$ at $x||y||m_1||0, r_1||m_2||1, \ldots, r_{\eta-1}||m_\eta||\eta - 1$ in order and $\mathcal{P}^*$ does not return the same value for different queries during the entire query process.

3. E_3 represents that $\mathcal{P}^*$ predicts that $\mathbb{P}^\rho$ queries $x||y||m_1||0$ for the first time in the j-th query accurately.

4. E_4 represents that $\mathsf{cvs} = m_1||r_1||\ldots||m_\eta||r_\eta$ for $F(x) = y$ is in $\mathcal{P}^*$'s $\mathsf{SeenStates}$ set and $\mathcal{P}^*$ wins the game.

First, we prove $\Pr[E_1 \wedge \mathsf{negl} E_2] \leq \mathsf{negl}(\lambda)$. Let r_0 denote $x||y$. There are three cases covering $E_1 \wedge \mathsf{negl} E_2$: (i) E_1 happens but $\mathbb{P}$ does not query $r_{i-1}||m_i||i-1$ for some $i \in \{1, \ldots, \eta-1\}$; (ii) E_1 happens but $\mathbb{P}$ queries $r_i||m_{i+1}||i$ before querying $r_{i-1}||m_i||i-1$ for some $i \in \{1, \ldots, \eta-1\}$; (iii) E_1 happens but $\mathcal{P}^*$ returns the same value for different queries. The probability of case (i) and the probability of case (ii) are both $\mathsf{negl}(\lambda)$ as $\mathbb{P}^\rho$ can not correctly guess the output of ρ for any input except with $\mathsf{negl}(\lambda)$. The probability of case (iii) is also $\mathsf{negl}(\lambda)$ as $\mathbb{P}^\rho$ can not find a collision of ρ except with $\mathsf{negl}(\lambda)$. By union bound, $\Pr[E_1 \wedge \mathsf{negl} E_2] \leq \mathsf{negl}(\lambda)$. Suppose $\Pr[E_1] = p = \mathsf{nonegl}(\lambda)$, $\Pr[E_1 \wedge E_2] = \Pr[E_1] - \Pr[E_1 \wedge \mathsf{negl} E_2] = \mathsf{nonegl}(\lambda) - \mathsf{negl}(\lambda) = \mathsf{nonegl}(\lambda)$. Then we have $\Pr[E_1 \wedge E_2 \wedge E_3] = \Pr[E_3 | E_1 \wedge E_2] \cdot \Pr[E_1 \wedge E_2] \geq \frac{1}{\Delta} \cdot \mathsf{nonegl}(\lambda) = \mathsf{nonegl}(\lambda)$.

Next, we show $\Pr[E_4 | E_1 \wedge E_2 \wedge E_3] = 1$. We prove that if $E_1 \wedge E_2 \wedge E_3$ happens, $\delta(r_i) = m_1||r_1||\ldots||m_i||r_i$ for $1 \leq i \leq \eta$ by induction. For each i, $\delta(r_i)$ was included in δ only once as there is no collision during the query phase. For $i = 1$, when $\mathbb{P}$ queries $x||y||m_1||0$ in the j-th query, $\mathcal{P}^*$ sets $\delta(r_1) := m_1||r_1$. For $i = k$, suppose $\mathcal{P}^*$ sets $\delta(r_k) := m_1||r_1||\ldots||m_k||r_k$ when $\mathbb{P}$ queries $r_{k-1}||m_k||k-1$, when $\mathbb{P}$ queries $r_k||m_{k+1}||k$ hereafter, $\mathcal{P}^*$ sets $\delta(r_{k+1}) := \delta(r_k)||m_k||r_{k+1} = m_1||r_1||\ldots||m_{k+1}||r_{k+1}$. Hence $\mathsf{cvs} = m_1||r_1||\ldots||m_\eta||r_\eta$ for $F(x) = y$ will be in $\mathcal{P}^*$'s $\mathsf{SeenStates}$ set and $\mathcal{P}^*$ will win the game. $\Pr[E_4] \geq \Pr[E_1 \wedge E_2 \wedge E_3] = \mathsf{nonegl}(\lambda)$. ∎

Theorem 7.3.5. *If the interactive proof* T *for* $F(x) = y$ *with* η *rounds is secure against state restoration attacks, after inserting* $c = \mathsf{poly}(\lambda)$*-round interaction in the* i*-th round of* T *as in Protocol 27, the new interactive protocol* T_{du} *is also secure against state restoration attacks for* $F(x) = y$*.*

Proof. (sketch) Let $\mathcal{P}$ be the prover in T and $\mathcal{P}_{\mathsf{du}}$ be the prover in T_{du}. Suppose T_{du} inserts c-round interaction with arbitrary messages of $(\mathsf{du}_1, \nu_1, \ldots, \mathsf{du}_c, \nu_c)$ in the i-th round of T. If $\mathcal{P}_{\mathsf{du}}$ can win the game described in Definition 7.3.3 with probability p for x, y satisfying $F(x) \neq y$ by generating a cvs of $(m_1, r_1, \ldots, m_i, \mathsf{du}_1, \nu_1 \ldots, \mathsf{du}_c, \nu_c, r_i, m_{i+1}, r_{i+1}, \ldots, m_\eta, r_\eta)$, then $\mathcal{P}$ can invoke $\mathcal{P}_{\mathsf{du}}$ to generate the cvs of $(m_1, r_1, \ldots, m_i, r_i, m_{i+1}, r_{i+1}, \ldots, m_\eta, r_\eta)$ to win Game 1 with probability at least p. ∎

Replacing T with T_{du} and applying the non-interactive transformation to T_{du} indicate that we can integrate an authenticated path in the Merkle tree into such a protocol T at the cost of extra $\log N$ rounds, where N is the size of the Merkle tree. Therefore, for each verifier $\mathcal{V}_j$, Protocol 27 integrate $\log N$ Merkle paths into Protocol 25 at the cost of extra $\log^2 N$ rounds.

In Protocol 26, the statement $\mathcal{P}$ wants to convince V_j is equivalent to $F_j(\mathsf{com}_{\tilde{V}_d}) = [C(\mathbf{in})]_j$, where $\mathsf{com}_{\tilde{V}_d}$ is the commitment of $\tilde{V}_d$ and $\mathcal{P}$ broadcasts to all verifiers at the beginning. F_j represents that there exists a degree-t univariate polynomial $f(x) = c_0 + c_1 x + \ldots + c_t x^t$ such that $f(\mathbb{w}^j) = [C(\mathbf{in})]_j$ and $\mathsf{com}_{\tilde{V}_d} = \mathsf{MVPC.Commit}(\tilde{c}, pp)$, where $\tilde{c}$ the multilinear extension of $(c_0, \ldots, c_t)$. For each $\mathcal{V}_j$, Protocol 26 practises our new heuristic transformation on Protocol 27 for $F_j(\mathsf{com}_{\tilde{V}_d}) = [C(\mathbf{in})]_j$ to make the proof non-interactive. The protocol will be sound after the transformation.

And thus we have the following theorem.

Theorem 7.3.6. *For each* $\mathcal{V}_j$ *and* $\mathcal{P}$*, Protocol 26 is a non-interactive argument system for the function* $\boldsymbol{out}_j = [C(\boldsymbol{in})]_j$ *such that* $\boldsymbol{out} = C(\boldsymbol{in})$*. The proof size is* $O(d \log |C| + \log^2 n + \log^2 N)$ *and the verifier time is* $O(d \log |C| + \log^2 n + \log^2 N)$*. The total prover time is* $O(|C| + N \log N + n \log n)$*.*

Efficiency. Compared to Protocol 25, the extra proof for each verifier is $\log N$ authentication paths each being of length $\log N$. Hence the proof size for each $\mathcal{V}_j$ has an extra term of $O(\log^2 N)$. The extra computation for each $\mathcal{V}_j$ is validating $\log N$ authentication paths contained in the proof by querying the random oracle $\log^2 N$ times. Thus the verification time for each verifier becomes $O(d \log |C| + \log^2 n + \log^2 N)$. The extra computation on the prover is building $\log N$ Merkle trees of size N by querying the random oracle $O(N \log N)$ times to merge the randomness in Step 4. Therefore, the total prover time is still $O(|C| + n \log n + N \log N)$ asymptotically.

Corollary 7.3.7. *For prover $\mathcal{P}$ and N verifiers $\mathcal{V}_j$ for $j \in [N]$, there exists a non-interactive argument system for the function between every $\mathcal{P}$ and $\mathcal{V}_j$ that **out**$_j = f(\mathbb{w}^j)$ and **out** $= \mathsf{FFT}(c_0, \ldots, c_t)$ with the proof size of $O(\log^2 N)$ and the verification time of $O(\log^2 N)$. The prover time is $O(N \log N)$ and the total communication cost is $O(N \log^2 N)$ given $t = \Theta(N)$.*

7.4 KZG-Based Polynomial Commitment with Prover Batching

In this section, we propose a new scheme based on the KZG polynomial commitment with prover batching, such that generating all proofs only takes $O(N \log N)$ time, without introducing any overhead on the proof size and the verifier time. We first present the formal algorithms of the original KZG polynomial commitment and then introduce our new scheme.

7.4.1 KZG Polynomial Commitment

The KZG polynomial commitment relies on the bilinear map, which is defined below.

Bilinear map. 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.

The KZG polynomial commitment is as follows.

- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, t)$: Given the security parameter and a bound on the degree of the polynomial, it runs $(p, g, \mathbb{G}, e, \mathbb{G}_T) \leftarrow \mathsf{BilGen}(1^\lambda)$. Output $\mathsf{pp} = [p, g, \mathbb{G}, e, \mathbb{G}_T, \{g^{\tau^0}, g^{\tau^1}, ..., g^{\tau^t}\}]$.
- $\mathsf{com}_f \leftarrow \mathsf{Commit}\ (f, \mathsf{pp})$: Given a polynomial $f(x) = \sum_{i=0}^{t} c_i x^i$, it computes $\mathsf{com}_f = g^{f(\tau)} = \Pi_{i=0}^{t} (g^{\tau^i})^{c_i}$.
- $\{(y, \pi)\} \leftarrow \mathsf{Open}\ (f, a, \mathsf{pp})$: For an evaluation point a, the prover computes $y = f(a)$ and polynomial $q(x) = \frac{f(x)-y}{x-a}$. Let the coefficients of q be $(q_0, q_1, ..., q_{t-1})$. The prover computes $\pi = g^{q(\tau)} = \Pi_{i=0}^{t-1} (g^{\tau^i})^{q_i}$.
- $\{1, 0\} \leftarrow \mathsf{Verify}\ (\mathsf{com}_f, a, y, \pi, \mathsf{pp})$: Given the commitment com_f, the evaluation point a, the answer y and the proof π, the verifier checks if $e(\mathsf{com}/g^y, g) \stackrel{?}{=} e(\pi, g^\tau / g^a)$. It outputs 1 if the check passes, and 0 otherwise.

The scheme is computationally-hiding under the discrete log assumption and computationally binding under the l-SBDH assumption. The prover time of the KZG commitment is $O(t)$ modular exponentiations, the proof size is $O(1)$, a single element in the base group, and the verifier time is $O(1)$, one bilinear pairing.

FFT on group elements. As the FFT algorithm only involves additions and scalar multiplications with the powers of the root of unity ω, the algorithm can be applied to a vector of elements in the base group of the bilinear map by replacing the additions with multiplications and the multiplications with exponentiations in the base group. In particular, let $A = (a_0, \ldots, a_N)$ and $g^A = (g^{a_0}, \ldots, g^{a_N})$, one can evaluate $\mathsf{FFT}(g^A) = (g^{f(\omega^0)}, \ldots, g^{f(\omega^{N-1})})$ for $f(x) = \sum_{i=0}^{N} a_i x^i$ in time $O(N \log N)$, without knowing $(a_0, \ldots, a_N)$. Similarly, one can also compute the convolution with a public vector $B = (b_0, \ldots, b_N)$ "on the exponent", i.e., $g^{A*B} = \mathsf{IFFT}(\mathsf{FFT}(g^A) \odot \mathsf{FFT}(B))$, where $\odot$ denotes element-wise exponentiation.

7.4.2 Our New Prover Batching Technique

In our new scheme, the public parameters pp, the commitment com_f and the proof π together with $\mathsf{Keygen}, \mathsf{Commit}, \mathsf{Verify}$ are exactly the same as the KZG commitment. The main contribution is that we present a new batched algorithm for the prover to generate proofs for N different evaluation points. The key idea of our scheme is to evaluate the polynomials at different powers of the N-th root of unity ω, which enables us to invoke the FFT algorithm to compute the proofs efficiently — but observing how to leverage the FFT algorithm is non-trivial. Recall that in the dealing round of the VSS scheme, for each party $i \in [N]$ the dealer computes $s_i = f(u_i)$ and $\pi_i = g^{q_i(\tau)} = g^{\frac{f(\tau)-f(u_i)}{\tau-u_i}}$. By setting the public evaluation point of party i as $u_i = \omega^i$, the dealer can compute all s_i in $O(N \log N)$ time using the FFT algorithm. However, computing the proofs π_i is more challenging, as τ is the secret key and is not explicitly given to the dealer. The dealer only has access to the public parameters $g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^t}$.

To solve this, we examine the structure of the polynomials $q_i(x)$ for $i \in [N]$. We define a bivariate polynomial $q(x, y)$ as

$$q(x, y) = \frac{f(x) - f(y)}{x - y}. \tag{7.2}$$

Then, $q_i(\tau) = q(\tau, \omega^i)$ and the proofs are $\pi_i = g^{q(\tau, y)}$ for $y = \omega^i, i \in [N]$. Let $f(x) = \sum_{j=0}^{t} c_j x^j$, we have:

$$q(\tau, y) = \frac{f(\tau) - f(y)}{\tau - y} = \frac{\sum_{j=0}^{t} c_j(\tau^j - y^j)}{(\tau - y)} = \sum_{j=1}^{t} c_j \sum_{k=1}^{j} y^{k-1} \tau^{j-k}, \tag{7.3}$$

as $\tau^j - y^j = (\tau - y) \cdot \sum_{k=1}^{j} y^{k-1} \tau^{j-k}$ for $j = 1, \ldots, N$, and the constant term c_0 cancels out for $j = 0$. By changing the order of the summations, the equation above equals to

$$\sum_{k=1}^{t} y^{k-1} \sum_{j=k}^{t} c_j \tau^{j-k} = \sum_{k=1}^{t} h_k y^{k-1}, \tag{7.4}$$

where $h_k = \sum_{j=k}^{t} c_j \tau^{j-k}$. As shown by the equations above, $q(\tau, y)$ is a degree-$(t-1)$ polynomial of variable y. If we can precompute all g^{h_k} for $k = 1 \ldots, t$, we can evaluate $g^{q(\tau, y)}$ at $y = \omega^i$ for $i \in [N]$ in $O(N \log N)$ time using the FFT algorithm on the elements in the base group.

Algorithm 21 $(\pi_0, \ldots, \pi_{N-1}) \leftarrow \mathsf{multi\_proof}(f, \omega, N, \mathsf{pp})$

Input: Polynomial $f(x) = \sum_{i=0}^{t} c_i x^i$, the number of parties N, the N-th root of unity ω and the public parameter pp containing $g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^t}$ and $(p, g, e, \mathbb{G}, \mathbb{G}_T)$.
Output: Proofs of the KZG commitment $\pi_i = g^{q_i(\tau)}$, for $i \in [N]$.

1: Set $C = (c_1, c_2, ..., c_t), g^T = (g^{\tau^{t-1}}, g^{\tau^{t-2}}, ..., g^{\tau^0})$.
2: Compute the convolution $g^H = g^{C*T} = \mathsf{IFFT}(\mathsf{FFT}(g^T) \odot \mathsf{FFT}(C))$, where $\odot$ denotes element-wise exponentiation.
3: $g^{h_k} = g^{\sum_{j=k}^{t} c_j \tau^{j-k}} = g^{H[k+t-2]}$ for each $k = 1, \ldots, t$.
4: For polynomial $h(y) = \sum_{k=1}^{t} h_k y^{k-1}$, compute $(g^{h(\omega^0)}, g^{h(\omega^1)}, \ldots, g^{h(\omega^{N-1})}) = \mathsf{FFT}(g^{h_1}, g^{h_2}, \ldots, g^{h_t})$.
5: Return $\pi_i = g^{q_i(\tau)} = g^{h(\omega^i)}$

Precomputing g^{h_k}. We observe that h_k is in the form of a convolution, and we can precompute all g^{h_k} using FFT. Let $C = (c_1, c_2, ..., c_t), T = (\tau^{t-1}, \tau^{t-2}, ..., \tau^0)$, and let $H = C * T$ be their convolution. As described in Section 7.2, we have:

$$H[\ell] = \sum_{m=0}^{\ell} C[m]T[\ell - m] = \sum_{m=0}^{\ell} c_{m+1} \tau^{t-1-(\ell-m)} . \tag{7.5}$$

By setting $\ell = k + t - 2$ and $j = m + 1$, $c_{m+1} = c_j$ and $\tau^{t-1-(\ell-m)} = \tau^{j-k}$ in Equation 7.5. Moreover, c_j is defined to be nonzero for $j \in [1, t]$, and τ^{j-k} is defined to be nonzero for $j \in [k, k + t - 1]$. Therefore, $H[\ell] = \sum_{j=1}^{k+t-1} c_j \tau^{j-k} = \sum_{j=k}^{t} c_j \tau^{j-k} = h_k$. Thus, the dealer can precompute all g^{h_k} for $k = 1, \ldots, t$ using FFT and IFFT on g^T and C in $O(t \log t)$ time without knowing τ.

Complexity analysis. We present the formal algorithm in Algorithm 21. Combining the two steps, the overall complexity of the dealer is $O(N \log N)$ modular exponentiations. The proof size and the verifier time remain $O(1)$ per evaluation. In fact, our scheme is a more efficient algorithm to generate multiple proofs, and each proof size and verification time remain exactly the same as the original KZG polynomial commitment. The security of our scheme follows directly from the security proofs in [KZG].

Note that both our new scheme and the original KZG scheme only achieve computationally-hiding and binding, but not the stronger notion of proof of knowledge and zero knowledge in Definition 7.2.2. However, they suffice to prove security for the application of VSS and DKG as shown in [KZG], as the secret and the polynomial are randomly generated. Follow-up works such as [ZGKPP17a] propose variants that achieve proof of knowledge and zero knowledge using randomized commitment and opening, and knowledge assumptions. Our scheme with prover batching also works on these variants with minimal changes and achieves stronger notions. We sketch the algorithms in the full version.

7.5 Implementation and Evaluation

We fully implemented our proposed schemes with prover batching and present the experimental results in this section.

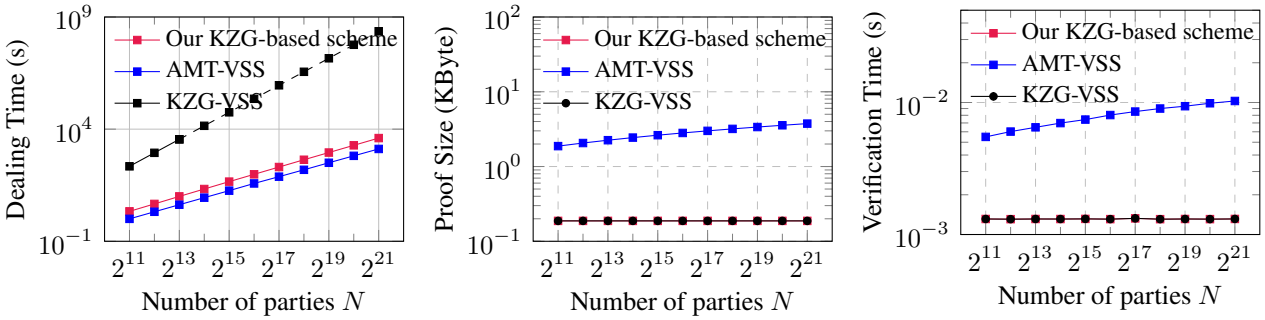


Figure 7.1: VSS Comparison, Trusted setup version

Implementation. We implemented our proposed schemes in C++ consisting of around 3000 lines of code. We used the ate-pairing library [Ate] for bilinear maps in the scheme with the trusted setup, and the GMP library [Gmp] for large numbers and arithmetic on a finite field. The implementation of our transparent polynomial commitment was based on the open-source codebase of the scheme in [ZXZS]. We used the same extension field $\mathbb{F}_{p^2}$ for $p = 2^{61} - 1$, which provides 100+ bits of security.

Configuration. We ran the experiments on an AWS c5a.24xlarge instance, which was equipped with an AMD EYPC 7002 CPU with 96 cores, 187 GB RAM[4]. All parties were executed on the same machine. We only report the numbers for the optimistic case of the VSS and DKG schemes where all the proofs are generated honestly. The polynomial commitment takes the majority of the time in this case, which is the main focus of this paper. In all the experiments, we set the degree of the polynomial as $t = N/2$, and the number of parties N ranges from 2^{11} to 2^{21}.

Counterpart comparison. In the VSS setting, we compare our KZG-based polynomial commitment scheme with two schemes that also require a trusted setup (Section 7.5.1): *(i)* naïvely running the KZG polynomial commitment for N verifiers, which incurs a prover time of $O(N^2)$; and *(ii)* the authenticated multipoint evaluation tree (AMT) scheme in [Tom+20]. We executed the open-source code of [Tom+20] on the same machine for a fair comparison. We then compare our transparent polynomial commitment scheme with running a transparent counterpart, named Virgo [ZXZS], N times to produce N proofs (Section 7.5.2). Finally, we evaluate the performance of our KZG-based and transparent schemes under DKG application, compared with AMT-DKG instantiation [Tom+20] (Section 7.5.3).

7.5.1 VSS with Trusted Setup

As shown in Figure 7.1, the running time of the dealer in our KZG-based scheme only grows quasi-linearly with the number of parties. It only takes 2.2s to generate the proofs for 2^{11} parties and takes 3,995s for 2^{21} parties. This is significantly faster than running the KZG commitment naïvely and the speedup is 100–58,000×. We could not run the naïve scheme beyond $N = 2^{12}$ due to its long-running time. Therefore, we ran up to 2^{12} parties and extrapolated the result for the larger number of parties. Comparing to the AMT scheme [Tom+20], the prover time of our scheme is slightly worse. It is 2.2× slower than AMT for $N = 2^{11}$

[4]Our KZG-based scheme only takes 2.8GB of memory in the largest instance. In our transparent scheme, the memory usage can be reduced to several gigabytes with proper pipelining by streaming the proof to each verifier without affecting the prover time.

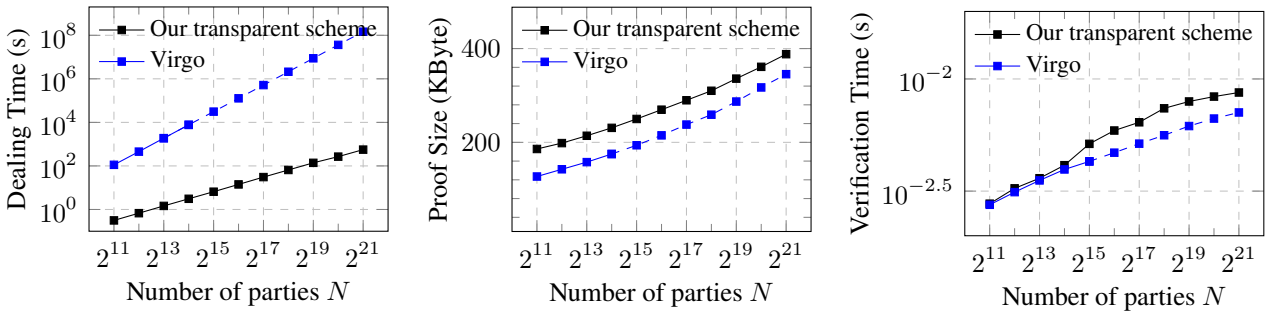


Figure 7.2: VSS Comparison, Transparent setup version

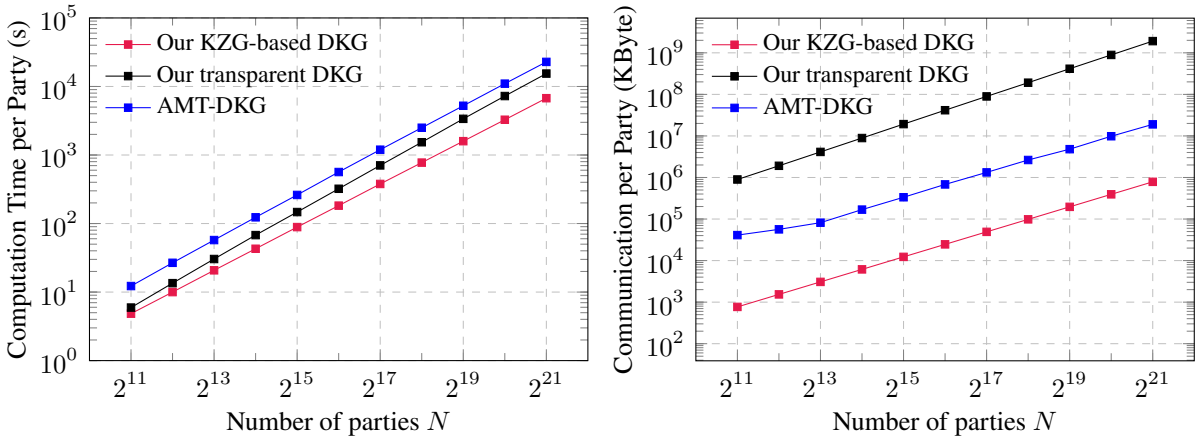


Figure 7.3: DKG Comparison

and 3× slower for $N = 2^{21}$. This is because our scheme involves 3 FFTs on the base group of the bilinear map, and the constant in our asymptotic complexity is slightly larger than that in AMT.

The proof size and the verifier time in our scheme are much smaller than AMT. They are always 192 bytes and 1.3ms *regardless of the number of parties*, which are the same as the original KZG scheme. By contrast, the proof size and the verifier time grow logarithmically in AMT. Specifically, the proof size is 20× larger than our scheme, and the verifier time is 4.2-7.8× slower. This shows that our schemes achieve much higher scalability than the state-of-the-art.

7.5.2 VSS with Transparent Setup

Figure 7.2 presents the performance of the VSS scheme with our scheme compared with Virgo. As shown in the figure, the dealing time of our transparent scheme is very fast. It only takes 0.3s to generate proofs for 2^{11} parties and 560s for 2^{21} parties. This is 700-260,000× faster than the naïve approach, which again takes a quadratic time and does not scale in practice. One may observe an interesting result that the dealing time of our transparent scheme is indeed an order of magnitude (i.e., 7-10.5×) faster than the schemes with trusted setup in Section 7.5.1. This is because our transparent scheme only incurs cheap symmetric-key operations such as hashing and field arithmetic instead of the costly modular exponentiation. This performance gain is significant even though it is generally not captured in the asymptotic cost.

The proof size and the verifier time of our transparent scheme are comparable to Virgo, as we merely introduce an additional sumcheck for each verifier. The proof size varies from 200KiB to 390KiB, and the verifier time varies from 2.8ms to 8.7ms. The proof size is larger than the KZG-based schemes because of the underlying techniques of interactive proofs. However, notice that our transparent scheme removes the trusted setup, which is critical in some applications.

7.5.3 Distributed Key Generation Experiment

We report the total computation time and communication for each party of the DKG schemes using our polynomial commitments in Figure 7.3, and compare it with AMT-DKG [Tom+20].

The overall computation time of our protocols grows quasi-linearly with the number of parties. For example, it takes $15,400$s for our transparent scheme to run a DKG of 2^{21} participants, and $6,700$s for our KZG-based scheme. These are $1.5\times$ and $3.3\times$ faster than the AMT scheme respectively. This is because our transparent scheme only incurs cheap symmetric operations, as discussed above, despite being asymptotically logarithmically slower than AMT. On the other hand, our KZG-based scheme incurs a lower verification time for each party to verify the proofs from the other parties. Moreover, our transparent scheme is slower than our KZG-based scheme in the application of DKG. This is because although the prover time of our transparent scheme is faster, its verifier time is slower ($O(\log^2 N)$ vs. $O(1)$). In DKG, each party verifies the proof of every other party, which becomes the bottleneck of our transparent scheme.

The total communication of our KZG-based scheme is orders of magnitude smaller than AMT. Specifically, it is always $192 \cdot N$ bytes in our scheme, while the proof size of AMT-DKG grows quasilinearly. Concretely our KZG-based scheme achieves the communication of only 0.8GB for $N = 2^{21}$, which is $20\times$ smaller than AMT. Due to techniques to remove the trusted setup, the communication in our transparent scheme is $100\times$ larger than AMT, which matches their asymptotic cost difference (i.e., $O(N \log^2 N)$ vs. $O(N \log N)$).

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.

[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: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[Ajt96] M. Ajtai. "Generating hard instances of lattice problems". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM. 1996, pp. 99–108.

[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).

[Ate] *Ate-pairing*. `https://github.com/herumi/ate-pairing`.

[Aur] *libiop*. `https://github.com/scipr-lab/libiop`.

[Axe] *Axelar*. `https://axelar.network/`. 2022.

[BAZB19] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. "Zether: Towards Privacy in a Smart Contract World." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 191.

[BBBPWM] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *Proceedings of the Symposium on Security and Privacy (SP), 2018*. Vol. 00, pp. 319–338.

[BBCDPGL18] C. Baum, J. Bootle, A. Cerulli, R. Del Pino, J. Groth, and V. Lyubashevsky. "Sublinear Lattice-Based Zero-Knowledge Arguments for Arithmetic Circuits". In: *Annual International Cryptology Conference*. Springer. 2018, pp. 669–699.

[BBF18] D. Boneh, B. Bünz, and B. Fisch. *Batching techniques for accumulators with applications to iops and stateless blockchains*. Tech. rep. Cryptology ePrint Archive, Report 2018/1188, Tech. Rep, 2018.

[BC99] N. P. Byott and R. J. Chapman. "Power sums over finite subspaces of a field". In: *Finite Fields and Their Applications* 5.3 (1999), pp. 254–265.

[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.

[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: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 336–365.

[BCGJM18] J. Bootle, A. Cerulli, J. Groth, S. Jakobsen, and M. Maller. "Arya: Nearly linear-time zero-knowledge proofs for correct program execution". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2018, pp. 595–626.

[BCGMMW18] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. *Zexe: Enabling Decentralized Private Computation*. Cryptology ePrint Archive, Report 2018/962. `https://eprint.iacr.org/2018/962`. 2018.

[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: *Proceedings of the Symposium on Security and Privacy SP, 2014*. 2014.

[BFL91] L. Babai, L. Fortnow, and C. Lund. "Non-deterministic exponential time has two-prover interactive protocols". In: *Computational complexity* 1.1 (1991), pp. 3–40.

[BFRSBW] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. "Verifying computations with state". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP, 2013*.

[BFS19] B. Bünz, B. Fisch, and A. Szepieniec. *Transparent SNARKs from DARK Compilers*. Cryptology ePrint Archive, Report 2019/1229. 2019.

[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.

[BGV] S. Benabbas, R. Gennaro, and Y. Vahlis. "Verifiable Delegation of Computation over Large Datasets". In: *CRYPTO 2011*, pp. 111–131.

[BKP11] M. Backes, A. Kate, and A. Patra. "Computational verifiable secret sharing revisited". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2011, pp. 590–609.

[Bla79] G. R. Blakley. "Safeguarding cryptographic keys". In: *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society. 1979.

[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.

[BOGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation”. In: *STOC*. 1988.

[Bot] “Human-guided burrito bots raise questions about the future of robo-delivery”. In: 2019.

[BP] E. Boyle and R. Pass. “Limits of Extractability Assumptions with Distributional Auxiliary Input”. In: *ASIACRYPT 2015*, pp. 236–261.

[BPS16] I. Bentov, R. Pass, and E. Shi. “Snow White: Provably Secure Proofs of Stake.” In: *IACR Cryptol. ePrint Arch.* 2016.919 (2016).

[BPTG15] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. “Machine learning classification over encrypted data.” In: *NDSS*. Vol. 4324. 2015, p. 4325.

[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: *Annual International Cryptology Conference*. Springer. 2019, pp. 701–732.

[BSCGTV] 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*.

[BSCGTV15] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. “Secure sampling of public parameters for succinct zero knowledge proofs”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 287–304.

[BSCRSVW19] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. “Aurora: Transparent succinct arguments for R1CS”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2019, pp. 103–128.

[BSCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. “Interactive oracle proofs”. In: *TCC*. Springer. 2016.

[BSCTV] 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*.

[BSCTV14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Scalable zero knowledge via cycles of elliptic curves”. In: *CRYPTO 2014*. 2014, pp. 276–294.

[BTVW14] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish. *Verifiable computation using multiple provers*. Cryptology ePrint Archive, Report 2014/846. `https://eprint.iacr.org/2014/846`. 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).

[CCD88] D. Chaum, C. Crépeau, and I. Damgard. "Multiparty Unconditionally Secure Protocols". In: *STOC*. 1988.

[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.

[CD] R. Cramer and I. Damgård. "Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free?" In: *Annual International Cryptology Conference, 1998*.

[CFQ] M. Campanelli, D. Fiore, and A. Querol. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs." In: *CCS 2019*.

[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.

[CGGN17] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo. "Zero-knowledge contingent payments revisited: Attacks and payments for services". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 229–243.

[CGJKR99] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. "Adaptive security for threshold cryptosystems". In: *CRYPTO*. 1999.

[CGMA85] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. "Verifiable secret sharing and achieving simultaneity in the presence of faults". In: *FOCS*. 1985.

[Cha+17] M. Chase et al. "Post-quantum zero-knowledge and signatures from symmetric-key primitives". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 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*. 2020, pp. 738–768.

[Cir] *ed25519-circom*. https://github.com/Electron-Labs/ed25519-circom. 2022.

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[CMT12] G. Cormode, M. Mitzenmacher, and J. Thaler. "Practical Verified Computation with Streaming Interactive Proofs". In: *ITCS*. 2012.

[Coi] *Cryptocurrency prices, charts and market capitalizations*. 2022. URL: https://coinmarketcap.com/.

[Cos] *Cosmos*. https://cosmos.network/. 2022.

[Cos+] C. Costello et al. "Geppetto: Versatile Verifiable Computation". In: *S&P 2015*.

[COS19] A. Chiesa, D. Ojha, and N. Spooner. *Fractal: Post-Quantum and Transparent Recursive Proofs from Holography*. Cryptology ePrint Archive, Report 2019/1076. https://eprint.iacr.org/2019/1076. 2019.

[Dar] *DARPA SIEVE program*. https://www.darpa.mil/news-events/2019-07-18.

[DFKP15] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. "Proofs of space". In: *Annual Cryptology Conference*. Springer. 2015, pp. 585–605.

[DG17] D. Dua and C. Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[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).

[DXR] S. Das, Z. Xiang, and L. Ren. *Asynchronous Data Dissemination and its Applications*. Cryptology ePrint Archive, Report 2021/777.

[Edd] *ed25519-circom*. https://github.com/Electron-Labs/ed25519-circom. 2022.

[Fel87] P. Feldman. "A practical scheme for non-interactive verifiable secret sharing". In: *FOCS*. 1987.

[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: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[FG] D. Fiore and R. Gennaro. "Publicly Verifiable Delegation of Large Polynomials and Matrix Computations, with Applications". In: *CCS 2012*, pp. 501–512.

[Fil] *Filecoin: A Decentralized Storage Network*. 2014. URL: https://filecoin.io/filecoin.pdf.

[FJR15] M. Fredrikson, S. Jha, and T. Ristenpart. "Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, 2015. DOI: 10.1145/2810103.2813677. URL: https://doi.org/10.1145%2F2810103.2813677.

[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: An optimizing compiler for verifiable, zero-knowledge neural network inferences". In: *Cryptology ePrint Archive* (2021).

[FS] A. Fiat and A. Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Crypto 1986*.

[GGG17] Z. Ghodsi, T. Gu, and S. Garg. "Safetynets: Verifiable execution of deep neural networks on an untrusted cloud". In: *Advances in Neural Information Processing Systems*. 2017, pp. 4672–4681.

[GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. “Quadratic Span Programs and Succinct NIZKs without PCPs”. In: *EUROCRYPT 2013*. 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.

[GJKR99] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. “Secure distributed key generation for discrete-log based cryptosystems”. In: *Eurocrypt*. 1999.

[GJMMST21] K. Gurkan, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu. “Aggregatable Distributed Key Generation”. In: (2021).

[GKMMM18] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. “Updatable and universal common reference strings with applications to zk-SNARKS”. In: *Annual International Cryptology Conference*. Springer. 2018, pp. 698–728.

[GKMPS20] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song. *Storing and Retrieving Secrets on a Blockchain*. Cryptology ePrint Archive, Report 2020/504. 2020.

[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.

[GLLTXZ] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. *Efficient Asynchronous Byzantine Agreement without Private Setups*. Cryptology ePrint Archive, Report 2021/810.

[GMO16] I. Giacomelli, J. Madsen, and C. Orlandi. “ZKBoo: Faster Zero-Knowledge for Boolean Circuits.” In: *USENIX Security Symposium*. 2016, pp. 1069–1083.

[Gmp] *The GNU Multiple Precision Arithmetic Library*. `https://gmplib.org/`.

[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/`.

[Gro09] J. Groth. “Linear algebra with sub-linear zero-knowledge arguments”. In: *Advances in Cryptology-CRYPTO 2009*. 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.

[Gro16a] J. Groth. “On the Size of Pairing-Based Non-interactive Arguments”. In: *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*. 2016, pp. 305–326.

[Gro16b] J. Groth. “On the Size of Pairing-Based Non-interactive Arguments”. In: *EUROCRYPT 2016*. 2016, pp. 305–326.

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

[Het] *Hetzner*. `https://www.hetzner.com/`. 2022.

[HJKY95] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. "Proactive secret sharing or: How to cope with perpetual leakage". In: *Annual International Cryptology Conference*. Springer. 1995, pp. 339–352.

[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).

[IS90] I. Ingemarsson and G. J. Simmons. "A protocol to set up shared secret schemes without the assistance of a mutually trusted party". In: *Workshop on the Theory and Application of of Cryptographic Techniques*. 1990.

[Jsna] *jSNARK*. `https://github.com/akosba/jsnark`.

[Jsnb] *jsnark*. `https://github.com/akosba/jsnark`. 2015.

[Kat10] A. Kate. "Distributed Key Generation and Its Applications". In: (2010).

[KG09] A. Kate and I. Goldberg. "Distributed key generation for the internet". In: *ICDCS*. 2009.

[Kil92] J. Kilian. "A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract)". In: *Proceedings of the ACM Symposium on Theory of Computing*. 1992.

[KKMS20] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman. "Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures." In: *CCS*. 2020.

[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*.

[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.

[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.

[Lee20] J. Lee. *Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments*. Cryptology ePrint Archive, Report 2020/1274. 2020.

[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] *Libra implementation*. https://github.com/sunblaze-ucb/fastZKP/tree/Libra.

[Libb] "libsnark". In: 2014.

[Lip12] H. Lipmaa. "Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments". In: *Theory of Cryptography Conference*. 2012.

[LMPR] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. "SWIFFT: A Modest Proposal for FFT Hashing". In: *Fast Software Encryption*. Springer Berlin Heidelberg, pp. 54–72. DOI: 10.1007/978-3-540-71039-4_4. URL: https://doi.org/10.1007%2F978-3-540-71039-4_4.

[LXZ21] T. Liu, X. Xie, and Y. Zhang. "ZkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2968–2985.

[Mar+19] S. K. D. Maram et al. *CHURP: Dynamic-Committee Proactive Secret Sharing*. Cryptology ePrint Archive, Report 2019/017. https://eprint.iacr.org/2019/017. 2019.

[MBKM19] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. *Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings*. Cryptology ePrint Archive, Report 2019/099. https://eprint.iacr.org/2019/099. 2019.

[Mer] R. C. Merkle. "A Certified Digital Signature". In: *CRYPTO 1989*, pp. 218–238.

[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).

[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.

[NBBR16] W. Neji, K. Blibech, and N. Ben Rajeb. "Distributed key generation protocol with a new complaint management strategy". In: *Security and communication networks* 9.17 (2016), pp. 4585–4595.

[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.

[Ped91] T. P. Pedersen. "Non-interactive and information-theoretic secure verifiable secret sharing". In: *CRYPTO*. 1991.

[PHGR13] B. Parno, J. Howell, C. Gentry, and M. Raykova. "Pinocchio: Nearly practical verifiable computation". In: *S&P 2013*. 2013, pp. 238–252.

[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.

[RBO89] T. Rabin and M. Ben-Or. "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority". In: *STOC*. 1989.

[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).

[Sch79] J. T. Schwartz. "Probabilistic algorithms for verification of polynomial identities". In: *International Symposium on Symbolic and Algebraic Manipulation*. Springer. 1979, pp. 200–215.

[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.

[Set20a] S. Setty. "Spartan: Efficient and general-purpose zkSNARKs without trusted setup". In: *Annual International Cryptology Conference*. Springer. 2020.

[Set20b] S. Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup". In: *CRYPTO 2020*. Springer International Publishing, 2020, pp. 704–737.

[Sha79] A. Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[Sha92] A. Shamir. “Ip= pspace”. In: *Journal of the ACM (JACM)* 39.4 (1992), pp. 869–877.

[SL20] S. Setty and J. Lee. *Quarks: Quadruple-efficient transparent zkSNARKs*. Cryptology ePrint Archive, Report 2020/1275. 2020.

[SLL08] D. A. Schultz, B. Liskov, and M. Liskov. “Mobile proactive secret sharing”. In: *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. ACM. 2008, pp. 458–458.

[Spa] *Spartan*. `https://github.com/microsoft/Spartan`. 2020.

[Sta] *Starkware*. `https://starkware.co/`. 2022.

[Tam03] R. Tamassia. “Authenticated data structures”. In: *European symposium on algorithms*. Springer. 2003, pp. 2–5.

[Tan11] O. Tange. “GNU Parallel - The Command-Line Power Tool”. In: *The USENIX Magazine* (2011). URL: `http://www.gnu.org/s/parallel`.

[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: *CRYPTO*. Ed. by R. Canetti and J. A. Garay. 2013.

[Tha15] J. Thaler. *A Note on the GKR Protocol*. Available at `http://people.cs.georgetown.edu/jthaler/GKRNote.pdf`. 2015.

[TKK19] A. Tueno, F. Kerschbaum, and S. Katzenbeisser. “Private evaluation of decision trees using sublinear cost”. In: *Proceedings on Privacy Enhancing Technologies* 2019.1 (2019), pp. 266–286.

[Tom+20] A. Tomescu et al. “Towards scalable threshold cryptosystems”. In: *S & P*. 2020.

[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.

[Val08] P. Valiant. “Incrementally verifiable computation or proofs of knowledge imply time/space efficiency”. In: *Theory of Cryptography Conference*. Springer. 2008, pp. 1–18.

[VC05] J. Vaidya and C. Clifton. “Privacy-preserving decision trees over vertically partitioned data”. In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2005, pp. 139–152.

[Vira] *Virgo implementation*. `https://github.com/sunblaze-ucb/Virgo`. 2020.

[Virb] *Virgo implementation*. `https://github.com/TAMUCrypto/virgo-plus`. 2021.

[VP19] A. Vlasov and K. Panarin. “Transparent Polynomial Commitment Scheme with Polylogarithmic Communication Complexity.” In: *IACR Cryptol. ePrint Arch.* (2019).

[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.

[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.

[WB15] M. Walfish and A. J. Blumberg. "Verifying computations without reexecuting them". In: *Commun. ACM* 58.2 (2015), pp. 74–84.

[Wei69] C. J. Weinstein. *Quantization effects in digital filters*. Tech. rep. MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, 1969.

[WHGSW16] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish. "Verifiable asics". In: *S & P*. 2016.

[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*. 2018.

[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).

[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.

[YLFKM21] T. Yurek, L. Luo, J. Fairoze, A. Kate, and A. K. Miller. "hbACSS: How to Robustly Share Many Secrets." In: *IACR Cryptol. ePrint Arch.* 2021 (2021).

[You] *YouTube includes NFTs in new creator tools*. 2022. URL: `https://www.nbcnews.com/pop-culture/viral/youtube-includes-nfts-new-creator-tools-rcna15813`.

[YY13] J. Yuan and S. Yu. "Proofs of retrievability with public verifiability and constant communication cost in cloud". In: *Proceedings of the 2013 international workshop on Security in cloud computing*. ACM. 2013, pp. 19–26.

[ZFZS20] J. Zhang, Z. Fang, Y. Zhang, and D. Song. “Zero knowledge proofs for decision tree predictions and accuracy”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 2039–2053.

[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. “vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases”. In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 863–880.

[ZGKPP17c] 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: *Proceeding of IEEE Symposium on Security and Privacy (S&P)*. 2018.

[Zha+19] L. Zhao et al. “VeriML: Enabling Integrity Assurances and Fair Payments for Machine Learning as a Service”. In: *arXiv preprint arXiv:1909.06961* (2019).

[Zha+20] J. Zhang et al. *Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time*. Cryptology ePrint Archive, Report 2020/1247. 2020.

[Zip79] R. Zippel. “Probabilistic algorithms for sparse polynomials”. In: *International Symposium on Symbolic and Algebraic Manipulation*. Springer. 1979, pp. 216–226.

[Zkc] “Zero knowledge contingent payment”. In: 2016.

[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.

[ZXZS] J. Zhang, T. Xie, Y. Zhang, and D. Song. “Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof”. In: *S&P 2020*.