NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2024-11

June, 2024

A New Paradigm for Efficient and Scalable Zero-Knowledge Proofs

Chenkai Weng

Abstract

Zero-Knowledge proof (ZKP) is a combination of cryptographic protocol and mathematical proof that allows a prover to convince a verifier on the correctness of a statement, without revealing its private information that validates the statement. It is a privacy-enhancing technique that can not only prove NP statements, but also correct operations over secrets. The exemplary functionalities of ZKP-based systems include proving sufficiency of bank deposit without revealing the amount; proving correct inference of an image by a private neural network model; proving correct SQL execution on a private database.

Most of existing ZKPs are designed for the asynchronous setting with focus on noninteractiveness, public verifiability and succinctness. However, they suffer from long proof generation time and large memory usage. In this thesis, we propose a new paradigm for streaming interactive zero-knowledge proof protocols. By allowing a constant number of round communication, it enables low proving time and memory overhead, which leads to significantly better efficiency, regarding the end-to-end running time, and scalability, regarding the complexity of statement that can be proven with constrained memory resource.

(1) We introduce the vector oblivious linear evaluation (VOLE), which is the most important building block for these interactive ZKPs. It allows a prover to efficiently commit to extended witnesses in tens to hundreds of nanoseconds per wire.

(2) We present the Wolverine protocol, which utilizes VOLE to construct an interactive commitment scheme. Then it leverages the cut-and-bucketing technique to verify the correctness

of extended witnesses with respect to non-linear gates. The Wolverine protocols works for both small (F2) and large (Fp) fields. Furthermore, we present the Quicksilver protocol that improves from Wolverine on its verification process and reduces its communication by 3×. Its method is also extended to the verification of polynomial sets.
(3) To further improve the communication overhead, we propose the AntMan protocol that proves (B,C)−SIMD circuits with cost O(C), while the naive implementation requires O(BC). By adding a check of wiring consistency, it is able to prove an arbitrary size-C circuit with O(C3/4) communication overhead.
(4) We discuss the Mystique protocol that enables Quicksilver to handle mixed statements. Specifically, it allows the statement representation to compose both Boolean and arithmetic circuits as well as polynomials. This design allows efficient proof of complicated statements and promotes the versatility of ZKP applications such as the zero-knowledge machine learning (ZKML).

Keywords

Applied Cryptography, Zero-knowledge proofs, Vector oblivious linear evaluation

NORTHWESTERN UNIVERSITY

A New Paradigm for Efficient and Scalable Zero-Knowledge Proofs

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Chenkai Weng

EVANSTON, ILLINOIS

June 2024

ABSTRACT

A New Paradigm for Efficient and Scalable Zero-Knowledge Proofs

Chenkai Weng

Zero-Knowledge proof (ZKP) is a combination of cryptographic protocol and mathematical proof that allows a prover to convince a verifier on the correctness of a statement, without revealing its private information that validates the statement. It is a privacy-enhancing technique that can not only prove NP statements, but also correct operations over secrets. The exemplary functionalities of ZKP-based systems include proving sufficiency of bank deposit without revealing the amount; proving correct inference of an image by a private neural network model; proving correct SQL execution on a private database.

Most of existing ZKPs are designed for the asynchronous setting with focus on non-interactiveness, public verifiability and succinctness. However, they suffer from long proof generation time and large memory usage. In this thesis, we propose a new paradigm for streaming interactive zero-knowledge proof protocols. By allowing a constant number of round communication, it enables low proving time and memory overhead, which leads to significantly better efficiency, regarding the end-to-end running time, and scalability,

regarding the complexity of statement that can be proven with constrained memory resource.

(1) We introduce the vector oblivious linear evaluation (VOLE), which is the most important building block for these interactive ZKPs. It allows a prover to efficiently commit to extended witnesses in tens to hundreds of nanoseconds per wire.

(2) We present the Wolverine protocol, which utilizes VOLE to construct an interactive commitment scheme. Then it leverages the cut-and-bucketing technique to verify the correctness of extended witnesses with respect to non-linear gates. The Wolverine protocols works for both small ($\mathbb{F}_2$) and large ($\mathbb{F}_p$) fields. Furthermore, we present the Quicksilver protocol that improves from Wolverine on its verification process and reduces its communication by $3\times$. Its method is also extended to the verification of polynomial sets.

(3) To further improve the communication overhead, we propose the AntMan protocol that proves $(B, C)-$SIMD circuits with cost $O(C)$, while the naive implementation requires $O(BC)$. By adding a check of wiring consistency, it is able to prove an arbitrary size-C circuit with $O(C^{3/4})$ communication overhead.

(4) We discuss the Mystique protocol that enables Quicksilver to handle mixed statements. Specifically, it allows the statement representation to compose both Boolean and arithmetic circuits as well as polynomials. This design allows efficient proof of complicated statements and promotes the versatility of ZKP applications such as the zero-knowledge machine learning (ZKML).

Acknowledgements

I would like to express my deepest gratitude to my advisor Professor Xiao Wang. He introduced me to the world of cryptography, and guided my through the journey of research. He not only provided me with the knowledge and skills to be a cryptographer, but also tremendous opportunities to engage with the great IACR community. I am also grateful to Professor Jonathan Katz, Jennie Rogers, Xinyu Xing, and Doctor Antigoni Polychroniadou, for kindly serving in my committee and helping improve my thesis. Also, I could not have undertaken this journey without Antigoni Polychroniadou, Melissa Chase and Dahlia Malkhi, who mentored me during my internships at JPMorgan AI Research, Microsoft Research and Chainlink Labs.

Special thanks to my collaborators for their guidance and assistance at every stage of our projects. Thanks should also go to my friends, labmates, and my fellow JP Morgan interns who are always ready to lend a hand when I am going through hard times. Lastly, I would be remiss in not mentioning my parents and my girlfriend for their unwavering support and belief in me.

Table of Contents

CHAPTER 1

Introduction

Zero-Knowledge Proof (ZKP) enables a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a statement is true, without revealing any information beyond the validity of the statement [59, 58, 11]. ZKP has found its applications in private software verification, anonymous blockchain transaction, electronic voting and machine learning. Prior popular ZKP systems include zk-SNARKs with trusted setup [63, 52], transparent interactive oracle proofs (IOPs) [12, 99, 60], MPC-in-the-Head (MPCitH) [70, 2], ZK from garbled circuits (ZKGC) [72, 50], etc. These protocols are optimized with different goals, and they all suffer from some drawbacks as well.

Among all the performance matrics that are considered for ZKP systems, the priority usually leans toward the interactiveness, public verifiability, setup assumption, proving time, verifying time, proof size, and memory usage. In the regime of non-interactive and public verifiable ZKPs, the zk-SNARKs with trusted setup usually have shortest (constant) proof size but worse proof generation time. MPCitH protocols (except for Ligero [2]) enjoy fast proof generation but have linear-size proofs. IOPs achieve the best balance between these properties by having relatively fast proving time and polylogrithmic proof size. Moreover, these schemes need to process all proof data at once, which results in memory usage linear to the complexity of the statement (i.e., circuit size or the number of R1CS constraints). On the other hand, interactive ZKPs such as ZKGC have lightweight prover algorithms and potentially support the streaming setting and thus

only need constant memory usage. However, it requires a proof size not only linear to the complexity of the statement, but the security parameter as well.

Given the above facts, none of these schemes are suitable for large scale ZKP tasks such as the proof of databases query processing with private tables, smart contract virtual machine execution with private metadata, and machine learning inference with private models. They either require unreasonable proof generation time and overwhelming memory overhead, or incur large communication overhead. The restriction on computation (regarding hardware and software co-design), memory and bandwidth have become the main factors that hinder the development and deployment of ZKPs.

This thesis introduces a new paradigm for the zero-knowledge proof named VOLE-ZK, in which VOLE stands for a cryptographic primitive – vector oblivious linear evaluation (VOLE). By introducing a constant number of round-trip communication between a prover and a designated verifier, it allows streaming setting that is $10 - 10^3\times$ faster than IOPs and other zk-SNARKs regarding the proving time. Unlike ZKGC, its communication overhead is only linear to the complexity of the statement, but not the computational security parameter. More importantly, its memory usage is constant because it leverages the VOLE-based commit-and-prove paradigm to process the proof on the fly. Based on these properties, VOLE-ZK becomes the first proof system that is able to effectively prove complicated statement using commodity hardware such as laptops and cellphones. In contrast, IOPs and other zk-SNARKs usually require a powerful server with hundreds of Gigabytes or even Terabytes to fulfill the same task; ZKGC requires $10^2\times$ larger bandwidth resources and only works for the Boolean circuit representation.

VOLE-ZK. Wolverine [96] and LPZK [46] proposed VOLE-based ZKP constructions with different verification procedures. The efficiency, scalability and flexibility of VOLE-ZK soon inspire more innovations on this topic. Quicksilver [100] propose VOLE-based batch ZK proof for polynomials statements. Mac'n'Chees [9] proposes ZK for disjunctive statements and achieves communication overhead only proportional to the longest branch. Both [97] and [6] study the conversion between Boolean and arithmetic circuits inside a VOLE-ZK statement. Meanwhile, [6] with its followed-up work [7] propose the ZK over arithmetic circuit while the wire values are in a $\mathbb{Z}_{2^k}$ ring, which contains integers that are more easily operated by CPU. LPZKv2 [43] breaks the $|C| \log_2 p$ communication boundary for layered circuits. AntMan [98] further achieves sublinear communication overhead for SIMD circuits. Franzese et al. design a VOLE-ZK in the RAM model. It cantains an efficient constant-overhead ZK-RAM which can be of independent interest. The largest-scale application so far appears in Mystique [97] and ZKSQL [81]. In Mystique, VOLE-ZK is used to prove the correct inference of a private ResNet-101 model (101 neural network layers, 42.5 million model parameters) on CIFAR-10 images. ZKSQL proves the correctness of ad-hoc SQL query processing with respect to a private committed database.

Organization. The rest of thesis starts with the vector oblivious linear evaluation (VOLE) at Chapter 2. In Chapter 3, we show the evolving of generic VOLE-based ZKP protocols including the Wolverine and Quicksilver. They are followed by the sublinear ZKP protocol shown in Chapter 4. In the end, we describe the efficient conversions for VOLE-based ZKPs and their applications in Chapter 5.

1.1. Notation

We use λ and ρ to denote the computational and statistical security parameters, respectively. We let $\mathsf{negl}(\cdot)$ denote a negligible function, and use log to denote logarithms in base 2. We write $x \leftarrow\!\!\$\, S$ to denote sampling x uniformly from a set S, and $x \leftarrow\!\!\$\, \mathcal{D}$ to denote sampling x according to a distribution $\mathcal{D}$. We define $[a, b) = \{a, \ldots, b-1\}$ and write $[n] = \{1, \ldots, n\}$. We use bold lower-case letters like $\boldsymbol{a}$ for row vectors, and bold upper-case letters like $\mathbf{A}$ for matrices. We let $\boldsymbol{a}[i]$ denote the ith component of $\boldsymbol{a}$ (with $\boldsymbol{a}[0]$ the first entry), and let $\boldsymbol{a}[i:j]$ represent the subvector $(\boldsymbol{a}[i], \ldots, \boldsymbol{a}[j-1])$.

A circuit $\mathcal{C}$ over a field $\mathbb{F}_p$ is defined by a set of input wires $\mathcal{I}_{\mathsf{in}}$ and output wires $\mathcal{I}_{\mathsf{out}}$, along with a list of gates of the form $(\alpha, \beta, \gamma, T)$, where α, β are the indices of the input wires of the gate, γ is the index of the output wire of the gate, and $T \in \{\mathsf{Add}, \mathsf{Mult}\}$ is the type of the gate. If $p = 2$, then $\mathcal{C}$ is a boolean circuit with $\mathsf{Add} = \oplus$ and $\mathsf{Mult} = \wedge$. If $p > 2$ is prime, then $\mathcal{C}$ is an arithmetic circuit where $\mathsf{Add}/\mathsf{Mult}$ correspond to addition/multiplication in $\mathbb{F}_p$. We let C denote the number of Mult gates in the circuit.

When we work in an extension field $\mathbb{F}_{p^r}$ of $\mathbb{F}_p$, we fix some monic, irreducible polynomial $f(X)$ of degree r and so $\mathbb{F}_{p^r} \cong \mathbb{F}_p[X]/f(X)$. We let $\mathsf{X} \in \mathbb{F}_{p^r}$ denote the element corresponding to $X \in \mathbb{F}_p[X]/f(X)$; thus, every $w \in \mathbb{F}_{p^r}$ can be written uniquely as $w = \sum_{i=0}^{r-1} w_i \cdot \mathsf{X}^i$ with $w_i \in \mathbb{F}_p$ for all i, and we may view elements of $\mathbb{F}_{p^r}$ equivalently as vectors in $\mathbb{F}_p^r$. When we write arithmetic expressions involving both elements of $\mathbb{F}_p$ and elements of $\mathbb{F}_{p^r}$, it is understood that values in $\mathbb{F}_p$ are viewed as lying in $\mathbb{F}_{p^r}$ in the natural way. We let $\mathbb{F}^*$ denote the nonzero elements of a field $\mathbb{F}$.

1.2. Preliminaries

1.2.1. Information-Theoretic MACs and Batch Opening

We use *information-theoretic message authentication codes* (IT-MACs) [83, 37] to authenticate values in a finite field $\mathbb{F}_p$ using an extension field $\mathbb{F}_{p^r} \supseteq \mathbb{F}_p$. In more detail, let $\Delta \in \mathbb{F}_{p^r}$ be a *global key*, sampled uniformly, that is known only by one party $\mathsf{P_B}$. A value $x \in \mathbb{F}_p$ known by the other party $\mathsf{P_A}$ can be authenticated by giving $\mathsf{P_B}$ a uniform key $\mathsf{K}[x] \in \mathbb{F}_{p^r}$ and giving $\mathsf{P_A}$ the corresponding MAC tag

$$\mathsf{M}[x] = \mathsf{K}[x] + \Delta \cdot x \in \mathbb{F}_{p^r}.$$

We denote such an authenticated value by $[x]$. Authenticated values are additively homomorphic, i.e., if $\mathsf{P_A}$ and $\mathsf{P_B}$ hold authenticated values $[x], [x']$ then they can locally compute $[x''] = [x + x']$ by having $\mathsf{P_A}$ set $x'' := x + x'$ and $\mathsf{M}[x''] := \mathsf{M}[x] + \mathsf{M}[x']$ and having $\mathsf{P_B}$ set $\mathsf{K}[x''] := \mathsf{K}[x] + \mathsf{K}[x']$. Similarly, for a public value $b \in \mathbb{F}_p$, the parties can locally compute $[y] = [x + b]$ or $[z] = [bx]$. We denote these operations by $[x''] = [x] + [x']$, $[y] = [x] + b$, and $[z] = b \cdot [x]$, respectively.

We extend the above notation to vectors of authenticated values as well. In that case, $[\boldsymbol{u}]$ means that (for some n) $\mathsf{P_A}$ holds $\boldsymbol{u} \in \mathbb{F}_p^n$ and $\boldsymbol{w} \in \mathbb{F}_{p^r}^n$, while $\mathsf{P_B}$ holds $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ with $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$. An *authenticated multiplication triple* consists of authenticated values $[x], [y], [z]$ where $z = x \cdot y$.

Batch opening of authenticated values. An authenticated value $[x]$ can be "opened" by having $\mathsf{P_A}$ send $x \in \mathbb{F}_p$ and $\mathsf{M}[x] \in \mathbb{F}_{p^r}$ to $\mathsf{P_B}$, who then verifies that $\mathsf{M}[x] \stackrel{?}{=} \mathsf{K}[x] + \Delta \cdot x$. This has soundness error $1/p^r$, and requires sending an additional $r \log p$ bits (beyond x

itself). While this can be repeated in parallel when opening multiple authenticated values $[x_1], \ldots, [x_\ell]$, communication can be reduced using batching [83, 37].

We describe two approaches for batch checking of authenticated values. The first relies on a cryptographic hash function H. Specifically, $\mathsf{P_A}$ sends (in addition to the values $x_1, \ldots, x_\ell$ themselves) a digest $h := \mathsf{H}(\mathsf{M}[x_1], \ldots, \mathsf{M}[x_\ell])$ of all the MAC tags; $\mathsf{P_B}$ then checks that $h \stackrel{?}{=} \mathsf{H}(\mathsf{K}[x_1] + \Delta \cdot x_1, \ldots, \mathsf{K}[x_\ell] + \Delta \cdot x_\ell)$. Modeling H as a random oracle with 2κ-bit output, it is not hard to see that the soundness error (i.e., the probability that $\mathsf{P_A}$ can successfully cheat about *any* value) is upper bounded by $(q_{\mathsf{H}}^2+1)/2^{2\kappa}+1/p^r$, where q_{H} denotes the number of queries that $\mathsf{P_A}$ makes to H. The communication overhead is only 2κ bits, independent of ℓ.

The second approach, which is information theoretic, works as follows:

(1) $\mathsf{P_A}$ sends $x_1, \ldots, x_\ell \in \mathbb{F}_p$ to $\mathsf{P_B}$.

(2) $\mathsf{P_B}$ picks uniform $\chi_1, \ldots, \chi_\ell \in \mathbb{F}_{p^r}$ and sends them to $\mathsf{P_A}$.

(3) $\mathsf{P_A}$ computes $\mathsf{M}[x] := \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{M}[x_i]$, and sends it to $\mathsf{P_B}$.

(4) $\mathsf{P_B}$ computes $x := \sum_{i=1}^{\ell} \chi_i \cdot x_i \in \mathbb{F}_{p^r}$ and $\mathsf{K}[x] := \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{K}[x_i] \in \mathbb{F}_{p^r}$. It accepts the opened values if and only if $\mathsf{M}[x] = \mathsf{K}[x] + \Delta \cdot x$.

The soundness error of this approach is given by Lemma 1.

Lemma 1. *Let $x_1, \ldots, x_\ell \in \mathbb{F}_p$ and $\mathsf{M}[x_1], \ldots, \mathsf{M}[x_\ell] \in \mathbb{F}_{p^r}$ be arbitrary values known to $\mathsf{P_A}$, and let Δ and $\{\mathsf{K}[x_i] = \mathsf{M}[x_i] - \Delta \cdot x_i\}_{i=1}^{\ell}$, for uniform $\Delta \in \mathbb{F}_{p^r}$, be given to $\mathsf{P_B}$. The probability that $\mathsf{P_A}$ can successfully open values $(x'_1, \ldots, x'_\ell) \neq (x_1, \ldots, x_\ell)$ to $\mathsf{P_B}$ is at most $2/p^r$.*

Proof. Fix $(x'_1, \ldots, x'_\ell) \neq (x_1, \ldots, x_\ell)$ sent by $\mathsf{P_A}$ in the first step. If we let $\omega \stackrel{\text{def}}{=} \sum_{i=1}^{\ell} \chi_i \cdot (x'_i - x_i)$, then the probability (over uniform choice of $\{\chi_i\}$) that $\omega = 0$ is at most $1/p^r$.

Assume $\omega \neq 0$. If $\mathsf{P_A}$ sends $\mathsf{M} \in \mathbb{F}_{p^r}$, then $\mathsf{P_B}$ accepts only if

$$\begin{aligned} \mathsf{M} &= \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{K}[x_i] + \Delta \cdot \sum_{i=1}^{\ell} \cdot \chi_i \cdot x'_i \\ &= \sum_{i=1}^{\ell} \chi_i \cdot (\mathsf{M}[x_i] - \Delta \cdot x_i) + \Delta \cdot \sum_{i=1}^{\ell} \chi_i \cdot x'_i \\ &= \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{M}[x_i] + \Delta \cdot \omega. \end{aligned}$$

Everything in the final expression is fixed except for Δ. Moreover, $\mathsf{P_A}$ succeeds iff $\Delta = \omega^{-1} \cdot (\mathsf{M} - \sum_{i=1}^{\ell} \chi_i \cdot \mathsf{M}[x_i])$, which occurs with probability $1/p^r$. □

We can make the second approach *non-interactive*, using the Fiat-Shamir heuristic in the random-oracle model, by computing the coefficients $\{\chi_i\}$ as the output of a hash function H evaluated on the values $\{x_i\}$ sent by $\mathsf{P_A}$ in the first step. Adapting the above proof, one can show that this has soundness error at most $(q_{\mathsf{H}} + 2)/p^r$.

Hereafter, we write $\mathsf{Open}([\boldsymbol{x}])$ to denote a generic batch opening of a vector of authenticated values. In addition, we write $\mathsf{CheckZero}([\boldsymbol{x}])$ for the special case where all x_i are supposed to be 0 and so need not be sent. We let ϵ_{open} denote the soundness error (which depends on the technique used); when using either of the techniques described above, ϵ_{open} is independent of the number ℓ of authenticated values opened.

1.2.2. Additively Homomorphic Encryption

We describe the definition of additively homomorphic encryption (AHE) schemes in the private-key setting, which specifies the abstract properties that we need for one of our ZKP protocol [98]. To simplify the description of our protocol, we assume that the plaintexts lie in a field $\mathbb{F}$. An AHE scheme consists of the Setup algorithm that generates the set of public parameters par, a key-generation algorithm KeyGen, an encryption algorithm Enc and a decryption algorithm Dec. In our IT-PAC protocol, we let $\langle m \rangle = \mathsf{Enc}(\mathsf{sk}, m; r)$ denote the ciphertext on a message m encrypted with a secret key sk and a randomness r.

We require that the AHE scheme satisfies the standard chosen plaintext attack (CPA) security, and achieves the *circuit privacy* [71]. Furthermore, we need that the AHE scheme provides the *degree-restriction* property: for some integer $k \geq 1$, given the set of public parameters par and the ciphertexts $\langle \Lambda \rangle, \ldots, \langle \Lambda^k \rangle$ for a uniform $\Lambda \in \mathbb{F}$ as input, it is infeasible to compute a ciphertext $\langle f(\Lambda) \rangle$ such that $f(\cdot)$ is a polynomial of degree at least $k+1$. The notion of *linear targeted malleability* defined by Bitansky et al. [16], which eliminates the computation on ciphertexts other than affine linear maps, implies that the above degree-restriction property holds. For the implementation, we instantiate the AHE scheme using the BGV homomorphic encryption with a single level [30]. Following the analysis [76], we have that the BGV-AHE scheme is a valid candidate for linear targeted malleability, and thus satisfies the degree-restriction property. In our optimized implementation, a rotation key is used to rotate the slots in ciphertexts. For BGV with rotation keys, there is no obvious way of computing multiplication or operations of any higher order. In fact, under the circularity assumption, the rotation key (that is merely a

ciphertext) does not reveal any secret information, and thus does not allow an adversary to break the degree-restriction property. In the following, we provide the definition of AHE schemes and a brief introduction of BGV-AHE.

An additively homomorphic encryption (AHE) scheme $\mathsf{AHE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is defined as follows:

- $\mathsf{Setup}(1^\lambda)$: On input a security parameter λ, this algorithm outputs a set of public parameters par.
- $\mathsf{KeyGen}(\mathsf{par})$: On input the set of public parameters par, this algorithm outputs a secret key sk.
- $\mathsf{Enc}(\mathsf{sk}, m)$: On input the secret key sk and a message $m \in \mathbb{F}$ where par is assumed to be an implicit input, this algorithm outputs a ciphertext c.
- $\mathsf{Dec}(\mathsf{sk}, c)$: On input the secret key sk and a ciphertext c where par is an implicit input, this algorithm outputs a message m.
- *Additively homomorphic operations:* Given the set of public parameters par and a ciphertext $c = \mathsf{Enc}(\mathsf{sk}, x)$, a party $\mathcal{P}$ can compute a ciphertext $c' = y \cdot c - b$ without knowing secret key sk, where $y, b \in \mathbb{F}$ are known by $\mathcal{P}$. The party $\mathcal{V}$ owning sk can decrypt c' to obtain $a = \mathsf{Dec}(\mathsf{sk}, c') = x \cdot y - b \in \mathbb{F}$. Here, we require that the AHE scheme achieves *circuit privacy*, which guarantees that c' does not leak any information about y and b even to the owner of sk. We refer the reader to [71, 22, 41] for the formal definition of circuit privacy. Furthermore, given par and two ciphertexts $c_1 = \mathsf{Enc}(\mathsf{sk}, m_1)$ and $c_2 = \mathsf{Enc}(\mathsf{sk}, m_2)$, one can compute $c_1 + c_2 = \mathsf{Enc}(\mathsf{sk}, m_1 + m_2)$ without knowing sk.

Informally, we say that the scheme AHE is correct, if the ciphertext c on a message m, which is obtained by $\mathsf{Enc}(\mathsf{sk}, m)$ or the additively homomorphic operations of some ciphertexts generated by $\mathsf{Enc}(\mathsf{sk}, \cdot)$, can be decrypted to m via $\mathsf{Dec}(\mathsf{sk}, c)$ with probability $1 - \mathsf{negl}(\lambda)$.

Our implementation adopts the BGV homomorphic encryption with a single level [30] to instantiate the scheme AHE. In the BGV-AHE scheme, while the ciphertexts are defined over a ring $R_q = R/qR$, the plaintexts are lied in a ring $R_p = R/pR$, where $R = \mathbb{Z}[X]/(X^N + 1)$ is a polynomial ring with integer coefficients modulo $X^N + 1$, N is a power-of-two integer and $p, q \in \mathbb{N}$ are co-prime. Using the packing technique, we can pack multiple values in a single ciphertext and support parallel computation in the single instruction multiple data (SIMD) way. In particular, we can set $\mathbb{F} = \mathbb{F}_p$ for a prime $p = 1 \pmod{2N}$ and consider an element $a \in R_p$ as a vector in $\mathbb{F}^N$. Although the BGV-AHE scheme supports N plaintext slots (i.e., allowing to encrypt N messages from $\mathbb{F}$ in a single ciphertext), we still use the notation $\mathsf{Enc}(\mathsf{sk}, m)$ to denote the encryption of a single message m for the sake of simplicity. The circuit privacy of the BGV-AHE scheme is achieved using the noise-flooding technique [54]. We refer the reader to [30, 76] for details of the BGV-AHE scheme. Besides, we can also use the BFV homomorphic encryption with a single level [29, 49] to instantiate the AHE scheme, where the circuit privacy can be guaranteed in the more efficient way using the recent rounding technique [41].

Functionality $\mathcal{F}_{\mathsf{ZK}}$

Upon receiving $(\mathsf{prove}, \mathcal{C}, w)$ from a prover $\mathcal{P}$ and $(\mathsf{verify}, \mathcal{C})$ from a verifier $\mathcal{V}$ where the same (boolean or arithmetic) circuit $\mathcal{C}$ is input by both parties, send true to $\mathcal{V}$ if $\mathcal{C}(w) = 1$; otherwise, send false to $\mathcal{V}$.

Figure 1.1. **The zero-knowledge functionality.**

1.2.3. Security Model and Functionalities

We use the *universal composability* (UC) framework [34] to prove security in the presence of a malicious, static adversary. We say that a protocol Π *UC-realizes* an ideal functionality $\mathcal{F}$ if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a PPT adversary (simulator) $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$ with arbitrary auxiliary input z, the output distribution of $\mathcal{Z}$ in the *real-world* execution where the parties interact with $\mathcal{A}$ and execute Π is computationally indistinguishable from the output distribution of $\mathcal{Z}$ in the *ideal-world* execution where the parties interact with $\mathcal{S}$ and $\mathcal{F}$.

The protocol that we construct in this work UC-realizes the standard zero-knowledge functionality $\mathcal{F}_{\mathsf{ZK}}$, reproduced in Figure 1.1 for completeness. (We omit session identifiers in all our ideal functionalities for the sake of readability.) Our ZK protocol relies on the *subfield Vector Oblivious Linear Evaluation* (sVOLE) functionality (see Figure 2.1), which is the same as that by Boyle et al. [25], except that the adversary is allowed to make a global-key query on Δ and would incur aborting for an incorrect guess. After an initialization that is done once, this functionality allows two parties to repeatedly generate a vector of authenticated values known to P_A.

We review the standard ideal functionality for oblivious transfer (OT) in Figure 1.2.

In Figure 1.3 we define a functionality $\mathcal{F}_{\mathsf{EQ}}$ implementing a weak equality test that reveals P_A's input to P_B. This functionality can be easily realized as follows: (1) P_B

Functionality $\mathcal{F}_{\mathsf{OT}}$

On receiving (m_0, m_1) with $|m_0| = |m_1|$ from a sender $\mathsf{P_A}$ and $b \in \{0,1\}$ from a receiver $\mathsf{P_B}$, send m_b to $\mathsf{P_B}$.

Figure 1.2. The OT functionality between $\mathsf{P_A}$ and $\mathsf{P_B}$.

Functionality $\mathcal{F}_{\mathsf{EQ}}$

Upon receiving V_{A} from $\mathsf{P_A}$ and V_{B} from $\mathsf{P_B}$, send $(V_{\mathsf{A}} \stackrel{?}{=} V_{\mathsf{B}})$ and V_{A} to $\mathsf{P_B}$, and do:

- If $\mathsf{P_B}$ is honest and $V_{\mathsf{A}} = V_{\mathsf{B}}$, or is corrupted and sends continue, then send $(V_{\mathsf{A}} \stackrel{?}{=} V_{\mathsf{B}})$ to $\mathsf{P_A}$.
- If $\mathsf{P_B}$ is honest and $V_{\mathsf{A}} \neq V_{\mathsf{B}}$, or is corrupted and sends abort, then send abort to $\mathsf{P_A}$.

Figure 1.3. Functionality for a weak equality test.

commits to V_{B}; (2) $\mathsf{P_A}$ sends V_{A} to $\mathsf{P_B}$; (3) $\mathsf{P_B}$ outputs $(V_{\mathsf{A}} \stackrel{?}{=} V_{\mathsf{B}})$ and aborts if they are not equal, and then opens V_{B}; (4) if $\mathsf{P_B}$ opened its commitment to a value V_{B}, then $\mathsf{P_A}$ outputs $(V_{\mathsf{A}} \stackrel{?}{=} V_{\mathsf{B}})$; otherwise it aborts. UC commitments can be realized efficiently in the random-oracle model.

1.3. Related Works

Succinct ZKPs with trusted setup. Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) schemes are the most popular and widely adopted ZKP schemes in industry and academic research. Their most significant advantages are non-interactiveness, public verifiability and succinctness (short proof and fast verification). Earlier protocols assume a trusted setup where a trusted party generates common reference strings (CRS) and publishes them to a public bulletin board. The provers prove statements assocaited with CRS and verifiers need CRS to verify the proofs [62, 13, 53, 85, 38, 63]. CRS also fecilitates the construction of constant-size proofs which sometimes contain only a few large field elements. However, most of these

SNARKs require per-program setup: a CRS is tied to a fixed statement This is fine for blockchain applications where a prover only needs to constantly prove the same statement. However, it hinders the application of ZKP to many other fields where provers need to prove impromptu statements. This problem is partially solved by employing structured reference strings (SRS). The protocols based on SRS only needs one setup for a large class of statements and SRS can be updated with a lower cost than being re-generated [82]. Another issue for these kind of schemes is the assumption of a trusted third party who generates CRS or SRS. Since ZKPs are usually used in decentralized systems, the trusted parties usually do not exist. However, a proper generation of these strings directly relates to the security of the underlying proofs, as the randomness taken as input of CRS or SRS generation algorithms should not be known by any party. A common approach is to utilize secure multi-party computations (MPC) techniques to have many parties jointly participate in the generation of CRS or SRS [84, 39]. MPC guarantees that the randomness will never be reconstructed as long as at least one parties honestly samples the randomness and deletes it after the protocol execution.

zk-SANRKs from interactive oracle proofs. Interactive oracle proofs (IOP) is a class of ZKP system between the prover and verifiers [12]. It enables a prover to construct an oracle about the statement and its witnesses. Verifiers access the oracle by sampling random queries and are convinced if the prover responses correctly. In polynomial interactive oracle proofs, the oracle is usually presented as a committed polynomial and its query is usually instantiated by polynomial evaluations [32, 91]. The IOP paradigm enables many concretely efficient transparent zk-SNARKs in which the proof generation does not require CRS or SRS that are generated by trusted parties. The most notable ones

are a group of linear-time IOPs with linear proof generation and polylogarithmic proof sizes [99, 60]. Although they incur larger proof size compared to the above zk-SNARKs with trusted setup, their prover running time is usually at least two order of magnitude faster.

ZKP from secure multi-party computation. Ishai et al. first proposed a ZKP from secure multi-party computation [70] and it is called MPC-in-the-head (MPCitH) by convention. In MPCitH schemes, a prover emulates an MPC execution of the statement circuit, which takes input its private witness. After the views of emulated parties are committed, the verifier asks to open a random party's view and checks its correctness regarding the messages it receives and sends. It leverages the privacy property of MPC to maintain the zero-knowledge, and robustness to achieve soundness. Numerous improvements have been proposed to enhance the performance of MPCitH, including a more rigorous security and performance analysis in ZKBoo [55], application to post-quantum digital signature in ZKBoo++ [35], the utilization of preprocessing MPC by Katz et al. [73], the use of sacrifice-based MPC in Baum et al. [10], and the use of recursive polynomial-based check in Delpech de Saint Guilhem et al. [42]. A special scheme is Ligero [2]. It is the first MPCitH scheme that has sublinear (square root) proof size. However, its comes with larger computational overhead compared to other MPCitH schemes.

Another proof system is ZKP from garbled circuits (ZKGC) [72, 50]. In these schemes, the prover evaluates pre-determined Boolean circuits in Garbled circuits generated and committed by the verifier. The zero-knowledge is reduced to the oblivious transfer. Its drawback is high communication overhead which has a multiplicative factor of the computational security parameter.

Space-preserving ZKPs. Previous work on complexity-preserving zero-knowledge proofs study efficient proof generation with constrained space or time budget [17, 18, 47, 67, 15, 14]. Bootle et al. propose elastic SNARKs that can either achieve linear time and space complexity, or reduce the RAM consumption to $O(\log C)$ with $O(C \log^2 C)$ computational complexity [19]. Assume an NP relation that can be verified in time T and space S by a RAM program, Bangalore et al. [5] propose a public-coin ZKP based on collision-resistant hash functions that allows the prover to run in time $\tilde{O}(T)$ and space $\tilde{O}(S)$, with proof size $\tilde{O}(T/S)$. Their space-preserving ZKP is converted from Ligero [2].

Recent recursive zk-SNARK and incremental verifiable computation (IVC) propose succinct arguments for composed circuits, which can be evaluated step by step [79, 77, 78, 90, 23, 31]. These techniques increase the scalability of the prover, who separately generates proof for each step while simultaneously proves its consistency with all previous steps without going over the history data. They can potentially support streaming proofs in a way that the input and witness for future steps are not necessary known until those steps are reached. However, many of them only support structured circuit which are divided into a sequence of components that share the same structure. More advanced IVCs cross this barrier, however they reveal the output of each step thus does not provide the zero-knowledge guarantee when they are treated as general ZK [79, 77].

CHAPTER 2

Subfield Vector Oblivious Linear Evaluation

In this section, we present a subfield vector oblivious linear evaluation (sVOLE) protocol which is the most important building block of the interactive commitment scheme that is used in VOLE-ZK. It can be executed by $\mathcal{P}$ and $\mathcal{V}$ during the preprocessing phase. As an independent interest, sVOLE is an important building block for MPC protocols as well. The functionality of $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ is shown in Figure 2.1.

In Section 2.1, we first present an sVOLE protocol with linear communication complexity. Although this already suffices for our ZK protocol, we can obtain much better efficiency using "sVOLE extension" (by analogy with OT extension), by which we extend a small number of "base" sVOLE correlations into a larger number of sVOLE correlations. Toward this end, in Section 2.2 we construct a protocol for *single-point* sVOLE (spsVOLE) in the $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$-hybrid model, where spsVOLE is like sVOLE except that the vector of authenticated values has only a *single* nonzero entry. Then, in Section 2.3, we present an efficient protocol for "sVOLE extension" using spsVOLE as a subroutine and relying on a variant of the *Learning Parity with Noise* (LPN) assumption. We provide some intuition for each protocol in the relevant section. Our implementation shows that this protocol outperforms all prior work; we discuss its concrete performance in Section 2.4.

Functionality $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$

Initialize: Upon receiving init from $\mathsf{P_A}$ and $\mathsf{P_B}$, sample $\Delta \leftarrow\!\$\, \mathbb{F}_{p^r}$ if $\mathsf{P_B}$ is honest or receive $\Delta \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store global key Δ and send Δ to $\mathsf{P_B}$, and ignore all subsequent init commands.

Extend: This procedure can be run multiple times. Upon receiving (extend, ℓ) from $\mathsf{P_A}$ and $\mathsf{P_B}$, do:

(1) If $\mathsf{P_B}$ is honest, sample $\mathsf{K}[\boldsymbol{x}] \leftarrow\!\$\, \mathbb{F}^{\ell}_{p^r}$. Otherwise, receive $\mathsf{K}[\boldsymbol{x}] \in \mathbb{F}^{\ell}_{p^r}$ from the adversary.
(2) If $\mathsf{P_A}$ is honest, sample $\boldsymbol{x} \leftarrow\!\$\, \mathbb{F}^{\ell}_{p}$ and compute $\mathsf{M}[\boldsymbol{x}] := \mathsf{K}[\boldsymbol{x}] + \Delta \cdot \boldsymbol{x} \in \mathbb{F}^{\ell}_{p^r}$. Otherwise, receive $\boldsymbol{x} \in \mathbb{F}^{\ell}_{p}$ and $\mathsf{M}[\boldsymbol{x}] \in \mathbb{F}^{\ell}_{p^r}$ from the adversary, and then recompute $\mathsf{K}[\boldsymbol{x}] := \mathsf{M}[\boldsymbol{x}] - \Delta \cdot \boldsymbol{x} \in \mathbb{F}^{\ell}_{p^r}$.
(3) Send $(\boldsymbol{x}, \mathsf{M}[\boldsymbol{x}])$ to $\mathsf{P_A}$ and $\mathsf{K}[\boldsymbol{x}]$ to $\mathsf{P_B}$.

Global-key query: If $\mathsf{P_A}$ is corrupted, receive $(\mathsf{guess}, \Delta')$ from the adversary with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send $\mathsf{success}$ to $\mathsf{P_A}$ and ignore any subsequent global-key query. Otherwise, send abort to both parties and abort.

Figure 2.1. **Functionality for subfield VOLE.**

Functionality $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$

Initialize: Upon receiving init from parties $\mathsf{P_A}$, $\mathsf{P_B}$, sample $\Delta \leftarrow\!\$\, \mathbb{F}_{p^r}$ if $\mathsf{P_B}$ is honest, and receive $\Delta \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store global key Δ, send Δ to $\mathsf{P_B}$, and ignore all subsequent init commands. Let $\boldsymbol{\Delta}_B \in \{0,1\}^{rm}$ be the bit-decomposition of Δ, where $m = \lceil \log p \rceil$.

Extend: Upon receiving (extend, u) with $u \in \mathbb{F}_p$ from $\mathsf{P_A}$ and (extend) from $\mathsf{P_B}$, this functionality operates as follows:

(1) Sample $v \leftarrow\!\$\, \mathbb{F}_{p^r}$. If $\mathsf{P_B}$ is corrupted, instead receive $v \in \mathbb{F}_{p^r}$ from the adversary.
(2) Compute $w := v + \Delta \cdot u \in \mathbb{F}_{p^r}$.
(3) If $\mathsf{P_A}$ is corrupted, receive $w \in \mathbb{F}_{p^r}$ and $\boldsymbol{u} \in \mathbb{F}^{rm}_{p}$ from the adversary, and recompute

$$v := w - \langle \boldsymbol{g} * \boldsymbol{u}, \boldsymbol{\Delta}_B \rangle \in \mathbb{F}_{p^r},$$

where $*$ denotes the component-wise product.
(4) Output (u, w) to $\mathsf{P_A}$ and v to $\mathsf{P_B}$.

Figure 2.2. **Functionality for correlated oblivious product evaluation with errors (COPEe).**

2.1. Base sVOLE

We present a "base" sVOLE protocol that is based on oblivious transfer (OT) and is inspired by prior work of Keller et al. [74, 75]. Our protocol relies on the *correlated oblivious product evaluation with errors* (COPEe) functionality $\mathcal{F}_{\mathsf{COPEe}}$, which extends the analogous functionality introduced by Keller et al. [75] to the subfield case we are interested in. We show how to UC-realize $\mathcal{F}_{\mathsf{COPEe}}$ from OT.

2.1.1. Correlated Oblivious Product Evaluation with Errors

Functionality $\mathcal{F}_{\mathsf{COPEe}}$ is described in Figure 2.2, where $m = \lceil \log p \rceil$. In $\mathcal{F}_{\mathsf{COPEe}}$, we define a "gadget vector" $\boldsymbol{g} \in \mathbb{F}_{p^r}^{rm}$ by

$$\boldsymbol{g} = \left((1, 2, \ldots, 2^{m-1}), (1, 2, \ldots, 2^{m-1}) \cdot \mathsf{X}, \ldots, (1, 2, \ldots, 2^{m-1}) \cdot \mathsf{X}^{r-1}\right).$$

For a vector $\boldsymbol{x} \in \mathbb{F}_{p^r}^{rm}$, we define

$$\langle \boldsymbol{g}, \boldsymbol{x} \rangle = \sum_{i=0}^{r-1} \left(\sum_{j=0}^{m-1} \boldsymbol{x}[i \cdot m + j] \cdot 2^j \right) \cdot \mathsf{X}^i \in \mathbb{F}_{p^r},$$

where the definition can be extended to the cases $\boldsymbol{x} \in \{0,1\}^{rm}$ or $\boldsymbol{x} \in \mathbb{F}_p^{rm}$ by viewing $\boldsymbol{x}$ as lying in $\mathbb{F}_{p^r}^{rm}$ in the natural way. The bit-decomposition of $\Delta \in \mathbb{F}_{p^r}$ is the string $\boldsymbol{\Delta}_B \in \{0,1\}^{rm}$ satisfying $\langle \boldsymbol{g}, \boldsymbol{\Delta}_B \rangle = \Delta$.

In Figure 2.3, we present a protocol $\Pi_{\mathsf{COPEe}}^{p,r}$ that UC-realizes $\mathcal{F}_{\mathsf{COPEe}}^{p,r}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model. This protocol follows the construction of Keller et al. [75], which is in turn based on the IKNP OT-extension protocol [69] and Gilboa's approach [56] for oblivious product evaluation. The main difference from prior work is that we support the subfield case.

Protocol $\Pi^{p,r}_{\mathsf{COPEe}}$

Let $m = \lceil \log p \rceil$ and PRF be a keyed function.

Initialize: This initialization procedure is executed only once.

(1) For $i \in [rm]$, $\mathsf{P_A}$ samples $K^i_0, K^i_1 \leftarrow\!\$\, \{0,1\}^\lambda$. $\mathsf{P_B}$ samples $\Delta \leftarrow\!\$\, \mathbb{F}_{p^r}$ and lets $\boldsymbol{\Delta}_B = (\Delta_1, \ldots, \Delta_{rm}) \in \{0,1\}^{rm}$ be its bit-decomposition.

(2) For $i \in [rm]$, $\mathsf{P_A}$ sends (K^i_0, K^i_1) to $\mathcal{F}_{\mathsf{OT}}$ and $\mathsf{P_B}$ sends $\Delta_i \in \{0,1\}$ to $\mathcal{F}_{\mathsf{OT}}$, which returns $K^i_{\Delta_i}$ to $\mathsf{P_B}$.

Extend: This procedure can be executed multiple times. For the jth input $u \in \mathbb{F}_p$ from $\mathsf{P_A}$, the parties execute the following:

(3) For $i \in [rm]$, do the following in parallel:
   (a) $\mathsf{P_A}$ sets $w^i_0 := \mathsf{PRF}(K^i_0, j)$ and $w^i_1 := \mathsf{PRF}(K^i_1, j)$ with $w^i_0, w^i_1 \in \mathbb{F}_p$; $\mathsf{P_B}$ computes $w^i_{\Delta_i} := \mathsf{PRF}(K^i_{\Delta_i}, j)$.
   (b) $\mathsf{P_A}$ sends $\tau^i := w^i_0 - w^i_1 - u \in \mathbb{F}_p$ to $\mathsf{P_B}$.
   (c) $\mathsf{P_B}$ computes $v^i := w^i_{\Delta_i} + \Delta_i \cdot \tau^i = w^i_0 - \Delta_i \cdot u \in \mathbb{F}_p$.

(4) Let $\boldsymbol{v} = (v^1, \ldots, v^{rm})$ and $\boldsymbol{w} = (w^1_0, \ldots, w^{rm}_0)$ such that $\boldsymbol{w} = \boldsymbol{v} + u \cdot \boldsymbol{\Delta}_B \in \mathbb{F}^{rm}_p$.

(5) $\mathsf{P_A}$ outputs $w = \langle \boldsymbol{g}, \boldsymbol{w} \rangle \in \mathbb{F}_{p^r}$ and $\mathsf{P_B}$ outputs $v = \langle \boldsymbol{g}, \boldsymbol{v} \rangle \in \mathbb{F}_{p^r}$, where $w = v + \Delta \cdot u \in \mathbb{F}_{p^r}$.

Figure 2.3. COPEe protocol in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model.

Lemma 2. *If* PRF *is a pseudorandom function, then* $\Pi^{p,r}_{\mathsf{COPEe}}$ *UC-realizes* $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$ *in the* $\mathcal{F}_{\mathsf{OT}}$*-hybrid model.*

The proof of Lemma 2 can be straightforwardly obtained by following the proof of Keller et al. [75], and is thus omitted.

2.1.2. Base sVOLE

In Figure 2.4, we present a protocol $\Pi^{p,r}_{\mathsf{base\text{-}sVOLE}}$ that UC-realizes $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ in the $\mathcal{F}_{\mathsf{COPEe}}$-hybrid model. We first describe a sub-protocol $\Pi^{p,r}_{\mathsf{base\text{-}LsVOLE}}$, which allows two parties to generate sVOLE correlations with a selective-failure leakage on Δ, meaning that a malicious $\mathsf{P_A}$ is allowed to guess a subset of Δ and the protocol execution aborts for an incorrect guess. In this sub-protocol, $\mathsf{P_B}$ performs a *correlation check* in steps 3 and 4

Protocol $\Pi^{p,r}_{\mathsf{base\text{-}sVOLE}}$

Sub-protocol $\Pi^{p,r}_{\mathsf{base\text{-}LsVOLE}}$ with selective-failure leakage:

(1) $\mathsf{P_A}$ and $\mathsf{P_B}$ send init to $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$, which returns Δ to $\mathsf{P_B}$.

(2) $\mathsf{P_A}$ samples $u_i \leftarrow\!\!\$\, \mathbb{F}_p$ for $i \in [0,n)$ and $a_h \leftarrow\!\!\$\, \mathbb{F}_p$ for $h \in [0,r)$. For $i \in [0,n)$, $\mathsf{P_A}$ sends (extend, u_i) to $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$ and $\mathsf{P_B}$ sends (extend) to $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$, which returns $w_i \in \mathbb{F}_{p^r}$ to $\mathsf{P_A}$ and $v_i \in \mathbb{F}_{p^r}$ to $\mathsf{P_B}$ such that $w_i = v_i + \Delta \cdot u_i$. For $h \in [0,r)$, both parties also call $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$ on respective inputs (extend, a_h) and (extend), following which $\mathsf{P_A}$ gets $c_h \in \mathbb{F}_{p^r}$ and $\mathsf{P_B}$ obtains $b_h \in \mathbb{F}_{p^r}$ such that $c_h = b_h + \Delta \cdot a_h$.

(3) $\mathsf{P_B}$ samples $\chi_0, \ldots, \chi_{n-1} \leftarrow\!\!\$\, \mathbb{F}_{p^r}$, and sends them to $\mathsf{P_A}$. Then $\mathsf{P_A}$ computes $x := \sum_{i=0}^{n-1} \chi_i \cdot u_i + \sum_{h=0}^{r-1} a_h \cdot \mathsf{X}^h$, $z := \sum_{i=0}^{n-1} \chi_i \cdot w_i + \sum_{h=0}^{r-1} c_h \cdot \mathsf{X}^h$, and sends (x,z) to $\mathsf{P_B}$.

(4) $\mathsf{P_B}$ computes $y := \sum_{i=0}^{n-1} \chi_i \cdot v_i + \sum_{h=0}^{r-1} b_h \cdot \mathsf{X}^h$ and checks that $z = y + \Delta \cdot x$. If not, $\mathsf{P_B}$ aborts.

(5) For $i \in [0,n)$, $\mathsf{P_A}$ defines $\boldsymbol{u}[i] = u_i$ and $\boldsymbol{w}[i] = w_i$, and $\mathsf{P_B}$ sets $\boldsymbol{v}[i] = v_i$.

Full protocol without any leakage: Let $\ell = \lceil 2\rho/r \log p \rceil + 1$.

(1) Both parties execute the above sub-protocol with parameters p and $k = \ell \cdot r$. Then, $\mathsf{P_A}$ obtains $(\boldsymbol{u}, \boldsymbol{w}) \in \mathbb{F}_p^n \times \mathbb{F}_{p^k}^n$ and $\mathsf{P_B}$ gets $\Delta \in \mathbb{F}_{p^k}$ and $\boldsymbol{v} \in \mathbb{F}_{p^k}^n$ such that $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$. By viewing an element in $\mathbb{F}_{p^k}$ as a vector in $\mathbb{F}_{p^r}^\ell$, two parties obtain $\{\boldsymbol{w}_i\}_{i\in[\ell]}$ and $\{(\Delta_i, \boldsymbol{v}_i)\}_{i\in[\ell]}$ respectively such that $\boldsymbol{w}_i = \boldsymbol{v}_i + \boldsymbol{u} \cdot \Delta_i$, $\boldsymbol{w}_i, \boldsymbol{v}_i \in \mathbb{F}_{p^r}^n$ and $\Delta_i \in \mathbb{F}_{p^r}$.

(2) $\mathsf{P_B}$ samples $\alpha_1, \ldots, \alpha_\ell \leftarrow\!\!\$\, \mathbb{F}_{p^r}$ and sends them to $\mathsf{P_A}$.
$\mathsf{P_A}$ computes $\boldsymbol{t} := \sum_{i=1}^{\ell} \alpha_i \cdot \boldsymbol{w}_i$; $\mathsf{P_B}$ computes $\boldsymbol{s} := \sum_{i=1}^{\ell} \alpha_i \cdot \boldsymbol{v}_i$ and $\Gamma := \sum_{i=1}^{\ell} \alpha_i \cdot \Delta_i$, where $\boldsymbol{t} = \boldsymbol{s} + \Gamma \cdot \boldsymbol{u}$.

(3) $\mathsf{P_A}$ outputs $\boldsymbol{u}$ and $\boldsymbol{t}$; $\mathsf{P_B}$ outputs Γ and $\boldsymbol{s}$.

Figure 2.4. **Base sVOLE protocol in the $\mathcal{F}_{\mathsf{COPEe}}$-hybrid model.**

to verify that the resulting sVOLE correlations are correct (i.e., $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$). Then, based on $\Pi^{p,r}_{\mathsf{base\text{-}LsVOLE}}$, we show how to generate sVOLE correlations without such leakage using the leftover hash lemma [68]. In protocol $\Pi^{p,r}_{\mathsf{base\text{-}sVOLE}}$, all the uniform coefficients (i.e., $\{\chi_i\}, \{\alpha_i\}$) can be computed from a random seed and a hash function modeled as a random oracle.

We prove the following theorem.

Theorem 1. *Protocol* $\Pi^{p,r}_{\mathsf{base\text{-}sVOLE}}$ *UC-realizes* $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ *in the* $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$*-hybrid model. In particular, no PPT environment* $\mathcal{Z}$ *can distinguish the real-world execution from the ideal-world execution, except with probability at most* $(r \log p)^2/p^r + 1/2^\rho$.

Recall that our protocol $\Pi^{p,r}_{\mathsf{base\text{-}sVOLE}}$ is established over the sub-protocol $\Pi^{p,r}_{\mathsf{base\text{-}LsVOLE}}$ with a selective-failure leakage on Δ (the first part of Figure 2.4). Thus, we first prove that protocol $\Pi^{p,r}_{\mathsf{base\text{-}LsVOLE}}$ UC-realizes functionality $\mathcal{F}^{p,r}_{\mathsf{LsVOLE}}$, where $\mathcal{F}^{p,r}_{\mathsf{LsVOLE}}$ is the same as $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ except that the global-key query is replaced with the following selective-failure queries:

- Wait for the adversary to input (guess, S) where S efficiently describes a subset of $\mathbb{F}_{p^r}$. If $\Delta \in S$, then send $\mathsf{success}$ to the adversary and continue. Otherwise, send abort to both parties and abort.

Based on the leftover hash lemma [68], we can prove that the full protocol (the second part of Figure 2.4) UC-realizes $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ in the $\mathcal{F}^{p,\ell\cdot r}_{\mathsf{LsVOLE}}$-hybrid model, where the resulting global key Γ is uniform in $\mathbb{F}_{p^r}$ except with probability at most $1/2^\rho$, as the inner product defines a universal hash function. Replacing $\mathcal{F}^{p,\ell\cdot r}_{\mathsf{LsVOLE}}$ with sub-protocol $\Pi^{p,\ell\cdot r}_{\mathsf{LsVOLE}}$, we obtain that protocol $\Pi^{p,r}_{\mathsf{base\text{-}sVOLE}}$ UC-realizes $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$.

Below, we focus on proving that sub-protocol $\Pi^{p,r}_{\mathsf{base\text{-}LsVOLE}}$ UC-realizes $\mathcal{F}^{p,r}_{\mathsf{LsVOLE}}$. We first consider the case of a malicious $\mathsf{P_A}$ and then consider the case of a malicious $\mathsf{P_B}$. In each case, we construct a PPT simulator $\mathcal{S}$ that runs a PPT adversary $\mathcal{A}$ as a subroutine and emulates $\mathcal{F}^{p,r}_{\mathsf{COPEe}}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

Malicious $\mathsf{P_A}$. Given access to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$,$\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COPEe}}^{p,r}$, and receives $(w_i, \boldsymbol{u}_i)$ for $i \in [0, n)$ and $(c_h, \boldsymbol{a}_h)$ for $h \in [0, r)$ from $\mathcal{A}$, where $\boldsymbol{u}_i, \boldsymbol{a}_h \in \mathbb{F}_p^{rm}$ and $m = \lceil \log p \rceil$. (In the honest case, we have that $\boldsymbol{u}_i = (u_i, \ldots, u_i)$ for some $u_i \in \mathbb{F}_p$ and $\boldsymbol{a}_h = (a_h, \ldots, a_h)$ for some $a_h \in \mathbb{F}_p$.)

(2) $\mathcal{S}$ samples $\chi_0, \ldots, \chi_{n-1} \leftarrow\!\!\$\, \mathbb{F}_{p^r}$ and sends them to $\mathcal{A}$. Then, $\mathcal{S}$ receives $x \in \mathbb{F}_{p^r}$ and $z \in \mathbb{F}_{p^r}$ from $\mathcal{A}$. Next, $\mathcal{S}$ computes an adversarially chosen error

$$e_z := z - \sum_{i=0}^{n-1} \chi_i \cdot w_i - \sum_{h=0}^{r-1} c_h \cdot \mathsf{X}^h \in \mathbb{F}_{p^r}.$$

(3) $\mathcal{S}$ computes a set S_Δ as follows:

- Solve the following equation:

$$\left\langle \boldsymbol{g} \cdot x - \boldsymbol{g} * \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{u}_i - \boldsymbol{g} * \sum_{h=0}^{r-1} \boldsymbol{a}_h \cdot \mathsf{X}^h, \boldsymbol{\Delta}_B \right\rangle = e_z \tag{2.1}$$

- For each solution $\boldsymbol{\Delta}_B$, compute $\Delta := \langle \boldsymbol{g}, \boldsymbol{\Delta}_B \rangle$ and add Δ to the set S_Δ.

(4) $\mathcal{S}$ sends $(\mathsf{guess}, S_\Delta)$ to $\mathcal{F}_{\mathsf{LsVOLE}}^{p,r}$. If receiving abort from $\mathcal{F}_{\mathsf{LsVOLE}}^{p,r}$, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ continues the simulation.

(5) $\mathcal{S}$ computes another set $\tilde{S}_{\tilde{\Delta}}$ as follows:

- Solve the following equation:

$$\left\langle \boldsymbol{g} \cdot x - \boldsymbol{g} * \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{u}_i - \boldsymbol{g} * \sum_{h=0}^{r-1} \boldsymbol{a}_h \cdot \mathsf{X}^h, \tilde{\boldsymbol{\Delta}}_B \right\rangle = 0 \tag{2.2}$$

- For each solution $\tilde{\boldsymbol{\Delta}}_B$, compute $\tilde{\Delta} := \langle \boldsymbol{g}, \tilde{\boldsymbol{\Delta}}_B \rangle$ and add $\tilde{\Delta}$ into the set $\tilde{S}_{\tilde{\Delta}}$.

(6) If $\tilde{S}_{\tilde{\Delta}}$ only involves a single entry 0, then $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ chooses any nonzero element $\tilde{\Delta} \in \tilde{S}_{\tilde{\Delta}}$, and then for $i \in [0, n)$, computes

$$u_i := \tilde{\Delta}^{-1} \cdot \langle \boldsymbol{g} * \boldsymbol{u}_i, \tilde{\boldsymbol{\Delta}}_B \rangle \tag{2.3}$$

where $\langle \boldsymbol{g}, \tilde{\boldsymbol{\Delta}}_B \rangle = \tilde{\Delta}$. (In the following analysis, we will show that u_i is unique over all possible $\tilde{\Delta}$ in set $\tilde{S}_{\tilde{\Delta}}$.)

(7) For $i \in [0, n)$, $\mathcal{S}$ computes an adversarially chosen error $\boldsymbol{e}_i := \boldsymbol{u}_i - (u_i, \ldots, u_i) \in \mathbb{F}_p^{rm}$, and then computes $w_i' := w_i - \langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{\Delta}_B \rangle \in \mathbb{F}_{p^r}$ for any $\boldsymbol{\Delta}_B$ such that $\langle \boldsymbol{g}, \boldsymbol{\Delta}_B \rangle \in S_\Delta$. Then, $\mathcal{S}$ sends $\boldsymbol{u} = (u_0, \ldots, u_{n-1})$ and $\boldsymbol{w} = (w_0', \ldots, w_{n-1}')$ to functionality $\mathcal{F}_{\mathsf{LsVOLE}}^{p,r}$.

The simulation for the protocol transcript is straightforward. Below, we first consider the case of $p = 2$, and later discuss the case of a prime $p > 2$. In the real protocol execution, the correlation check has the following equation:

$$x \cdot \Delta = z - y = \sum_{i=0}^{n-1} \chi_i \cdot (w_i - v_i) + \sum_{h=0}^{r-1} (c_h - b_h) \cdot \mathsf{X}^h + e_z \tag{2.4}$$

For a malicious $\mathsf{P_A}$, we have that $w_i - v_i = \langle \boldsymbol{g} * \boldsymbol{u}_i, \boldsymbol{\Delta}_B \rangle$ for $i \in [0, n)$ and $c_h - b_h = \langle \boldsymbol{g} * \boldsymbol{a}_h, \boldsymbol{\Delta}_B \rangle$ for $h \in [0, r)$. Thus, we can rewrite equation (2.4) as follows:

$$\begin{aligned} & x \cdot \Delta - \sum_{i=0}^{n-1} \chi_i \cdot \langle \boldsymbol{g} * \boldsymbol{u}_i, \boldsymbol{\Delta}_B \rangle - \sum_{h=0}^{r-1} \langle \boldsymbol{g} * \boldsymbol{a}_h, \boldsymbol{\Delta}_B \rangle \cdot \mathsf{X}^h = e_z \\ \Leftrightarrow & \left\langle \boldsymbol{g} \cdot x - \boldsymbol{g} * \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{u}_i - \boldsymbol{g} * \sum_{h=0}^{r-1} \boldsymbol{a}_h \cdot \mathsf{X}^h, \boldsymbol{\Delta}_B \right\rangle = e_z. \end{aligned}$$

Therefore, the set S_Δ corresponds to $\mathcal{A}$'s guess of Δ, and the probability of aborting in the ideal-world execution is the same as that in the real-world execution.

For any two different solutions $\Delta, \Delta' \in S_\Delta$, we define $\tilde{\Delta} = \Delta - \Delta' \in \mathbb{F}_{p^r}$ and thus $\tilde{\boldsymbol{\Delta}}_B = \boldsymbol{\Delta}_B - \boldsymbol{\Delta}'_B \in \{0,1\}^{rm}$. From equation (2.1), we easily obtain that equation (2.2) holds. This also shows that the set S_Δ for equation (2.1) is an affine subspace of $\mathbb{F}_{p^r}$. Note that the set $\tilde{S}_{\tilde{\Delta}}$ from equation (2.2) is a linear space parallel to S_Δ. If there is only one

solution for equation (2.1), then $\tilde{S}_{\tilde{\Delta}}$ includes only one zero entry. In this case, $\mathcal{S}$ aborts, and the probability that the real protocol execution does not abort is at most $1/p^r$.

For $h \in [0, r)$, we define

$$a_h = \tilde{\Delta}^{-1} \cdot \langle \boldsymbol{g} * \boldsymbol{a}_h, \tilde{\boldsymbol{\Delta}}_B \rangle \tag{2.5}$$

where $\tilde{\Delta} = \langle \boldsymbol{g}, \tilde{\boldsymbol{\Delta}}_B \rangle \in \tilde{S}_{\tilde{\Delta}}$ is used to compute u_i for $i \in [0, n)$ in equation (2.3). Clearly, equations (2.3) and (2.5) provide a solution for $x = \sum_{i=0}^{n-1} \chi_i \cdot u_i + \sum_{h=0}^{r-1} a_h \cdot \mathsf{X}^h$ such that for some $\tilde{\Delta} = \langle \boldsymbol{g}, \tilde{\boldsymbol{\Delta}}_B \rangle \in \tilde{S}_{\tilde{\Delta}}$ we have $\sum_{h=0}^{r-1} \langle \boldsymbol{g} \cdot a_h - \boldsymbol{g} * \boldsymbol{a}_h, \tilde{\boldsymbol{\Delta}}_B \rangle = 0$ for all $h \in [0, r)$ and $\langle \boldsymbol{g} \cdot u_i - \boldsymbol{g} * \boldsymbol{u}_i, \tilde{\boldsymbol{\Delta}}_B \rangle = 0$ for all $i \in [0, n)$. Below, we need to prove that the $\{u_i\}_{i \in [0,n)}$ computed by equation (2.3) give the unique solution for a sufficiently large subspace of $\tilde{S}_{\tilde{\Delta}}$. Now, we assume that for some $l \in \mathbb{N}$, for each $f \in [l]$, there exists a different set $\{u_{f,i}\}_{i \in [0,n)}$ along with the set $\{a_{f,h}\}_{h \in [0,r)}$ such that $x = \sum_{i=0}^{n-1} \chi_i \cdot u_{f,i} + \sum_{h=0}^{r-1} a_{f,h} \cdot \mathsf{X}^h$ and

$$\langle \boldsymbol{g} \cdot u_{f,i} - \boldsymbol{g} * \boldsymbol{u}_i, \tilde{\boldsymbol{\Delta}}_B^f \rangle = 0 \text{ for all } i \in [0, n) \text{ and } \sum_{h=0}^{r-1} \langle \boldsymbol{g} \cdot a_{f,h} - \boldsymbol{g} * \boldsymbol{a}_h, \tilde{\boldsymbol{\Delta}}_B^f \rangle \cdot \mathsf{X}^h = 0, \tag{2.6}$$

for all $\tilde{\Delta}^f = \langle \boldsymbol{g}, \tilde{\boldsymbol{\Delta}}_B^f \rangle \in \tilde{S}_f \subseteq \tilde{S}_{\tilde{\Delta}}$ such that $|\tilde{S}_f| > 1$. The condition of $|\tilde{S}_f| > 1$ is required for $\mathcal{A}$ to pass the correlation check with probability more than $1/p^r$. Since $\tilde{S}_f$ is a linear space for all $f \in [l]$ and $\tilde{S}_f \cap \tilde{S}_{f'} = \{0\}$ from the definition, and $|\tilde{S}_{\tilde{\Delta}}| \leq p^r$ by definition, we have that $l \leq r \log p$.

Let $f \neq f' \in [l]$. From equation (2.2) and $x = \sum_{i=0}^{n-1} \chi_i \cdot u_{f',i} + \sum_{h=0}^{r-1} a_{f',h} \cdot \mathsf{X}^h$, we have:

$$\sum_{i=0}^{n-1} \chi_i \cdot u_{f',i} \cdot \tilde{\Delta}^f - \sum_{i=0}^{n-1} \chi_i \cdot \langle \boldsymbol{g} * \boldsymbol{u}_i, \tilde{\boldsymbol{\Delta}}_B^f \rangle + \sum_{h=0}^{r-1} a_{f',h} \cdot \tilde{\Delta}^f \cdot \mathsf{X}^h - \sum_{h=0}^{r-1} \langle \boldsymbol{g} * \boldsymbol{a}_h, \tilde{\boldsymbol{\Delta}}_B^f \rangle \cdot \mathsf{X}^h = 0.$$

Using equation (2.6), we obtain

$$\sum_{i=0}^{n-1} \chi_i \cdot (u_{f',i} - u_{f,i}) \cdot \tilde{\Delta}^f + \sum_{h=0}^{r-1} (a_{f',h} - a_{f,h}) \cdot \tilde{\Delta}^f \cdot \mathsf{X}^h = 0 \tag{2.7}$$

By definition, there exists some $j \in [0, n)$ such that $u_{f,j} \neq u_{f',j}$. Furthermore, there are at least two values for $\tilde{\Delta}^f \in \tilde{S}_f$, and thus we assume that in the above equation $\tilde{\Delta}^f \neq 0$. Thus, $(u_{f',j} - u_{f,j}) \cdot \tilde{\Delta}^f \neq 0$. Note that $\chi_0, \ldots, \chi_{n-1}$ are sampled uniformly at random and independent from the other values involved in equation (2.7). Therefore, equation (2.7) holds with probability at most $1/p^r$. There are fewer than $l^2 \leq (r \log p)^2$ pairs $f \neq f' \in [l]$. Thus, the overall probability is bounded by $(r \log p)^2 / p^r$.

We have established that there exists a unique solution u_i for $i \in [0, n)$. This means that for all $\tilde{\Delta} = \langle \boldsymbol{g}, \tilde{\boldsymbol{\Delta}}_B \rangle \in \tilde{S}_{\tilde{\Delta}}$, we have that $\langle \boldsymbol{g} \cdot u_i - \boldsymbol{g} * \boldsymbol{u}_i, \tilde{\boldsymbol{\Delta}}_B \rangle = 0$ for $i \in [0, n)$. Therefore, we obtain that $\langle \boldsymbol{g} * \boldsymbol{e}_i, \tilde{\boldsymbol{\Delta}}_B \rangle = 0$ for all $i \in [0, n)$. If there exists two different $\Delta, \Delta' \in S_\Delta$ such that $\langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{\Delta}_B \rangle \neq \langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{\Delta}'_B \rangle$ for some $i \in [0, n)$ where $\langle \boldsymbol{g}, \boldsymbol{\Delta}_B \rangle = \Delta$ and $\langle \boldsymbol{g}, \boldsymbol{\Delta}'_B \rangle = \Delta'$, then we define $\tilde{\boldsymbol{\Delta}}_B := \boldsymbol{\Delta}_B - \boldsymbol{\Delta}'_B$ and have that $\tilde{\Delta} = \langle \boldsymbol{g}, \tilde{\boldsymbol{\Delta}}_B \rangle \in \tilde{S}_{\tilde{\Delta}}$ and $\langle \boldsymbol{g} * \boldsymbol{e}_i, \tilde{\boldsymbol{\Delta}}_B \rangle \neq 0$. This is contradict with $\langle \boldsymbol{g} * \boldsymbol{e}_i, \tilde{\boldsymbol{\Delta}}_B \rangle = 0$. This concludes that $\langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{\Delta}_B \rangle$ is a unique value for all possible $\Delta = \langle \boldsymbol{g}, \boldsymbol{\Delta}_B \rangle \in S_\Delta$, and can be computed by the simulator using any $\Delta \in S_\Delta$. In the real protocol execution, $\mathcal{A}$ can compute $w'_i := w_i - \langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{\Delta}_B \rangle$ for $i \in [0, n)$ just as that computed by $\mathcal{S}$. Together with that

$w_i = v_i + \langle \boldsymbol{g} * \boldsymbol{u}_i, \boldsymbol{\Delta}_B \rangle$, we have that

$$w'_i = v_i + \langle \boldsymbol{g} * \boldsymbol{u}_i, \boldsymbol{\Delta}_B \rangle - \langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{\Delta}_B \rangle = v_i + \Delta \cdot u_i.$$

We now discuss the case of a prime $p > 2$. The main difference from the case of $p = 2$ is that the canonical maps between $\Delta \in \mathbb{F}_{p^r}$ and $\boldsymbol{\Delta}_B \in \{0,1\}^{rm}$ are not bijective. This implies that the solutions of equations (2.1) and (2.2) are not necessarily vectors of bits rather than elements of $\mathbb{F}_p$. Following the proof of [75, Lemma 2], we have that if $\tilde{S}_f$ includes at least two vectors that only consist of bits, which is necessary for the adversary to pass the correlation check with probability more than $1/p^r$, then it has dimension at least 1 for all $f \in [l]$. We also have the fact that $\tilde{S}_{\tilde{\Delta}}$ has dimension at most $r \log p$ and $\tilde{S}_f \cap \tilde{S}_{f'} = \{0\}$ for $f \neq f' \in [l]$ by definition. Together, we obtain that $l \leq r \log p$ as above.

Overall, we have that no environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution, except with probability at most $(r \log p)^2 / p^r$.

Malicious $\mathsf{P_B}$. $\mathcal{S}$ is given access to $\mathcal{F}_{\mathsf{LsVOLE}}^{p,r}$, and interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COPEe}}^{p,r}$, and receives the values Δ, v_i for $i \in [0, n)$ and b_h for $h \in [0, r)$ from $\mathcal{A}$.

(2) After receiving coefficients $\chi_1, \ldots, \chi_n \in \mathbb{F}_{p^r}$, $\mathcal{S}$ samples $x \leftarrow\!\!\$\, \mathbb{F}_{p^r}$, computes $y := \sum_{i=1}^{n} \chi_i \cdot v_i + \sum_{h=1}^{r} b_h \cdot \mathsf{X}^{h-1} \in \mathbb{F}_{p^r}$, and computes $z := y + x \cdot \Delta \in \mathbb{F}_{p^r}$. Then $\mathcal{S}$ sends (x, z) to adversary $\mathcal{A}$.

(3) $\mathcal{S}$ defines $\boldsymbol{v} = (v_1, \ldots, v_n)$ and sends $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ to functionality $\mathcal{F}_{\mathsf{LsVOLE}}^{p,r}$.

In the real protocol execution, the elements a_h for all $h \in [0, r)$ output by $\mathcal{F}_{\mathsf{COPEe}}^{p,r}$ are uniform in $\mathbb{F}_p$. Therefore, $\sum_{h=1}^{r} a_h \cdot \mathsf{X}^{h-1}$ is uniform in $\mathbb{F}_{p^r}$, and thus $x = \sum_{i=1}^{n} \chi_i \cdot u_i +$

$\sum_{h=1}^{r} a_h \cdot \mathsf{X}^{h-1}$ is uniformly random in $\mathbb{F}_{p^r}$. We obtain that the simulation is perfect. It is easy to see that the outputs of two parties have the same distribution between the real-world execution and the ideal-world execution.

Optimization. For many applications (e.g., our protocols) where learning the entire global key Δ is necessary in order to violate security of some higher-level protocol, it is unnecessary to eliminate the selective-failure leakage about Δ. This can be argued as follows. Assume the adversary guesses a set S (if there are multiple guesses then S is the intersection of all guessed sets) and is caught cheating if $\Delta \notin S$. The probability that the selective-failure attack is successful is $|S|/p^r$; conditioned on this event, the min-entropy of Δ is reduced to $\log |S|$. Therefore, the overall probability for the adversary to determine Δ is $|S|/p^r \cdot 2^{-\log |S|} = p^{-r}$, which is the same as the probability in the absence of any leakage. Similar observations have been used in secure-computation protocols [75, 36, 103].

2.2. Single-Point sVOLE

Single-point sVOLE is a variant of sVOLE where the vector of authenticated values contains exactly one nonzero entry. We present the associated functionality $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$ in Figure 2.5, where the vector length $n = 2^h$ is assumed to be a power of two for simplicity. In Figure 2.6, we present a protocol $\Pi_{\mathsf{spsVOLE}}^{p,r}$ that UC-realizes $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$ in the $(\mathcal{F}_{\mathsf{sVOLE}}^{p,r}, \mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}})$-hybrid model, where $\mathcal{F}_{\mathsf{OT}}$ is the standard OT functionality and $\mathcal{F}_{\mathsf{EQ}}$ corresponds to a weak equality test that reveals $\mathsf{P_A}$'s input to $\mathsf{P_B}$. Conceptually, the protocol can be divided into two steps: (1) the parties run a semi-honest protocol for generating a vector of authenticated values $[\boldsymbol{u}]$ having a single nonzero entry; then (2) a

Functionality $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$

Initialize: Upon receiving init from $\mathsf{P_A}$ and $\mathsf{P_B}$, sample $\Delta \leftarrow\$\, \mathbb{F}_{p^r}$ if $\mathsf{P_B}$ is honest and receive $\Delta \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store global key Δ, send Δ to $\mathsf{P_B}$, and ignore all subsequent init commands.

Extend: Upon receiving $(\mathsf{sp\text{-}extend}, n)$, where $n = 2^h$ for some $h \in \mathbb{N}$, from $\mathsf{P_A}$ and $\mathsf{P_B}$, do:

(1) If $\mathsf{P_B}$ is honest, sample $\boldsymbol{v} \leftarrow\$\, \mathbb{F}^n_{p^r}$. Otherwise, receive $\boldsymbol{v} \in \mathbb{F}^n_{p^r}$ from the adversary.
(2) If $\mathsf{P_A}$ is honest, then sample uniform $\boldsymbol{u} \in \mathbb{F}^n_p$ with exactly one nonzero entry, and compute $\boldsymbol{w} := \boldsymbol{v} + \Delta \cdot \boldsymbol{u} \in \mathbb{F}^n_{p^r}$. Otherwise, receive $\boldsymbol{u} \in \mathbb{F}^n_p$ (with at most one nonzero entry) and $\boldsymbol{w} \in \mathbb{F}^n_{p^r}$ from the adversary, and recompute $\boldsymbol{v} := \boldsymbol{w} - \Delta \cdot \boldsymbol{u} \in \mathbb{F}^n_{p^r}$.
(3) If $\mathsf{P_B}$ is corrupted, receive a set $I \subseteq [0, n)$ from the adversary. Let $\alpha \in [0, n)$ be the index of the nonzero entry of $\boldsymbol{u}$. If $\alpha \in I$, send success to $\mathsf{P_B}$ and continue. Otherwise, send abort to both parties and abort.
(4) Send $(\boldsymbol{u}, \boldsymbol{w})$ to $\mathsf{P_A}$ and $\boldsymbol{v}$ to $\mathsf{P_B}$.

Global-key query: If $\mathsf{P_A}$ is corrupted, receive $(\mathsf{guess}, \Delta')$ from the adversary with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send success to $\mathsf{P_A}$ and ignore any subsequent global-key query. Otherwise, send abort to both parties and abort.

Figure 2.5. **Functionality for single-point sVOLE.**

consistency check is performed to detect malicious behavior. We explain both steps in what follows.

$\mathsf{P_A}$ begins by choosing a uniform $\beta \in \mathbb{F}^*_p$ and a uniform index α. Letting $\boldsymbol{u} \in \mathbb{F}^n_p$ be the vector that is 0 everywhere except that $\boldsymbol{u}[\alpha] = \beta$, the goal is for the parties to generate $[\boldsymbol{u}]$. That is, they want $\mathsf{P_A}$ to hold $\boldsymbol{w} \in \mathbb{F}^n_{p^r}$ and $\mathsf{P_B}$ to hold $\boldsymbol{v} \in \mathbb{F}^n_{p^r}$ such that $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$. To do so, the parties begin by generating the authenticated value $[\beta]$; this is easy to do using a call to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. Next, they use a subroutine [27, 28, 26] based on the GGM construction [57] to enable $\mathsf{P_B}$ to generate $\boldsymbol{v} \in \mathbb{F}^n_{p^r}$ while allowing $\mathsf{P_A}$ to learn all the components of that vector except for $\boldsymbol{v}[\alpha]$. This is done in the following way. Let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ and $G' : \{0,1\}^\lambda \to \mathbb{F}^2_{p^r}$ be pseudorandom generators (PRGs). $\mathsf{P_B}$ chooses uniform $s \in \{0,1\}^\kappa$ and computes all nodes in a GGM tree of depth h with s at the root: That is, letting s^i_j denote the value at the jth node on the ith level of the

tree, $\mathsf{P_B}$ defines $s_0^0 := s$ and then for $i \in [1,h)$ and $j \in [0, 2^{i-1})$ computes $\left(s_{2j}^i, s_{2j+1}^i\right) := G(s_j^{i-1})$; finally, $\mathsf{P_B}$ computes a vector $\boldsymbol{v}$ at the leaves as $(\boldsymbol{v}[2j], \boldsymbol{v}[2j+1]) := G'(s_j^{h-1})$ for $j \in [0, 2^{h-1})$. Next, $\mathsf{P_B}$ lets K_0^i (resp., K_1^i) be the XOR of the values at the even (resp., odd) nodes on the ith level. (When $i = h$ we replace XOR with addition in $\mathbb{F}_{p^r}$.) We write

$$\left(\{v_j\}_{j\in[0,n)},\ \{(K_0^i, K_1^i)\}_{i\in[h]}\right) := \mathsf{GGM}(1^n, s)$$

to denote this computation done by $\mathsf{P_B}$. It is easily verified that if $\mathsf{P_A}$ is given $\{K_{\bar{\alpha}_i}^i\}_{i\in[h]}$ (where $\bar{\alpha}_i$ is the complement of the ith bit of α), then $\mathsf{P_A}$ can compute $\{\boldsymbol{v}[j]\}_{j\neq\alpha}$, while $\boldsymbol{v}[\alpha]$ remains computationally indistinguishable from uniform given $\mathsf{P_A}$'s view. We denote the resulting computation of $\mathsf{P_A}$ by $\{v_j\}_{j\neq\alpha} := \mathsf{GGM}'(\alpha, \{K_{\bar{\alpha}_i}^i\}_{i\in[h]})$. ($\mathsf{P_A}$ can obtain $\{K_{\bar{\alpha}_i}^i\}_{i\in[h]}$ using h OT invocations.)

Following the above, $\mathsf{P_A}$ sets $\boldsymbol{w}[i] := \boldsymbol{v}[i]$ for $i \neq \alpha$. Note that $\boldsymbol{w}[i] = \boldsymbol{v}[i] + \Delta \cdot \boldsymbol{u}[i]$ for $i \neq \alpha$ (since $\boldsymbol{u}[i] = 0$ for $i \neq \alpha$), so all that remains is for $\mathsf{P_A}$ to obtain the missing value $\boldsymbol{w}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$ (without revealing α, β to $\mathsf{P_B}$). Recall the parties already hold $[\beta]$, meaning that $\mathsf{P_A}$ holds $\mathsf{M}[\beta]$ and $\mathsf{P_B}$ holds $\mathsf{K}[\beta]$ with $\mathsf{M}[\beta] = \mathsf{K}[\beta] + \Delta \cdot \beta$. So if $\mathsf{P_B}$ sends $\mathsf{K}[\beta] - \sum_i \boldsymbol{v}[i]$, then $\mathsf{P_A}$ can compute the missing value as

$$\begin{aligned}\boldsymbol{w}[\alpha] &= \mathsf{M}[\beta] - (\mathsf{K}[\beta] - \textstyle\sum_i \boldsymbol{v}[i]) - \sum_{i\neq\alpha} \boldsymbol{v}[i] \\ &= \mathsf{M}[\beta] - \mathsf{K}[\beta] + \boldsymbol{v}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta.\end{aligned}$$

This completes the "semi-honest" portion of the protocol.

To verify correct behavior, we generalize the approach of Yang et al. [104] that applies only to the case $p = 2$. We want to verify that $\boldsymbol{w}[i] = \boldsymbol{v}[i]$ for $i \neq \alpha$, and $\boldsymbol{w}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$.

Protocol $\Pi^{p,r}_{\mathsf{spsVOLE}}$

Initialize: This procedure is executed only once.

- $\mathsf{P_A}$ and $\mathsf{P_B}$ send init to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns Δ to $\mathsf{P_B}$.

Extend: This procedure can be run multiple times. On input $n = 2^h$, the parties do:

(1) $\mathsf{P_A}$ and $\mathsf{P_B}$ send $(\mathsf{extend}, 1)$ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $(a, c) \in \mathbb{F}_p \times \mathbb{F}_{p^r}$ to $\mathsf{P_A}$ and $b \in \mathbb{F}_{p^r}$ to $\mathsf{P_B}$ such that $c = b + \Delta \cdot a$. Then, $\mathsf{P_A}$ samples $\beta \leftarrow\!\$\, \mathbb{F}_p^*$, sets $\delta := c$, and sends $a' := \beta - a \in \mathbb{F}_p$ to $\mathsf{P_B}$, who computes $\gamma := b - \Delta \cdot a'$. Note that $\delta = \gamma + \Delta \cdot \beta \in \mathbb{F}_{p^r}$, so the parties now hold $[\beta]$.

$\mathsf{P_A}$ samples $\alpha \leftarrow\!\$\, [0, n)$ and defines $\boldsymbol{u} \in \mathbb{F}_p^n$ as the vector that is 0 everywhere except $\boldsymbol{u}[\alpha] = \beta$.

(2) $\mathsf{P_B}$ samples $s \leftarrow\!\$\, \{0,1\}^\lambda$, runs $\mathsf{GGM}(1^n, s)$ to obtain $\big(\{v_j\}_{j\in[0,n)}, \{(K_0^i, K_1^i)\}_{i\in[h]}\big)$, and sets $\boldsymbol{v}[j] := v_j$ for $j \in [0, n)$. $\mathsf{P_A}$ lets $\bar{\alpha}_i$ be the complement of the ith bit of the binary representation of α. For $i \in [h]$, $\mathsf{P_A}$ sends $\bar{\alpha}_i \in \{0,1\}$ to $\mathcal{F}_{\mathsf{OT}}$ and $\mathsf{P_B}$ sends (K_0^i, K_1^i) to $\mathcal{F}_{\mathsf{OT}}$, which returns $K^i_{\bar{\alpha}_i}$ to $\mathsf{P_A}$. Then $\mathsf{P_A}$ runs $\{v_j\}_{j\neq\alpha} := \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i\in[h]})$.

(3) $\mathsf{P_B}$ sends $d := \gamma - \sum_{i\in[0,n)} \boldsymbol{v}[i] \in \mathbb{F}_{p^r}$ to $\mathsf{P_A}$. Then, $\mathsf{P_A}$ defines $\boldsymbol{w} \in \mathbb{F}_{p^r}^n$ as the vector with $\boldsymbol{w}[i] := v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] := \delta - \Big(d + \sum_{i\neq\alpha} \boldsymbol{w}[i]\Big)$. Note that $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u}$.

Consistency check:

(4) Both parties send (extend, r) to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{x}, \boldsymbol{z}) \in \mathbb{F}_p^r \times \mathbb{F}_{p^r}^r$ to $\mathsf{P_A}$ and $\boldsymbol{y}^* \in \mathbb{F}_{p^r}^r$ to $\mathsf{P_B}$ such that $\boldsymbol{z} = \boldsymbol{y}^* + \Delta \cdot \boldsymbol{x}$.

(5) $\mathsf{P_A}$ samples $\chi_i \leftarrow\!\$\, \mathbb{F}_{p^r}$ for $i \in [0, n)$, and writes $\chi_\alpha = \sum_{i=0}^{r-1} \chi_{\alpha,i} \cdot \mathsf{X}^i$. Let $\boldsymbol{\chi}_\alpha = (\chi_{\alpha,0}, \ldots, \chi_{\alpha,r-1}) \in \mathbb{F}_p^r$. $\mathsf{P_A}$ then computes $\boldsymbol{x}^* := \beta \cdot \boldsymbol{\chi}_\alpha - \boldsymbol{x} \in \mathbb{F}_p^r$ and sends $\big(\{\chi_i\}_{i\in[0,n)}, \boldsymbol{x}^*\big)$ to $\mathsf{P_B}$, who computes $\boldsymbol{y} := \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^* \in \mathbb{F}_{p^r}^r$.

(6) $\mathsf{P_A}$ computes $Z := \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_{\mathsf{A}} := \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] - Z \in \mathbb{F}_{p^r}$, while $\mathsf{P_B}$ computes $Y := \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_{\mathsf{B}} := \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] - Y \in \mathbb{F}_{p^r}$. Then $\mathsf{P_A}$ sends V_{A} to $\mathcal{F}_{\mathsf{EQ}}$, and $\mathsf{P_B}$ sends V_{B} to $\mathcal{F}_{\mathsf{EQ}}$. If either party receives false or abort from $\mathcal{F}_{\mathsf{EQ}}$, it aborts.

(7) $\mathsf{P_A}$ outputs $(\boldsymbol{u}, \boldsymbol{w})$ and $\mathsf{P_B}$ outputs $\boldsymbol{v}$.

Figure 2.6. **Single-point sVOLE protocol in the $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}})$-hybrid model.**

Intuitively, the parties do this by having $\mathsf{P_A}$ choose uniform $\chi_0, \ldots, \chi_{n-1} \in \mathbb{F}_{p^r}$ and then checking that

$$\sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] + \Delta \cdot \beta \cdot \chi_\alpha.$$

Of course, this must be done without revealing α, β to $\mathsf{P_B}$. To do so, $\mathsf{P_A}$ and $\mathsf{P_B}$ use $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ to compute $Z, Y \in \mathbb{F}_{p^r}$, respectively, such that $Z = Y + \Delta \cdot \beta \cdot \chi_\alpha$. (We discuss below how this is done.) They then use $\mathcal{F}_{\mathsf{EQ}}$ to check if $V_{\mathsf{A}} = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] - Z$ is equal to $V_{\mathsf{B}} = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] - Y$.

To complete the description, we show how the parties can generate Z, Y (held by $\mathsf{P_A}$, $\mathsf{P_B}$, respectively) such that $Z = Y + \Delta \cdot \beta \cdot \chi_\alpha$. (This is like an authenticated value $[\beta \cdot \chi_\alpha]$, but note that $\beta \cdot \chi_\alpha$ lies in $\mathbb{F}_{p^r}$ rather than $\mathbb{F}_p$.) $\mathsf{P_A}$ views $\chi_\alpha \in \mathbb{F}_{p^r}$ as $\boldsymbol{\chi}_\alpha = (\chi_{\alpha,0}, \ldots, \chi_{\alpha,r-1}) \in \mathbb{F}_p^r$ (i.e., $\chi_\alpha = \sum_{i \in [0,r)} \chi_{\alpha,i} \cdot \mathsf{X}^i$, where $\{\mathsf{X}^i\}_{i \in [0,r)}$ form a basis for $\mathbb{F}_{p^r}$ over $\mathbb{F}_p$), and then the parties use $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ to generate the vector of authenticated values $[\beta \cdot \boldsymbol{\chi}_\alpha]$. This means $\mathsf{P_A}$ holds $\boldsymbol{z}$ and $\mathsf{P_B}$ holds $\boldsymbol{y}$ such that $\boldsymbol{z} = \boldsymbol{y} + \Delta \cdot \beta \cdot \boldsymbol{\chi}_\alpha$. Let $Z = \sum_{i \in [0,r)} \boldsymbol{z}[i] \cdot \mathsf{X}^i$ and $Y = \sum_{i \in [0,r)} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. We have that

$$\begin{aligned} Z = \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i &= \sum_{i=0}^{r-1} (\boldsymbol{y}[i] + \Delta \cdot \beta \cdot \boldsymbol{\chi}_\alpha[i]) \cdot \mathsf{X}^i \\ &= \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i + \Delta \cdot \beta \cdot \sum_{i=0}^{r-1} \boldsymbol{\chi}_\alpha[i] \cdot \mathsf{X}^i \\ &= Y + \Delta \cdot \beta \cdot \chi_\alpha, \end{aligned}$$

as desired.

We remark that this check allows a malicious $\mathsf{P_A}$ to guess Δ, and allows a malicious $\mathsf{P_B}$ to guess a subset in which the index α lies. (This will become evident in the proof of security.) Such guesses are incorporated into the ideal functionality $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$.

We now formally prove security of the protocol.

Theorem 2. *If G and G' are pseudorandom generators, then $\Pi^{p,r}_{\mathsf{spsVOLE}}$ UC-realizes $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ in the $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}})$-hybrid model. In particular, no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution except with probability at most $1/p^r + \mathsf{negl}(\lambda)$.*

We first consider the case of a malicious $\mathsf{P_A}$ and then consider the case of a malicious $\mathsf{P_B}$. In each case, we construct a PPT simulator $\mathcal{S}$ given access to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ that runs the PPT adversary $\mathcal{A}$ as a subroutine, and emulates functionalities $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, and $\mathcal{F}_{\mathsf{EQ}}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

Malicious $\mathsf{P_A}$. Every time the extend procedure is run (on input n), $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ and records the values (a, c) that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. When $\mathcal{A}$ sends the message $a' \in \mathbb{F}_p$, then $\mathcal{S}$ sets $\beta := a' + a \in \mathbb{F}_p$ and $\delta := c$.

(2) For $i \in [1, h)$, $\mathcal{S}$ samples $K^i \leftarrow \{0,1\}^\lambda$; it also samples $K^h \leftarrow \mathbb{F}_{p^r}$. Then for $i \in [h]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OT}}$ by receiving $\bar{\alpha}_i \in \{0,1\}$ from $\mathcal{A}$, and returning $K^i_{\bar{\alpha}_i} := K^i$ to $\mathcal{A}$. It sets $\alpha := \alpha_1 \cdots \alpha_h$ and defines $\boldsymbol{u} \in \mathbb{F}_p^n$ as the vector that is 0 everywhere except that $\boldsymbol{u}[\alpha] := \beta$. Next, $\mathcal{S}$ computes $\{v_j\}_{j \neq \alpha} := \mathsf{GGM'}(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i \in [h]})$.

(3) $\mathcal{S}$ picks $d \leftarrow\!\!\$\, \mathbb{F}_{p^r}$ and sends it to $\mathcal{A}$. Then, $\mathcal{S}$ defines $\boldsymbol{w}$ as the vector of length n with $\boldsymbol{w}[i] := v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] := \delta - \big(d + \sum_{i \neq \alpha} \boldsymbol{w}[i]\big)$.

(4) $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ by recording $(\boldsymbol{x}, \boldsymbol{z})$ from $\mathcal{A}$.

(5) $\mathcal{S}$ receives $\{\chi_i\}_{i \in [0,n)}$ and $\boldsymbol{x}^* \in \mathbb{F}_p^r$ from $\mathcal{A}$, and sets $\boldsymbol{x'} := \boldsymbol{x}^* + \boldsymbol{x} \in \mathbb{F}_p^r$ and $x' := \sum_{i=0}^{r-1} \boldsymbol{x'}[i] \cdot \mathsf{X}^i$.

(6) $\mathcal{S}$ records $V_{\mathsf{A}} \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{EQ}}$. It then computes $V'_{\mathsf{A}} := \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and does:

- If $x' = \beta \cdot \chi_\alpha$, then $\mathcal{S}$ checks whether $V_{\mathsf{A}} = V'_{\mathsf{A}}$. If so, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. Otherwise, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.
- Otherwise, $\mathcal{S}$ computes $\Delta' := (V'_{\mathsf{A}} - V_{\mathsf{A}}) / (\beta \cdot \chi_\alpha - x') \in \mathbb{F}_{p^r}$ and sends a global-key query $(\mathsf{guess}, \Delta')$ to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. If $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ returns $\mathsf{success}$, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. Otherwise, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.

(7) Whenever $\mathcal{A}$ sends a global-key query $(\mathsf{guess}, \tilde{\Delta})$ to functionality $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, $\mathcal{S}$ forwards the query to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ and returns the answer to $\mathcal{A}$. If the answer is abort, $\mathcal{S}$ aborts.

In the above simulation, if $\mathcal{A}$ succeeds to guess Δ, then $\mathcal{S}$ simulates the $\mathcal{A}$'s view using Δ without making any further global-key query to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$.

We claim that the joint distribution of the view of $\mathcal{A}$ and the output of the honest $\mathsf{P_B}$ in the ideal-world execution above is computationally indistinguishable from their distribution in the real-world execution. By the standard analysis of the GGM construction, it is not hard to see that d and the $\{K^i_{\tilde{\alpha}_i}\}$ sent to $\mathcal{A}$ in the above simulation, as well as the vector $\boldsymbol{v}$ that would be output by $\mathsf{P_B}$ when it does not abort, are computationally indistinguishable from the corresponding values in the real protocol execution. It thus only remains to analyze steps 4–6, which determine whether $\mathsf{P_B}$ aborts.

Let $\beta = a' + a$, $\boldsymbol{x}' = \boldsymbol{x}^* + \boldsymbol{x}$, and $x' = \sum_{i=0}^{r-1} \boldsymbol{x}'[i] \cdot \mathsf{X}^i$, as above. (Note that $a', a, \boldsymbol{x}^*, \boldsymbol{x}$ are well-defined in the real-world execution as well.) In the real-world execution, $\mathsf{P_B}$ computes

$$\begin{aligned}
V_{\mathsf{B}} &= \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] - \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i \\
&= \sum_{i \neq \alpha} \chi_i \cdot \boldsymbol{v}[i] + \chi_\alpha \cdot \boldsymbol{v}[\alpha] - \sum_{i=0}^{r-1} (\boldsymbol{z}[i] - \Delta \cdot \boldsymbol{x}'[i]) \cdot \mathsf{X}^i \\
&= \sum_{i \neq \alpha} \chi_i \cdot \boldsymbol{v}[i] + \chi_\alpha \cdot (\delta - \Delta \cdot \beta - d - \sum_{i \neq \alpha} \boldsymbol{v}[i]) \\
&\quad - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i + \Delta \cdot x' \\
&= \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i - \Delta \cdot (\beta \cdot \chi_\alpha - x') \\
&= V'_{\mathsf{A}} - \Delta \cdot (\beta \cdot \chi_\alpha - x').
\end{aligned}$$

where $\boldsymbol{w}$ and V'_{A} are defined as in the description of $\mathcal{S}$ above. Say that $\mathcal{A}$ sends V_{A} to $\mathcal{F}_{\mathsf{EQ}}$. If $x' = \beta \cdot \chi_\alpha$ (as will be the case when $\mathcal{A}$ behaves honestly), then $\mathcal{F}_{\mathsf{EQ}}$ returns true iff $V_{\mathsf{A}} = V'_{\mathsf{A}}$. Otherwise, $\mathcal{F}_{\mathsf{EQ}}$ returns true iff $\Delta = (V'_{\mathsf{A}} - V_{\mathsf{A}})/(\beta \cdot \chi_\alpha - x')$. We thus see that the ideal-world behavior of $\mathcal{F}_{\mathsf{EQ}}$ matches what would occur in the real world.

Malicious $\mathsf{P_B}$. Simulator $\mathcal{S}$ interacts with $\mathcal{A}$ as follows. First, $\mathcal{S}$ simulates the initialization step by recording the global key $\Delta \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$. Then, every time the extend procedure is executed (on input n), $\mathcal{S}$ does:

(1) $\mathcal{S}$ records $b \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$. Then $\mathcal{S}$ samples $a' \leftarrow\!\!\$\, \mathbb{F}_p$ and sends it to $\mathcal{A}$. Next, $\mathcal{S}$ computes $\gamma := b - \Delta \cdot a'$, and then samples $\beta \leftarrow\!\!\$\, \mathbb{F}_p^*$ and sets $\delta := \gamma + \Delta \cdot \beta$.

(2) $\mathcal{S}$ records the values $\{(K_0^i, K_1^i)\}_{i\in[h]}$ sent to $\mathcal{F}_{\mathsf{OT}}$ by $\mathcal{A}$.

(3) $\mathcal{S}$ receives $d \in \mathbb{F}_{p^r}$ from $\mathcal{A}$. Then, for each $\alpha \in [0, n)$, it computes a vector $\boldsymbol{w}_\alpha$ as follows:

   (a) Execute $\{v_j^\alpha\}_{j\neq\alpha} := \mathsf{GGM}'(\alpha, \{K_{\tilde{\alpha}_i}^i\}_{i\in[h]})$ and set $\boldsymbol{w}_\alpha[i] = v_i^\alpha$ for $i \neq \alpha$.

   (b) Compute $\boldsymbol{w}_\alpha[\alpha] := \delta - (d + \sum_{i\neq\alpha} \boldsymbol{w}_\alpha[i])$.

(4) $\mathcal{S}$ records the vector $\boldsymbol{y}^*$ sent to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by $\mathcal{A}$.

(5) $\mathcal{S}$ samples $\chi_i \leftarrow\!\!\$\, \mathbb{F}_{p^r}$ for $i \in [0, n)$ and $\boldsymbol{x}^* \leftarrow\!\!\$\, \mathbb{F}_p^r$, and sends them to $\mathcal{A}$. Then $\mathcal{S}$ computes $\boldsymbol{y} := \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^*$.

(6) $\mathcal{S}$ computes $Y := \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. It then records V_{B} sent to $\mathcal{F}_{\mathsf{EQ}}$ by $\mathcal{A}$. Next, $\mathcal{S}$ computes a set $I \subseteq [0, n)$ as follows:

   (a) For $\alpha \in [0, n)$, compute $V_{\mathsf{A}}^\alpha := \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}_\alpha[i] - \Delta \cdot \beta \cdot \chi_\alpha - Y$.

   (b) Define $I := \{\alpha \in [0, n) \mid V_{\mathsf{A}}^\alpha = V_{\mathsf{B}}\}$.

   $\mathcal{S}$ sends I to $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$; if it returns abort, $\mathcal{S}$ picks $\tilde{\alpha} \leftarrow\!\!\$\, [0, n)\backslash I$, sends $\left(\mathsf{false}, V_{\mathsf{A}}^{\tilde{\alpha}}\right)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{EQ}}$, and then aborts. Otherwise, $\mathcal{S}$ sends $(\mathsf{true}, V_{\mathsf{B}})$ to $\mathcal{A}$.

(7) $\mathcal{S}$ chooses an arbitrary $\alpha \in I$ and computes a vector $\boldsymbol{v}$ as follows:

   (a) Set $\boldsymbol{v}[i] := \boldsymbol{w}_\alpha[i]$ for $i \in [0, n), i \neq \alpha$.

   (b) Set $\boldsymbol{v}[\alpha] := \gamma - d - \sum_{i\neq\alpha} \boldsymbol{v}[i]$.

   $\mathcal{S}$ sends $\boldsymbol{v}$ to $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$ and outputs whatever $\mathcal{A}$ outputs.

We first consider the view of adversary $\mathcal{A}$ in the ideal-world execution and the real-world execution. The values a' and $\boldsymbol{x}^*$ simulated by $\mathcal{S}$ have the same distribution as the real values, which are masked by a uniform element/vector output by $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$. The set I extracted by $\mathcal{S}$ corresponds to the selective failure attack on the output index α^* of P_{A}. If $\mathcal{S}$ receives abort from $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$, we have that $\alpha^* \notin I$. In the real protocol execution,

if $V_{\mathsf{B}} \neq V_{\mathsf{A}}^{\alpha^*}$, then P_{A} aborts. By previous considerations, this is equivalent to $\alpha^* \notin I$. Therefore, $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$ aborts if and only if the real protocol execution aborts. For an honest P_{A}, the index $\alpha^* \in [0, n)$ is sampled uniformly in both the real-world execution and the ideal-world execution. If receiving abort from $\mathcal{F}_{\mathsf{spsVOLE}}^{p,r}$, then $\mathcal{S}$ needs to send false along with an element $V_{\mathsf{A}}^{\tilde{\alpha}} \neq V_{\mathsf{B}}$ to $\mathcal{A}$. Although $\mathcal{S}$ does not know the actual index α^*, it can sample a random index $\tilde{\alpha}$ from the set $[0, n)\backslash I$ and send $V_{\mathsf{A}}^{\tilde{\alpha}}$ to $\mathcal{A}$. In the case of aborting, this simulation is perfect, since $\mathcal{Z}$ cannot obtain the output of P_{A} due to aborting, and the dummy index $\tilde{\alpha}$ has the same distribution as the actual index α^* under the condition that I is an incorrect guess.

Overall, we have that the adversary's view is perfectly indistinguishable between the real-world execution and the ideal-world execution. Below, we prove that except with probability $1/p^r$, the distribution of P_{A}'s output in the real-world execution is the same as that in the ideal-world execution. It is easy to see that the output vector $\boldsymbol{u}^*$ that is 0 everywhere except that $\boldsymbol{u}^*[\alpha^*] = \beta^*$ in the ideal-world execution and the real-world execution have the same distribution, from the above analysis and that β^* is perfectly hidden. In the following, we focus on proving the indistinguishability of $\boldsymbol{w}^*$ output by P_{A} between the ideal-world execution and the real-world execution. Firstly, we prove that the vector $\boldsymbol{v} \in \mathbb{F}_{p^r}^n$ computed by $\mathcal{S}$ in the step 7 is unique (i.e., independent of the choice $\alpha \in I$).

Claim 1. *For any $\alpha, \alpha' \in [0, n)$, let $\boldsymbol{v}_\alpha, \boldsymbol{v}_{\alpha'}$ be the vectors computed by $\mathcal{S}$ with α, α' following the step 7, then we have*

$$\Pr\left\{\boldsymbol{v}_\alpha \neq \boldsymbol{v}_{\alpha'} \;\middle|\; V_{\mathsf{A}}^{\alpha} = V_{\mathsf{A}}^{\alpha'}\right\} \leq \frac{1}{p^r}.$$

Proof. Since $V_{\mathsf{A}}^{\alpha} = V_{\mathsf{A}}^{\alpha'}$, we have

$$\sum_{i\in[0,n)} \chi_i \cdot \boldsymbol{w}_\alpha[i] - \Delta\cdot\beta\cdot\chi_\alpha - Y = \sum_{i\in[0,n)} \chi_i \cdot \boldsymbol{w}_{\alpha'}[i] - \Delta\cdot\beta\cdot\chi_{\alpha'} - Y \Leftrightarrow$$

$$\sum_{i\neq\alpha,\alpha'} \chi_i \cdot (\boldsymbol{w}_\alpha[i] - \boldsymbol{w}_{\alpha'}[i]) + \chi_\alpha \cdot (\boldsymbol{w}_\alpha[\alpha] - \boldsymbol{w}_{\alpha'}[\alpha] - \Delta\cdot\beta) + \chi_{\alpha'} \cdot (\boldsymbol{w}_\alpha[\alpha'] - \boldsymbol{w}_{\alpha'}[\alpha'] + \Delta\cdot\beta) = 0.$$

Note that Δ, β, $\boldsymbol{w}_\alpha$ and $\boldsymbol{w}_{\alpha'}$ have already been defined before $\{\chi_i\}_{i\in[0,n)}$ are sampled. Furthermore, each coefficient χ_i is uniform. Therefore, except with probability $1/p^r$, we have:

$$\boldsymbol{w}_\alpha[i] = \boldsymbol{w}_{\alpha'}[i] \text{ for } i \in [0,n), i \neq \alpha, \alpha',$$

$$\boldsymbol{w}_\alpha[\alpha] - \boldsymbol{w}_{\alpha'}[\alpha] = \boldsymbol{w}_{\alpha'}[\alpha'] - \boldsymbol{w}_\alpha[\alpha'] = \Delta\cdot\beta.$$

From the first equation, we directly obtain that $\boldsymbol{v}_\alpha[i] = \boldsymbol{v}_{\alpha'}[i]$ for $i \neq \alpha, \alpha'$. From the definitions of $\boldsymbol{w}_\alpha[\alpha]$ and $\boldsymbol{v}_\alpha[\alpha]$, we have that $\boldsymbol{v}_\alpha[\alpha] = \boldsymbol{w}_\alpha[\alpha] - \Delta\cdot\beta$. Together with $\boldsymbol{w}_\alpha[\alpha] = \boldsymbol{w}_{\alpha'}[\alpha] + \Delta\cdot\beta$, we further have that $\boldsymbol{v}_\alpha[\alpha] = \boldsymbol{w}_{\alpha'}[\alpha] = \boldsymbol{v}_{\alpha'}[\alpha]$. Similarly we also have $\boldsymbol{v}_{\alpha'}[\alpha'] = \boldsymbol{v}_\alpha[\alpha']$. □

Let $\boldsymbol{w}^*, \boldsymbol{u}^*$ be the output of P_A and $\boldsymbol{v}$ be the input from $\mathcal{S}$ (or P_B). It is obvious that $\boldsymbol{w}^* = \boldsymbol{v} + \Delta\cdot\boldsymbol{u}^*$ in the ideal-world execution. Now we look at the real-world execution. We define a vector $\boldsymbol{v}^*$ as $\boldsymbol{v}^*[i] = \boldsymbol{w}_{\alpha^*}[i]$ for $i \neq \alpha^*$ and $\boldsymbol{v}^*[\alpha^*] = \gamma - d - \sum_{i\neq\alpha^*} \boldsymbol{v}^*[i]$, where recall that α^* is the output index of P_A. From $\boldsymbol{w}_{\alpha^*}[\alpha^*] = \gamma + \Delta\cdot\beta^* - (d + \sum_{i\neq\alpha^*} \boldsymbol{w}_{\alpha^*}[i])$, we have that $\boldsymbol{w}_{\alpha^*}[\alpha^*] = \boldsymbol{v}^*[\alpha^*] + \Delta\cdot\beta^*$. Therefore, we obtain that $\boldsymbol{w}^* = \boldsymbol{v}^* + \Delta\cdot\boldsymbol{u}^*$ where $\boldsymbol{w}^* = \boldsymbol{w}_{\alpha^*}$. Note that $\boldsymbol{v}^*$ in both the ideal-world execution and the real-world execution are defined in the identical way, and thus have the same distribution. Based on Claim 1, we know that

in the ideal-world execution, $\boldsymbol{v}^*$ is indistinguishable from $\boldsymbol{v}$ computed by $\mathcal{S}$, except with probability at most $1/p^r$. Therefore $\boldsymbol{v}$ in the ideal-world execution is indistinguishable from $\boldsymbol{v}^*$ in the real-world execution, which implies the indistinguishability of the output of $\mathsf{P_A}$ in the ideal world and the real world.

Optimizations. We discuss various optimizations of the protocol shown in Figure 2.6:

(1) For large p (i.e., $\log p \geq \rho$), the parties can use the output of $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ directly as $[\beta]$ in step 1 of protocol $\Pi_{\mathsf{spsVOLE}}^{p,r}$, since $\beta \neq 0$ with overwhelming probability.

(2) In the consistency check, $\mathsf{P_A}$ can send uniform $\mathsf{seed} \in \{0,1\}^\lambda$ to $\mathsf{P_B}$, who then derives the $\{\chi_i\}$ from seed using a hash function modeled as a random oracle.

(3) When t extend executions are needed, we can batch the consistency checks using the ideas of Yang et al. [104] to reduce the total number of sVOLE correlations needed from $t \cdot (1+r)$ to $t+r$. The approach is as follows:

   (a) After t executions of the semi-honest portion of the extend phase, the parties hold $\{(\boldsymbol{u}_j, \boldsymbol{w}_j)\}_{j=1}^t$ and $\{\boldsymbol{v}_j\}_{j=1}^t$, respectively, where for all $j \in [t]$ we have $\boldsymbol{w}_j = \boldsymbol{v}_j + \Delta \cdot \boldsymbol{u}_j$ with $\boldsymbol{u}_j$ a vector that is 0 everywhere except $\boldsymbol{u}_j[\alpha_j] = \beta_j$. Then $\mathsf{P_A}$ and $\mathsf{P_B}$ send (extend, r) to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, which returns $(\boldsymbol{x}, \boldsymbol{z})$ to $\mathsf{P_A}$ and $\boldsymbol{y}^*$ to $\mathsf{P_B}$.

   (b) For $j \in [t]$, $\mathsf{P_A}$ samples $\chi_{i,j} \leftarrow\!\!\$\, \mathbb{F}_{p^r}$ for $i \in [0,n)$, and views $\chi_{\alpha_j,j}$ as the vector $\boldsymbol{\chi}_{\alpha_j,j} \in \mathbb{F}_p^r$. It then computes $\boldsymbol{x}^* := \sum_{j\in[t]} \beta_j \cdot \boldsymbol{\chi}_{\alpha_j,j} - \boldsymbol{x}$ and sends $\{\chi_{i,j}\}_{i\in[0,n),j\in[t]}$ and $\boldsymbol{x}^*$ to $\mathsf{P_B}$, who computes $\boldsymbol{y} := \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^* \in \mathbb{F}_{p^r}^r$.

   (c) $\mathsf{P_A}$ computes $V_{\mathsf{A}} := \sum_{i=0}^{n-1} \sum_{j=1}^{t} \chi_{i,j} \cdot \boldsymbol{w}_j[i] - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i$; $\mathsf{P_B}$ computes $V_{\mathsf{B}} := \sum_{i=0}^{n-1} \sum_{j=1}^{t} \chi_{i,j} \cdot \boldsymbol{v}_j[i] - \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. Then both parties check whether $V_{\mathsf{A}} = V_{\mathsf{B}}$ by calling $\mathcal{F}_{\mathsf{EQ}}$.

2.3. sVOLE Extension

We show here a protocol that can be viewed as a means of performing "sVOLE extension." That is, our protocol allows two parties to efficiently extend a small number of sVOLE correlations (created in a setup phase) to an arbitrary polynomial number of sVOLE correlations. The protocol relies on spsVOLE as a subroutine, as well as a variant of the LPN assumption that has been used in prior work [65, 25, 104].

Protocol overview. The parties use the base-sVOLE protocol to generate a length-k vector of authenticated values $[\boldsymbol{u}]$. They also use spsVOLE to generate t vectors of authenticated values, each of length n/t and having a single nonzero entry; they let $[\boldsymbol{e}]$ be the concatenation of those vectors. The parties then use a public matrix $\mathbf{A}$ to define the length-n vector of authenticated values $[\boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e}]$; by the LPN assumption, the corresponding values (which $\mathsf{P_A}$ knows) will appear pseudorandom to $\mathsf{P_B}$. This provides a way to extend k random sVOLE correlations to n pseudorandom sVOLE correlations once. As in prior work [104], however, we can generate $\ell = n - k$ correlations as many times as desired by simply using this idea to generate n sVOLE correlations and reserving the first k of those correlations for the next iteration of the extend phase.

LPN assumption. Let $\mathcal{D}_{n,t}$ denote the distribution over an error vector $\boldsymbol{e} \in \mathbb{F}_p^n$ in which $\boldsymbol{e}$ is divided into t blocks (each of length n/t), and each block of contains exactly one uniform nonzero entry at a uniform location within that block.

Definition 1 (LPN with static leakage [25])**.** *Let $\mathcal{G}$ be a polynomial-time algorithm that on input $1^k, 1^n, p$ outputs $\mathbf{A} \in \mathbb{F}_p^{k \times n}$. Let parameters k, n, t be implicit functions of security parameter κ. We say that the $\mathsf{LPN}^{\mathcal{G}}_{k,n,t,p}$ assumption holds if for all PPT*

algorithms $\mathcal{A}$ *we have*

$$\left| \Pr[\mathsf{LPN\text{-}Succ}^{\mathcal{G}}_{\mathcal{A}}(\kappa) = 1] - 1/2 \right| \leq \mathsf{negl}(\kappa),$$

where the experiment $\mathsf{LPN\text{-}Succ}^{\mathcal{G}}_{\mathcal{A}}(\kappa)$ *is defined as follows:*

(1) Sample $\mathbf{A} \leftarrow\!\!\$\, \mathcal{G}(1^k, 1^n, p)$, $\boldsymbol{u} \leftarrow\!\!\$\, \mathbb{F}_p^k$, *and* $\boldsymbol{e} \leftarrow\!\!\$\, \mathcal{D}_{n,t}$. *Let* $\alpha_1, \ldots, \alpha_t$ *be the indices of the nonzero entries in* $\boldsymbol{e}$ *(each of which is located in a disjoint block of length* n/t*).*

(2) $\mathcal{A}$ *outputs* t *subsets* $I_1, \ldots, I_t \subseteq [0, n)$. *If* $\alpha_i \in I_i$ *for all* $i \in [t]$, *then send* $\mathsf{success}$ *to* $\mathcal{A}$*; otherwise, abort the experiment and define* $b' := 0$.

(3) Pick $b \leftarrow\!\!\$\, \{0, 1\}$. *If* $b = 0$, *let* $\boldsymbol{x} := \boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e}$*; otherwise, sample* $\boldsymbol{x} \leftarrow\!\!\$\, \mathbb{F}_p^n$. *Send* $\boldsymbol{x}$ *to* $\mathcal{A}$, *who then outputs a bit* b' *(if the experiment did not abort).*

(4) The experiment outputs 1 iff $b' = b$.

Protocol description. In Figure 2.7, we present our sVOLE extension protocol in the $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}^{p,r}_{\mathsf{spsVOLE}})$-hybrid model. For simplicity, we assume a public matrix $\mathbf{A} \in \mathbb{F}_p^{k \times n}$, output by an efficient algorithm $\mathcal{G}(1^k, 1^n, p)$, that is fixed at the outset of the protocol. (It is also possible to have $\mathsf{P_A}$ generate $\mathbf{A}$ and then send it to $\mathsf{P_B}$.) We assume that $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ and $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ share the same initialization (i.e., use the same global key Δ). This holds, in particular, when we use protocol $\Pi^{p,r}_{\mathsf{spsVOLE}}$ from the previous section to UC-realize $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$.

Theorem 3. *If the* $\mathsf{LPN}^{\mathcal{G}}_{k,n,t,p}$ *assumption holds, then* $\Pi^{p,r}_{\mathsf{sVOLE}}$ *UC-realizes* $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ *in the* $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}^{p,r}_{\mathsf{spsVOLE}})$*-hybrid model.*

Proof. We first consider the case of a malicious $\mathsf{P_A}$ and then consider the case of a malicious $\mathsf{P_B}$. In each case, we construct a PPT simulator $\mathcal{S}$ given access to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ that

Protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$

Parameters: Fix n, k, t, and define $\ell = n - k$ and $m = n/t$. Let $\mathbf{A} \in \mathbb{F}_p^{k \times n}$ be a matrix output by $\mathcal{G}(1^k, 1^n, p)$.

Initialize: This procedure is executed only once.

(1) $\mathsf{P_A}$ and $\mathsf{P_B}$ send init to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $\Delta \in \mathbb{F}_{p^r}$ to $\mathsf{P_B}$.

(2) $\mathsf{P_A}$ and $\mathsf{P_B}$ send (extend, k) to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{u}, \boldsymbol{w})$ to $\mathsf{P_A}$ and $\boldsymbol{v}$ to $\mathsf{P_B}$ such that $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u} \in \mathbb{F}_{p^r}^k$.

Extend: This procedure can be executed multiple times.

(3) For $i \in [t]$, $\mathsf{P_A}$ and $\mathsf{P_B}$ send $(\mathsf{sp\text{-}extend}, m)$ to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$, which returns $(\boldsymbol{e}_i, \boldsymbol{c}_i)$ to $\mathsf{P_A}$ and $\boldsymbol{b}_i$ to $\mathsf{P_B}$ such that $\boldsymbol{c}_i = \boldsymbol{b}_i + \Delta \cdot \boldsymbol{e}_i \in \mathbb{F}_{p^r}^m$ and $\boldsymbol{e}_i \in \mathbb{F}_p^m$ has exactly one nonzero entry. If either party receives abort from $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ in any of these spsVOLE executions, it aborts.

(4) $\mathsf{P_A}$ defines $\boldsymbol{e} = (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_t) \in \mathbb{F}_p^n$ and $\boldsymbol{c} = (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t) \in \mathbb{F}_{p^r}^n$. Then $\mathsf{P_A}$ computes $\boldsymbol{x} := \boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e} \in \mathbb{F}_p^n$ and $\boldsymbol{z} := \boldsymbol{w} \cdot \mathbf{A} + \boldsymbol{c} \in \mathbb{F}_{p^r}^n$. $\mathsf{P_B}$ defines $\boldsymbol{b} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_t) \in \mathbb{F}_{p^r}^n$ and computes $\boldsymbol{y} := \boldsymbol{v} \cdot \mathbf{A} + \boldsymbol{b} \in \mathbb{F}_{p^r}^n$.

(5) $\mathsf{P_A}$ updates $\boldsymbol{u}, \boldsymbol{w}$ by setting $\boldsymbol{u} := \boldsymbol{x}[0 : k) \in \mathbb{F}_p^k$ and $\boldsymbol{w} := \boldsymbol{z}[0 : k) \in \mathbb{F}_{p^r}^k$, and outputs $(\boldsymbol{s}, \mathsf{M}[\boldsymbol{s}]) := (\boldsymbol{x}[k : n), \boldsymbol{z}[k : n)) \in \mathbb{F}_p^\ell \times \mathbb{F}_{p^r}^\ell$. $\mathsf{P_B}$ updates $\boldsymbol{v}$ by setting $\boldsymbol{v} := \boldsymbol{y}[0 : k) \in \mathbb{F}_{p^r}^k$, and outputs $\mathsf{K}[\boldsymbol{s}] := \boldsymbol{y}[k : n) \in \mathbb{F}_{p^r}^\ell$.

Figure 2.7. **The sVOLE extension protocol in the $(\mathcal{F}^{p,r}_{\mathsf{sVOLE}}, \mathcal{F}^{p,r}_{\mathsf{spsVOLE}})$-hybrid model.**

runs the adversary $\mathcal{A}$ as a subroutine, and emulates functionalities $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ and $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and $\mathcal{Z}$.

Malicious $\mathsf{P_A}$. $\mathcal{S}$ records the vectors $(\boldsymbol{u}, \boldsymbol{w}) \in \mathbb{F}_p^k \times \mathbb{F}_{p^r}^k$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ during initialization. Then in each iteration, $\mathcal{S}$ runs as follows:

(1) For $i \in [t]$, $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ and receives the value $\boldsymbol{e}_i \in \mathbb{F}_p^m$ (with at most one nonzero entry) and $\boldsymbol{c}_i \in \mathbb{F}_{p^r}^m$ from $\mathcal{A}$; it then defines $\boldsymbol{e} := (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_t) \in \mathbb{F}_p^n$ and $\boldsymbol{c} := (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t) \in \mathbb{F}_{p^r}^n$.

(2) $\mathcal{S}$ computes $\boldsymbol{x} := \boldsymbol{u} \cdot \mathbf{A} + \boldsymbol{e} \in \mathbb{F}_p^n$ and $\boldsymbol{z} := \boldsymbol{w} \cdot \mathbf{A} + \boldsymbol{c} \in \mathbb{F}_{p^r}^n$, and sends $\boldsymbol{x}[k : n) \in \mathbb{F}_p^\ell$ and $\boldsymbol{z}[k : n) \in \mathbb{F}_{p^r}^\ell$ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. It also locally updates $\boldsymbol{u} := \boldsymbol{x}[0 : k) \in \mathbb{F}_p^k$ and $\boldsymbol{w} := \boldsymbol{z}[0 : k) \in \mathbb{F}_{p^r}^k$ for the next iteration.

(3) If $\mathcal{A}$ ever makes a global key query Δ' to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$, then $\mathcal{S}$ forwards that query to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. If $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ responds with abort, $\mathcal{S}$ aborts; otherwise, it continues.

It is easy to see that the simulation provided by $\mathcal{S}$ is perfect.

Malicious $\mathsf{P_B}$. $\mathcal{S}$ runs $\mathcal{G}(1^k, 1^n, p)$ to generate $\mathbf{A} \in \mathbb{F}_p^{k\times n}$. During initialization, $\mathcal{S}$ records the values $\Delta \in \mathbb{F}_{p^r}$ and $\boldsymbol{v} \in \mathbb{F}_{p^r}^k$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, and sends Δ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. Then in each iteration, $\mathcal{S}$ runs as follows:

(1) For $i \in [t]$, $\mathcal{S}$ receives the value $\boldsymbol{b}_i \in \mathbb{F}_{p^r}^m$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$; it sets $\boldsymbol{b} := (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_t) \in \mathbb{F}_{p^r}^n$.

(2) For $i \in [t]$, $\mathcal{S}$ receives the set $I_i \subseteq [0, m)$ that $\mathcal{A}$ sends to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. Then $\mathcal{S}$ samples $\boldsymbol{e} \leftarrow \mathcal{D}_{n,t}$ and defines $\{\alpha_1, \ldots, \alpha_t\}$ to be the nonzero entries of $\boldsymbol{e}$. If $\alpha_i \bmod m \in I_i$ for all i, then $\mathcal{S}$ continues; otherwise, it aborts.

(3) $\mathcal{S}$ computes $\boldsymbol{y} := \boldsymbol{v} \cdot \mathbf{A} + \boldsymbol{b} \in \mathbb{F}_{p^r}^n$, and sends $\boldsymbol{y}[k : n) \in \mathbb{F}_{p^r}^\ell$ to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$. It also locally updates $\boldsymbol{v} := \boldsymbol{y}[0 : k) \in \mathbb{F}_{p^r}^k$ for the next iteration.

The view of $\mathcal{A}$ is simulated perfectly, and in both the ideal-world simulation and the ideal-world execution of the protocol the output $(\boldsymbol{s}, \mathsf{M}[\boldsymbol{s}])$ of $\mathsf{P_A}$ satisfies $\boldsymbol{y}[k, n) = \mathsf{M}[\boldsymbol{s}] - \Delta \cdot \boldsymbol{s}$. The difference is that in the ideal world $\boldsymbol{s}$ is uniform, whereas in the real world $\boldsymbol{s} = \boldsymbol{u}\cdot\mathbf{A}+\boldsymbol{e}$ for a uniform vector $\boldsymbol{u}$. It is not hard to see that this difference is undetectable if the $\mathsf{LPN}^{\mathcal{G}}_{k,n,t,p}$ assumption holds. □

Optimizations. In each iteration of the extend procedure, protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$ makes t calls to $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$. If $\mathcal{F}^{p,r}_{\mathsf{spsVOLE}}$ is instantiated by protocol $\Pi^{p,r}_{\mathsf{spsVOLE}}$ from Section 2.2, and we use the optimization described at the end of that section, the t calls to $\Pi^{p,r}_{\mathsf{spsVOLE}}$ require only $t + r$ calls to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$.

Moreover, we can push all the calls to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ into the initialization phase, so that the extend procedure does not invoke $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ at all. Specifically, if we make $n_0 = k+t+r$ calls to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ during initialization, we can run the extend procedure without any additional call to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$. Each time the extend procedure is run, we reserve n_0 of the sVOLE correlations that are produced for the following iteration, and output $n - n_0$ "usable" sVOLE correlations.

We can further optimize the generation of the initial set of n_0 sVOLE correlations during initialization. Let (k_0, n_0, t_0) be another set of LPN parameters. (Note that $n_0 \ll n$, so we can take $k_0 \ll k$ and $t_0 \approx t$ while achieving security comparable to what is achieved for the LPN parameters (n, k, t).) We then make $n_0' = k_0 + t_0 + r$ calls to the base-sVOLE protocol described in Section 2.1 to generate that number of sVOLE correlations, after which we run the extend procedure of $\Pi_{\mathsf{sVOLE}}^{p,r}$ once to obtain n_0 sVOLE correlations.

2.4. Performance Evaluation

In this section, we report on the performance of our sVOLE protocol. All our protocols were implemented and open-sourced in the EMP toolkit [94].

Parameter selection. As suggested in prior work [24, 87, 104], we choose the public LPN matrix $\mathbf{A}$ as a generator of a 10-local linear code, which means that each column of $\mathbf{A}$ contains exactly 10 (uniform) nonzero entries. This is advantageous since it means that computing each entry of $\boldsymbol{u} \cdot \mathbf{A}$ involves reading only 10 positions of $\boldsymbol{u} \in \mathbb{F}_p^k$. To ensure that reading those positions can be done quickly, we set k so that $\boldsymbol{u}$ fits in the L1 CPU cache (i.e., the size of $\boldsymbol{u}$ is less than 8 MB). With k fixed, for any choice of $n > k$ we

One-time setup			Extend execution		
k_0	n_0	t_0	k	n	t
19,870	642,048	2,508	589,760	10,805,248	1,319

Table 2.1. **LPN parameters used in our VOLE protocol.**

can take the smallest t for which all known attacks on the LPN problem require at least 2^{128} operations [24, 25]. When we apply the optimizations described at the end of the previous section to our protocol, we see that using LPN parameters (n, k, t) means that each invocation of the extend procedure results in $n-k-t-1$ usable VOLE correlations. We perform exhaustive search to find the smallest n so that $n - k - t - 1 \geq 10^7$. For the parameters of the setup phase, we follow the same step as above, except that we will ensure that $n_0 - k_0 - t_0 - 1 \geq k$. This results in the LPN parameters shown in Table 2.1.

2.4.1. VOLE over Large Fields.

In all our experiments, we use two Amazon EC2 instances of type `m5.4xlarge` with 16 vCPUs and 64 GB of RAM, using 5 threads. We artificially limit the network bandwidth as indicated in each experiment. All implementations achieve the statistical security parameter $\rho \geq 40$ and computational security parameter $\kappa = 128$.

We focus here on the performance of protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$ over large fields; specifically, we fix the Mersenne prime $p = 2^{61} - 1$ and set $r = 1$. (Since $r = 1$, sVOLE is equivalent to VOLE in this case.)

We evaluate the efficiency of protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$ in Table 2.2. The extend procedure requires very little communication (less than half a bit per usable VOLE correlation), and its execution time is largely unaffected by the network bandwidth above 100 Mbps.

	20 Mbps	50 Mbps	100 Mbps	500 Mbps	1 Gbps
Init. (*ms*)	1343	640	478	451	438
Extend (*ns*/VOLE)	101	87	85	85	85

Table 2.2. **Efficiency of our VOLE protocol as a function of network bandwidth.** The communication per VOLE correlation is 0.42 bits; the overall communication of the one-time setup is 1.1 MB.

	[87]	[40]	Ours (w/o setup)	Ours (w/ setup)
Communication (bits)	960	160	0.42	1.32
Execution time (*ns*)	2000	400	85	130

Table 2.3. **Our VOLE protocol vs. prior protocols.** We fix the network bandwidth to 500 Mbps and report the marginal cost per VOLE correlation. Running time for the protocol of Schoppmann et al. [87] is the time for communication alone; numbers for the protocol of Castro et al. [40] are taken from their paper and are based on the same network and CPU configuration but using 8 threads.

The one-time initialization only communicates 1.1 MB and takes roughly 478 milliseconds under a 100 Mbps network.

In Table 2.3, we compare our VOLE protocol with the best known protocols that have been implemented [87, 40]. Since our protocol needs an one-time setup, that can be amortized over multiple executions, we report our performance both without one-time setup (in case multiple extensions are executed), and the one with one-time setup (in case only one extension is executed). We fix the network bandwidth to 500 Mbps to match the experiments of Castro et al. [40]. Our protocol outperforms prior work even though prior work is secure only against *semi-honest* adversaries, whereas our protocol is secure in the malicious setting. Note in particular that the communication complexity of our protocol is orders of magnitude lower than prior work. Boyle et al. [25] also proposed a maliciously secure sVOLE protocol but only implemented their protocol for the special

case $p = 2, r = 128$. Based on their implementation in that case, we estimate that for our choice of p their protocol would communicate roughly 0.14 bits per sVOLE; however their computation is much heavier than ours and would take time at least 900 *ns* per VOLE correlation. Therefore, we believe that our protocol is still more efficient for most network bandwidth settings.

2.4.2. sVOLE over Field Extension

We focus here on the performance of protocol $\Pi^{p,r}_{\mathsf{sVOLE}}$ over a small field and its field extension; specifically, we fix $p = 2$ and set $r = 128$. We compare the performance of our protocols with the state-of-the art protocols for outputting COTs of 128-bit strings. For all experimental results, we use two Amazon EC2 machines of type `c5.4xlarge` with network bandwidth artificially limited. We use 5 threads for all implementations that we benchmarked.

We report the performance of our protocols in several different network settings and compare them with the state-of-the-art COT protocols including the optimized semi-honest IKNP OT extension [4], the maliciously secure KOS OT extension [74], and Boyle et al.'s two protocols [25] based on dual-LPN with a *regular* noise distribution. For the protocols other than ours, we use the libOTe library [86] for benchmark, but remove the last hashing on COT correlations so that the final output is correlated OT rather than random OT. We observe that this improves the running time of their protocols by roughly 15 *ms*.

We do not compare the semi-honest protocol by Schoppmann et al. [87] as they only implemented the VOLE protocol over a large field/ring rather than a COT protocol. We

Protocol	Comm./COT	10Mbps	50Mbps	100Mbps	500Mbps	1Gbps	5Gbps
Semi-Honest Security							
[4]	128 bits	128308	25704	12885	2627	1345	324
[25]	0.1 bits	1942	1961	1953	1966	1971	1966
Ferret-Uni	0.73 bits	821	306	262	264	262	261
Ferret-Reg	0.44 bits	536	215	176	159	158	160
Malicious Security							
[74]	128 bits	128314	25736	12924	2647	1387	344
[25]	0.1 bits	2113	2099	2091	2106	2083	2095
Ferret-Uni	0.73 bits	864	325	326	318	317	317
Ferret-Reg	0.44 bits	540	220	184	182	185	185

Table 2.4. **Comparison between our COT protocols and the state-of-the-art protocols.** All numbers reported are in milliseconds (ms) for computing 10^7 COTs. The one-time setup cost is not included.

estimate that our protocol is about 15× faster than theirs (without involving the one-time setup cost), since we improve the communication of their protocol by roughly 15× and also optimize the computation. The one-time setup cost of our protocol is larger than theirs, but the setup cost will be amortized to negligible when a huge number of COT correlations are needed. While Schoppmann et al.'s protocol [87] is in the semi-honest setting, our technique enables their protocol to obtain malicious security.

In Table 2.4, we evaluate the performance of different COT protocols in terms of communication and running time. Both of the IKNP-style OT extension protocols (IKNP [4] and KOS [74]) suffer from a high cost due to the high communication overhead. As shown in Table 2.4, the IKNP-style protocols need 128-bit communication per OT, while our protocol Ferret-Uni (resp., Ferret-Reg) needs only 0.73 bits (resp., 0.44 bits) per OT. Thus our protocols can achieve a huge performance gain (150×−40× for Ferret-Uni; 240×−70× for Ferret-Reg), when running in a network with restricted bandwidth (10Mbps−100Mbps).

Security	Protocol	Comm.	10Mbps	50Mbps	100Mbps	500Mbps	1Gbps	5Gbps
Semi-honest	Ferret-Uni	1.51 MB	1162	482	482	481	479	478
	Ferret-Reg	1.13 MB	811	183	106	41	35	30
Malicious	Ferret-Uni	1.51 MB	1166	486	485	484	485	484
	Ferret-Reg	1.13 MB	818	184	107	42	37	32

Table 2.5. **The efficiency for one-time setup of our COT protocols.** All numbers are in milliseconds (*ms*). For other protocols, the one-time setup takes about 30 *ms*.

Our protocol is also computationally cheaper than IKNP because our protocol does not need bit-matrix transposition required by IKNP-style protocols. Even when the network bandwidth is as high as 5 Gbps, Ferret-Reg is about 2× faster than IKNP [4] and KOS [74], and Ferret-Uni still outperforms the two IKNP-style protocols.

We also compare our COT protocols with the ones by Boyle et al. [25]. Their protocols are based on *dual-LPN with a regular noise distribution*, and thus have a small communication cost but a large computational overhead. Although our protocols require more communication, it is still faster than theirs even in slow network settings due to our high computational efficiency. We estimate the crossover point be around 2 Mbps, and that more computational resources will further bring up the crossover point. As a result, our protocol Ferret-Reg (using a similar LPN assumption) can improve the efficiency by a factor of $4\times-11\times$ under different network bandwidths.

We observe the overhead of strengthening semi-honest security to malicious security for these protocols when computing one correlated OT. While Boyle et al. [25] need an overhead of about 14 *ns*, our protocol Ferret-Reg only incurs an overhead of about 1 *ns*, which matches the overhead of KOS [74] and seems to be *optimal*.

One-time Setup. We also evaluate the performance in the one-time setup phase of our COT protocols. We take advantage of pre-processing OT to accelerate the extend processes when many OTs are required. The one-time setup generates M COTs (recall that $M = 616,092$ for Ferret-Uni and $M = 649,728$ for Ferret-Reg) by running an IKNP-style OT extension followed by a single COT iterative execution.

As shown in Table 2.5, the one-time setup takes at most 486 *ms* for Ferret-Uni and 184 *ms* for Ferret-Reg for any network with bandwidth at least 50 Mbps. When the network is faster than 500 Mbps, the running time of one-time setup is less than around 42 *ms* for Ferret-Reg. The IKNP-style protocols (IKNP [4] and KOS [74]) and the two protocols by Boyle et al. [25] only need a one-time setup of about 30 *ms*. However, the setup procedure needs to be performed *only once*, and then can be extended to generate *any polynomial number* of COTs as we want from the same setup. Therefore, if a great deal of COTs are computed via multiple iterations using the same setup, the one-time setup cost will become negligible in an amortized sense. When COT is used to construct MPC, the parties can execute the setup phase only once, and then generate the correlated OTs for many protocol executions. Moreover, even if COTs are generated in the preprocessing phase of MPC protocols, the setup process can only be executed once before the circuit size is known, and then the desired number of COTs are produced by iterative extensions after the circuit size is known by the parties. Due to these reasons, we optimize the parameters to improve the efficiency of our main iteration while keeping the one-time setup cost reasonable.

We also emphasize that even in the single-execution setting where the one-time setup is a part of the whole computation, the end-to-end performance of our COT protocols

Network Bandwidth	SPCOT all executions	LPN encoding	Ferret-Reg Total time	Ferret-Uni Extra time
10 Mbps	40	12	53	32
50 Mbps	9	12	22	10
100 Mbps	6	11	18	14

Table 2.6. **Microbenchmarks for our maliciously secure COT protocols.** All numbers are in nanoseconds (*ns*) and are the amortized time per COT correlation, without involving the one-time setup cost.

is still significantly better than prior work. In particular, our protocol Ferret-Uni (resp., Ferret-Reg) still improves the *end-to-end* efficiency by a factor of roughly $63\times-2\times$ (resp., $94\times-6\times$) when the network bandwidth is between 10 Mbps and 1 Gbps, compared to the state-of-the-art IKNP-style protocols. Furthermore, in terms of the *whole* efficiency, our protocol Ferret-Reg is about $5\times-9\times$ faster than Boyle et al.'s protocol, when the network bandwidth is between 50 Mbps and 5 Gbps.

Micro-benchmark. Table 2.6 shows the experimental results by micro-benchmarking our maliciously secure COT protocols under different network settings. The efficiency of Ferret-Reg is dominated by SPCOT and LPN computation. The 75% of running time is used to generate SPCOT correlations when the network bandwidth is limited to 10 Mbps, and it is reduced to 33% when the bandwidth is up to 100 Mbps. This is because SPCOT also has network transmission involved. The LPN encoding only requires the local computation, and takes about 12 *ns* per COT correlation. We also include the *extra* running time of Ferret-Uni compared to Ferret-Reg, due to the use of Cuckoo hashing.

CHAPTER 3

Zero-Knowledge Proofs from VOLE

In this chapter, we propose our zero-knowledge proof protocols based on the (subfield) vector oblivious linear evaluation discussed in Chapter 2. All of these protocols utilize sVOLE to construct a commitment scheme and they mainly differ in the verification protocols. We start with the Wolverine [96] protocol, which verifies the authenticated triples using the cut-and-bucketing. Then we discuss the QuickSilver [100], which improves from the verification protocol in Wolverine and achieves at least 3× improvement in terms of communication. Its technique is inspired by LPZK [46]. We also show a generalization version of Quicksilver which batch-proves the polynomial satisfiability with cost only linear to the degree of the polynomial. In the end, we use a series of experiments to demonstrate the concrete efficiency of these protocols.

3.1. VOLE-ZK from Cut-and-Bucketing

In Figure 3.1, we describe our zero-knowledge protocol Π_{ZK}, which operates in the $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$-hybrid model. Our protocol can be viewed as following a "GMW-style" approach to secure two-party computation using authenticated multiplication triples [83, 37]. In the secure-computation setting, the evaluation of a multiplication gate requires two rounds of interaction, since the parties hold shares of the values on the input wires, but neither party knows those values. In the ZK setting, however, the prover $\mathcal{P}$ knows the values on

Protocol Π_{ZK}

Inputs and parameters: The prover $\mathcal{P}$ and verifier $\mathcal{V}$ hold a circuit $\mathcal{C}$ over a finite field $\mathbb{F}_p$ with C multiplication gates; $\mathcal{P}$ holds a witness w such that $\mathcal{C}(w) = 1$. Fix parameters B, c, and r, and let $\ell = C \cdot B + c$.

Offline phase:

(1) $\mathcal{P}$ (acting as $\mathsf{P_A}$) and $\mathcal{V}$ (acting as $\mathsf{P_B}$) send init to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, which returns a uniform $\Delta \in \mathbb{F}_{p^r}$ to $\mathcal{V}$.

(2) $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{extend}, |\mathcal{I}_{\mathsf{in}}| + 3\ell + C)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, which returns authenticated values $\{[\lambda_i]\}_{i \in \mathcal{I}_{\mathsf{in}}}$, $\{([x_i], [y_i], [r_i])\}_{i \in [\ell]}$, and $\{[s_i]\}_{i \in [C]}$ to the parties.
(If $\mathcal{V}$ receives abort from $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, then it aborts.)

(3) For $i \in [\ell]$, $\mathcal{P}$ sends $d_i := x_i \cdot y_i - r_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[z_i] := [r_i] + d_i$.

Online phase:

(4) For $i \in \mathcal{I}_{\mathsf{in}}$, $\mathcal{P}$ sends $\Lambda_i := w_i - \lambda_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_i] := [\lambda_i] + \Lambda_i$.

(5) For each gate $(\alpha, \beta, \gamma, T) \in \mathcal{C}$, in topological order:
  (a) If $T = \mathsf{Add}$, then the two parties locally compute $[w_\gamma] := [w_\alpha] + [w_\beta]$.
  (b) If $T = \mathsf{Mult}$ and this is the ith multiplication gate, $\mathcal{P}$ sends $d := w_\alpha \cdot w_\beta - s_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_\gamma] := [s_i] + d$.

(6) $\mathcal{V}$ samples a random permutation π on $\{1, \ldots, \ell\}$ and sends it to $\mathcal{P}$. The two parties use π to permute the $\{([x_i], [y_i], [z_i])\}_{i \in [\ell]}$ obtained in step 3.

(7) For the ith multiplication gate $(\alpha, \beta, \gamma, \mathsf{Mult})$, where the parties obtained $([w_\alpha], [w_\beta], [w_\gamma])$ in step 5, do the following for $j = 1, \ldots, B$:
  (a) Let $([x], [y], [z])$ be the $\big((i-1)B + j\big)$th authenticated triple (after applying π in step 6).
  (b) The parties run $\delta_\alpha := \mathsf{Open}([w_\alpha] - [x])$ and $\delta_\beta := \mathsf{Open}([w_\beta] - [y])$. The parties then compute $[\mu] := [z] - [w_\gamma] + \delta_\beta \cdot [x] + \delta_\alpha \cdot [y] + \delta_\alpha \cdot \delta_\beta$, and finally run $\mathsf{CheckZero}([\mu])$.

(8) For each of the remaining c authenticated triples, say $([x], [y], [z])$, the parties run $x := \mathsf{Open}([x])$ and $y := \mathsf{Open}([y])$. They also compute $[\nu] := [z] - x \cdot y$ and then run $\mathsf{CheckZero}([\nu])$.

(9) For the single output wire $o \in \mathcal{I}_{\mathsf{out}}$ with authenticated value $[w_o]$, the parties run $\mathsf{CheckZero}([w_o] - 1)$.

Figure 3.1. **Zero-knowledge proof in the $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$-hybrid model.**

all wires; thus, evaluation of a multiplication gate can be done without any interaction at all. At a high level, our protocol consists of the following steps:

(1) **Initialization.** The parties prepare authenticated values $\{[\lambda_i]\}$ for the witness, and $\{[s_i]\}$ for each multiplication gate in the circuit. The parties also generate some number of authenticated multiplication triples $\{([x_i], [y_i], [z_i])\}$; a malicious prover may cause some or all of these triples to be incorrect (i.e., $z_i \neq x_i \cdot y_i$).

(2) **Circuit evaluation.** Starting with the authenticated values $\{[w_i]\}$ at the input wires, the parties inductively compute authenticated values for all the wires in the circuit. For addition gates, this is easy. For the i-th multiplication gate, the prover uses $[s_i]$ to enable the verifier to compute its component of the authenticated value for the output wire without revealing information about the values on the input wires. Specifically, given authenticated values $[w_\alpha], [w_\beta]$ on the input wires to the ith multiplication gate, the prover sends $w_\alpha \cdot w_\beta - s_i$ to the verifier; the prover and verifier then compute

$$[w_\gamma] := [s_i] + (w_\alpha \cdot w_\beta - s_i)$$

as the authenticated value of the output wire. All communication here is from the prover to the verifier, so the entire circuit can be evaluated using only one round of communication.

Once the parties have an authenticated value $[w_o]$ for the output wire, the prover simply opens that value, and the verifier checks that it is equal to 1.

(3) **Verifying correct behavior.** So far, nothing prevents a malicious prover from cheating. To detect cheating, the verifier needs to check the behavior of the prover at each multiplication gate using the initial set of authenticated multiplication triples the parties generated. This can be done in various ways. In the protocol

as described in Figure 3.1, which works for circuits over an arbitrary field, the verifier checks the behavior of the prover as follows (adapting [3]):

- The verifier checks a random subset of the authenticated triples to make sure they are correctly formed. For an authenticated multiplication triple $([x], [y], [z])$, this can be done by having the prover run $\mathsf{Open}([x])$ and $\mathsf{Open}([y])$ followed by $\mathsf{CheckZero}([z] - x \cdot y)$.
- The verifier then uses the remaining authenticated triples to check that each multiplication gate was computed correctly. For a multiplication gate with authenticated values $[w_\alpha], [w_\beta]$ on the input wires and $[w_\gamma]$ on the output wire, the relation $w_\gamma = w_\alpha w_\beta$ can be checked using an authenticated multiplication triple $([x], [y], [z])$ by having the prover run $\delta_\alpha := \mathsf{Open}([w_\alpha] - [x])$ and $\delta_\beta := \mathsf{Open}([w_\beta] - [y])$, followed by

$$\mathsf{CheckZero}\left([z] - [w_\gamma] + \delta_\beta \cdot [x] + \delta_\alpha \cdot [y] + \delta_\alpha \cdot \delta_\beta\right).$$

  Each multiplication gate is checked in this way using B authenticated multiplication triples.

Later, we describe other approaches for verifying correct behavior.

Note that the checks for the openings of all the authenticated values (i.e., all the executions of Open and $\mathsf{CheckZero}$) can be batched together at the end of the protocol.

Non-interactive online phase. The ZK protocol described in Figure 3.1 can be implemented in constant rounds. If we use the Fiat-Shamir heuristic both for deriving the permutation π as well as for non-interactive opening of authenticated values, the online phase can be made non-interactive.

3.1.1. Proof of Security

Before giving the proof of security for Π_{ZK}, we analyze the procedure used to check correctness of the multiplication gates. Consider some multiplication gate with authenticated values $[w_\alpha]$, $[w_\beta]$ on the input wires and $[w_\gamma]$ on the output wire. If $\mathcal{P}$ cheated, so $w_\gamma \neq w_\alpha \cdot w_\beta$, then this cheating will be detected in step 7 of the protocol unless all B of the multiplication triples used to check that gate are incorrect. (We ignore for now the possibility that $\mathcal{P}$ is able to successfully cheat when running Open/CheckZero.) But if too many of the initial multiplication triples are incorrect, then there is a high probability that $\mathcal{P}$ will be caught in step 8. We can analyze the overall probability with which a cheating $\mathcal{P}$ can successfully evade detection by considering an abstract "balls-and-bins" game with an adversary $\mathcal{A}$, which is based on a similar game considered previously in the context of secure three-party computation [3]. The game proceeds as follows:

(1) $\mathcal{A}$ prepares $\ell = CB + c$ balls $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$, each of which is either good or bad. $\mathcal{A}$ also prepares C bins, each of which is either good or bad. The balls $\{\mathcal{B}_i\}_{i\in[\ell]}$ correspond to the triples $\{([x_i], [y_i], [z_i])\}_{i\in[\ell]}$ defined in step 3 of the protocol, and the bins correspond to the triples $\{([w_\alpha], [w_\beta], [w_\gamma])\}$ defined for the multiplication gates during the circuit evaluation.

(2) Then, c random balls are chosen. If any of the chosen balls is bad, $\mathcal{A}$ loses. Otherwise, the game proceeds to the next step.

(3) The remaining CB balls are randomly partitioned into the C bins, with each bin receiving exactly B balls.

(4) We say that a bin is fully good (resp., fully bad) if it is labeled good and all the balls inside it are good (resp., labeled bad and all the balls inside it are bad). $\mathcal{A}$

wins if and only if there exists at least one bin that is fully bad, and all other bins are either fully good or fully bad.

Lemma 3. *Assume $c \geq B$. Then $\mathcal{A}$ wins the above game with probability at most $\binom{CB+c}{B}^{-1}$.*

Proof. Assume $\mathcal{A}$ makes m bins bad for $1 \leq m \leq C$. It is easy to see that $\mathcal{A}$ can only possibly win if exactly mB balls among $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$ are bad, and they are exactly placed in the m bins that are bad. We compute the probability that $\mathcal{A}$ wins for some fixed m.

Since exactly mB balls of the $\ell = CB + c$ balls are bad, the probability that none of the bad balls is chosen in step 2 of the game is exactly

$$\frac{\binom{\ell - mB}{c}}{\binom{\ell}{c}} = \frac{(\ell - mB)! \cdot (\ell - c)!}{\ell! \cdot (\ell - mB - c)!} = \frac{(CB + c - mB)! \cdot (CB)!}{(CB + c)! \cdot (CB - mB)!}.$$

Assume that this occurs. We are left with $\ell - c = CB$ balls, of which mB are bad. The probability that B bad balls are placed in each bad bin is

$$p_1 = \frac{(mB)! \cdot (CB - mB)!}{(CB)!}.$$

Thus, the probability that $\mathcal{A}$ wins is exactly

$$\frac{\binom{\ell - mB}{c}}{\binom{\ell}{c}} \cdot p_1 = \frac{(CB + c - mB)! \cdot (mB)!}{(CB + c)!} = \binom{CB + c}{mB}^{-1}.$$

For $c \geq B$, $1 \leq m \leq C$, this is maximized when $m = 1$. □

Now we prove security of protocol Π_{ZK}.

Theorem 4. *Let $c \geq B$. Protocol Π_{ZK} UC-realizes $\mathcal{F}_{\mathsf{ZK}}$ in the $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$-hybrid model. In particular, no environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution except with probability at most $\binom{CB+c}{B}^{-1} + p^{-r} + \epsilon_{\mathsf{open}}$.*

Proof. We first consider the case of a malicious prover (i.e., soundness) and then consider the case of a malicious verifier (i.e., zero knowledge). In each case, we construct a PPT simulator $\mathcal{S}$ given access to $\mathcal{F}_{\mathsf{ZK}}$, and running the PPT adversary $\mathcal{A}$ as a subroutine while emulating functionality $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ for $\mathcal{A}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and $\mathcal{Z}$.

Malicious prover. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ for $\mathcal{A}$ by choosing uniform $\Delta \in \mathbb{F}_{p^r}$ and recording all the values $\{\lambda_i\}_{i\in\mathcal{I}_{\mathsf{in}}}$, $\{(x_i, y_i, r_i)\}_{i\in[\ell]}$, and $\{s_i\}_{i\in[C]}$, and their corresponding MAC tags, sent to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by $\mathcal{A}$. These values define corresponding keys in the natural way.

(2) If $\mathcal{A}$ makes a global-key query $(\mathsf{guess}, \Delta')$ to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, then $\mathcal{S}$ checks if $\Delta = \Delta'$. If not, $\mathcal{S}$ sends abort to $\mathcal{A}$, sends $(\mathsf{prove}, \mathcal{C}, \bot)$ to $\mathcal{F}_{\mathsf{ZK}}$, and aborts. Otherwise, $\mathcal{S}$ sends $\mathsf{success}$ to $\mathcal{A}$ and continues.

(3) When $\mathcal{A}$ sends $\{\Lambda_i\}_{i\in\mathcal{I}_{\mathsf{in}}}$ in step 4, $\mathcal{S}$ sets $w_i := \lambda_i + \Lambda_i$ for $i \in \mathcal{I}_{\mathsf{in}}$.

(4) $\mathcal{S}$ runs the rest of the protocol as an honest verifier, using Δ and the keys defined in the first step. If the honest verifier outputs false, then $\mathcal{S}$ sends $(\mathsf{prove}, \mathcal{C}, \bot)$ to $\mathcal{F}_{\mathsf{ZK}}$ and aborts. If the honest verifier outputs true, then $\mathcal{S}$ sends $(\mathsf{prove}, \mathcal{C}, w)$ to $\mathcal{F}_{\mathsf{ZK}}$ where w is defined as above.

We assume that $\mathcal{A}$ does not correctly guess Δ; this is true except with probability at most p^{-r}. It is clear that the view of $\mathcal{A}$ is perfectly simulated by $\mathcal{S}$. Whenever the verifier

simulated by $\mathcal{S}$ outputs false, the real verifier outputs false as well (since $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathsf{ZK}}$). It thus only remains to bound the probability with which the simulated verifier run by $\mathcal{S}$ outputs true but the witness w sent by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{ZK}}$ satisfies $\mathcal{C}(w) = 0$. Below, we show that if $\mathcal{C}(w) = 0$ then the probability that the simulated verifier outputs true is at most $\binom{CB+c}{B}^{-1} + \epsilon_{\mathsf{open}}$.

If $\mathcal{C}(w) = 0$ then either $w_o = 0$ or else at least one of the triples $\{([w_\alpha], [w_\beta], [w_\gamma])\}$ defined at the multiplication gates during the circuit evaluation must be incorrect. In the former case, the probability that $\mathcal{P}$ succeeds when running $\mathsf{CheckZero}([w_o] - 1)$ is at most ϵ_{open}. In the latter case, Lemma 3 shows that the probability that $\mathcal{A}$ avoids being "caught" in steps 6–8 is at most $\binom{CB+c}{B}^{-1}$; if $\mathcal{A}$ is caught, then it succeeds in opening some incorrect value with probability at most ϵ_{open}. This completes the proof for the case of a malicious prover.

Malicious verifier. If $\mathcal{S}$ receives false from $\mathcal{F}_{\mathsf{ZK}}$, then it simply aborts. Otherwise, $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by recording the global key Δ, and the keys for all the authenticated values, sent to the functionality by $\mathcal{A}$. Then, $\mathcal{S}$ samples uniform values for $\{\lambda_i\}_{i\in\mathcal{I}_{\mathsf{in}}}$, $\{(x_i, y_i, r_i)\}_{i\in[\ell]}$, and $\{s_i\}_{i\in[C]}$, and computes their corresponding MAC tags in the natural way.

(2) $\mathcal{S}$ executes steps 3–8 of protocol Π_{ZK} by simulating the honest prover with input $w = 0^{|\mathcal{I}_{\mathsf{in}}|}$.

(3) In step 9, $\mathcal{S}$ computes $\mathsf{K}[w_o]$ (based on the keys sent to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ by $\mathcal{A}$) and then sets $\mathsf{M}[w_o] := \mathsf{K}[w_o] + \Delta$. Finally, it uses $\mathsf{M}[w_o]$ to run $\mathsf{CheckZero}([w_o] - 1)$ with $\mathcal{A}$.

The view of $\mathcal{A}$ simulated by $\mathcal{S}$ is distributed identically to its view in the real protocol execution. This completes the proof. □

3.1.2. Other Approaches for Verifying Correct Behavior

Here we describe alternative approaches for checking correctness of multiplication gates for large p (i.e., $\log p \geq \rho$).

Approach 1. The first approach can be viewed as a simplified version of the check used by SPDZ [37]. Both parties now prepare a *single* authenticated multiplication triple $([x], [y], [z])$ per multiplication gate (so only C in total), which may be incorrect if $\mathcal{P}$ is malicious. To check correctness of a multiplication gate with authenticated values $[w_\alpha]$, $[w_\beta]$ on the input wires and $[w_\gamma]$ on the output wire, the verifier sends a uniform $\eta \in \mathbb{F}_p$ to the prover, who responds by running $\delta_\alpha := \mathsf{Open}(\eta \cdot [w_\alpha] - [x])$ and $\delta_\beta := \mathsf{Open}([w_\beta] - [y])$, followed by

$$\mathsf{CheckZero}([z] - \eta \cdot [w_\gamma] + \delta_\beta \cdot [x] + \delta_\alpha \cdot [y] + \delta_\alpha \cdot \delta_\beta).$$

This has soundness error $1/p + \epsilon_{\mathsf{open}}$. To see this, say $w_\gamma = w_\alpha w_\beta + \Delta_w$ with $\Delta_w \neq 0$, and let $z = xy + \Delta_z$. Then $z - \eta \cdot w_\gamma + \delta_\beta \cdot x + \delta_\alpha \cdot y + \delta_\alpha \cdot \delta_\beta = 0$ iff $\eta = \Delta_z/\Delta_w$, which occurs with probability $1/p$. Note that this checking procedure can be done for all multiplication gates in parallel using a single value η, and the overall soundness error remains unchanged. It can also be made non-interactive using the Fiat-Shamir heuristic in the random-oracle model.

Approach 2: Trading off communication and computation. This approach, which is a simplified and improved variant of the polynomial approach used by SPDZ [37], reduces the communication complexity by roughly half (from 4 to 2 field elements per

gate) at the expense of increased computation. Intuitively, the prover and verifier define polynomials F, G, H that interpolate to $\{w_\alpha^i\}$, $\{w_\beta^i\}$, and $\{w_\gamma^i\}$, respectively. If $w_\gamma^i = w_\alpha^i \cdot w_\beta^i$ for all i, then $H = F \cdot G$, and this can be verified by checking whether $H(\nu) = F(\nu) \cdot G(\nu)$ at a random point $\nu \in \mathbb{F}_{p^r}$. Details follow.

Assume $p \geq 2C - 1$. Let $([w_\alpha^i], [w_\beta^i], [w_\gamma^i])$ be the authenticated values corresponding to the ith multiplication gate. The parties additionally compute $C - 1$ authenticated values $\{[s_i]\}_{i \in [C+1, 2C)}$; they also compute an authenticated multiplication triple $([x], [y], [z])$ (which may be incorrect if $\mathcal{P}$ is malicious) with $x, y, z \in \mathbb{F}_{p^r}$.[1] They then do the following:

(1) Let $F \in \mathbb{F}_p[X]$ (resp., $G \in \mathbb{F}_p[X]$) be the polynomial of degree at most $C-1$ such that $F(i) = w_\alpha^i$ (resp., $G(i) = w_\beta^i$) for $i \in [C]$. Note that $\mathcal{P}$ can compute F and G explicitly, and $\mathcal{P}$ and $\mathcal{V}$ can compute the authenticated value $[w_\alpha^k] \stackrel{\text{def}}{=} [F(k)]$ (resp., $[w_\beta^k] \stackrel{\text{def}}{=} [G(k)]$) for any $k \in \mathbb{F}_{p^r}$ using Lagrange interpolation over the shares $\{[w_\alpha^i]\}_{i \in [C]}$ (resp., $\{[w_\beta^i]\}_{i \in [C]}$).

(2) For $k \in [C + 1, 2C)$, $\mathcal{P}$ sends $d'_k := w_\alpha^k \cdot w_\beta^k - s_k$ to $\mathcal{V}$, and both parties compute $[w_\gamma^k] := [s_k] + d'_k$. Let $H \in \mathbb{F}_p[X]$ be the polynomial of degree at most $2C - 2$ such that $H(i) = w_\gamma^i$ for $i \in [2C - 1]$. Note that $\mathcal{P}$ can compute H explicitly, while $\mathcal{P}$ and $\mathcal{V}$ can compute the authenticated value $[H(k)]$ for any $k \in \mathbb{F}_{p^r}$ using Lagrange interpolation over the shares $[w_\gamma^i]$.

(3) $\mathcal{V}$ sends a uniform $\nu \in \mathbb{F}_{p^r}$ to $\mathcal{P}$. Then the parties compute authenticated values $[F(\nu)]$, $[G(\nu)]$, and $[H(\nu)]$.

[1] A uniform authenticated value $[z]$ with $z \in \mathbb{F}_{p^r}$ can be generated from r uniform authenticated values $[z_1], \ldots, [z_r]$ with $z_i \in \mathbb{F}_p$ by setting $z = \sum_i z_i \cdot \mathsf{X}^i$. An authenticated multiplication triple can be computed from such authenticated values in the natural way.

(4) Finally, $\mathcal{V}$ verifies that $F(\nu) \cdot G(\nu) = H(\nu)$ as in approach 1, above. That is, $\mathcal{V}$ sends a uniform $\eta \in \mathbb{F}_{p^r}$ to $\mathcal{P}$, who responds by running $\delta := \mathsf{Open}(\eta \cdot [F(\nu)] - [x])$ and $\sigma := \mathsf{Open}([G(\nu)] - [y])$, followed by

$$\mathsf{CheckZero}([z] - \eta \cdot [H(\nu)] + \sigma \cdot [x] + \delta \cdot [y] + \delta \cdot \sigma).$$

This has soundness error $(2C-1)/p^r + \epsilon_{\mathsf{open}}$. To see this, note that if there exists an $i \in [C]$ with $w^i_\alpha \cdot w^i_\beta \neq w^i_\gamma$ then the polynomials $F \cdot G$ and H are different, and so agree in at most $2C-2$ points. Thus, $F(\nu) \cdot G(\nu) \neq H(\nu)$ except with probability at most $(2C-2)/p^r$. When that is the case, an analysis in the first approach shows that the final check fails except with probability at most $1/p^r + \epsilon_{\mathsf{open}}$.

This approach can also be made non-interactive using the Fiat-Shamir heuristic in the random-oracle model.

3.2. Improved Committed Triple Verification

In this section, we present our ZK protocol for circuit satisfiability over any field with communication of only one field element per multiplication gate using sVOLE as a subroutine. First of all, we introduce a functionality (and the corresponding protocol) that extends sVOLE to additionally support *vector oblivious polynomial evaluation* (VOPE), which is crucial for our ZK protocols in this section and the next section. Then, we provide the details of our ZK protocol, and give the formal security proof.

Functionality $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$

Initialize: Upon receiving (init) from $\mathcal{P}$ and $\mathcal{V}$, sample $\Delta \leftarrow \mathbb{F}_{p^r}$ if $\mathcal{V}$ is honest, and receive $\Delta \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store Δ and send it to $\mathcal{V}$, and ignore all subsequent (init) commands.

Extend: This procedure can be run multiple times. Upon receiving (extend, ℓ) from $\mathcal{P}$ and $\mathcal{V}$, do the following:

(1) If $\mathcal{V}$ is honest, sample $\mathsf{K} \leftarrow \mathbb{F}^{\ell}_{p^r}$. Otherwise, receive $\mathsf{K} \in \mathbb{F}^{\ell}_{p^r}$ from the adversary.
(2) If $\mathcal{P}$ is honest, sample $\boldsymbol{x} \leftarrow \mathbb{F}^{\ell}_{p}$ and compute $\mathsf{M} := \mathsf{K} - \Delta \cdot \boldsymbol{x} \in \mathbb{F}^{\ell}_{p^r}$. Otherwise, receive $\boldsymbol{x} \in \mathbb{F}^{\ell}_{p}$ and $\mathsf{M} \in \mathbb{F}^{\ell}_{p^r}$ from the adversary, and then recompute $\mathsf{K} := \mathsf{M} + \Delta \cdot \boldsymbol{x} \in \mathbb{F}^{\ell}_{p^r}$.
(3) Send $(\boldsymbol{x}, \mathsf{M})$ to $\mathcal{P}$ and K to $\mathcal{V}$.

Vector Oblivious Polynomial Evaluation: Upon receiving (VOPE, d) from $\mathcal{P}$ and $\mathcal{V}$, do the following:

(1) If $\mathcal{V}$ is honest, sample $B \leftarrow \mathbb{F}_{p^r}$. Otherwise, receive $B \in \mathbb{F}_{p^r}$ from the adversary.
(2) If $\mathcal{P}$ is honest, sample $A_i \leftarrow \mathbb{F}_{p^r}$ for $i \in [d]$ and compute $A_0 := B - \sum_{i\in[d]} A_i \cdot \Delta^i$. Otherwise, receive $\{A_i\}_{i\in[0,d]}$ with $A_i \in \mathbb{F}_{p^r}$ from the adversary and recompute $B := \sum_{i\in[0,d]} A_i \cdot \Delta^i$.
(3) Send $\{A_i\}_{i\in[0,d]}$ to $\mathcal{P}$ and B to $\mathcal{V}$.

Figure 3.2. Functionality for extended subfield VOLE.

3.2.1. Extended Subfield Vector Oblivious Linear Evaluation

Extended sVOLE functionality. To accommodate our efficient ZK protocols for circuits and polynomial sets (described in Section 3.2 and Section 3.3), we propose an extended sVOLE functionality $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ defined in Figure 3.2 to generate authenticated values and special correlations related to random polynomials. This functionality is the same as that shown in Figure 2.1, except that it additionally allows two parties to obtain VOPE correlations over $\mathbb{F}_{p^r}$ with the guarantee that the same global key Δ is used between sVOLE and VOPE. In particular, given a polynomial-degree d input by both parties, this functionality will sample $d+1$ uniform coefficients over extension field $\mathbb{F}_{p^r}$ to define a random polynomial g, and then output the coefficients to a party $\mathcal{P}$ and $g(\Delta)$ to the other party $\mathcal{V}$.

Protocol for realizing extended sVOLE functionality. We construct the protocol to UC-realize functionality $\mathcal{F}_{\mathsf{ext\text{-}sVOLE}}^{p,r}$ by extending the sVOLE protocol. Recall that our extended functionality can be viewed as adding the support to output VOPE correlations over extension field $\mathbb{F}_{p^r}$. Our protocol to accomplish it takes two steps: 1) packing subfield VOLE correlations between $\mathbb{F}_p$ and $\mathbb{F}_{p^r}$ into VOLE correlations over $\mathbb{F}_{p^r}$; 2) multiplying independent VOLE correlations to obtain a VOPE correlation. We note that a malicious party $\mathcal{V}$ could cause the outputting coefficients $A_1, \ldots, A_{d-1}$ of honest party $\mathcal{P}$ to be always 0 by setting $\Delta = 0$ and all its keys as 0. To prevent the attack, we *iteratively* multiply the VOLE correlations over $\mathbb{F}_{p^r}$, and use an extra independent VOLE correlation to randomize the product of VOLE correlations after multiplication is computed in every iteration. Details of the protocol are described in Figure 3.3.

The security of this protocol is proved in the following theorem.

Theorem 5. *Protocol* $\Pi_{\mathsf{ext\text{-}sVOLE}}^{p,r}$ *shown in Figure 3.3 UC-realizes* $\mathcal{F}_{\mathsf{ext\text{-}sVOLE}}^{p,r}$ *with statistical error* $(d-1)/p^r$ *and information-theoretic security in the* $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$*-hybrid model.*

Proof. We construct a simulator $\mathcal{S}$ given access to $\mathcal{F}_{\mathsf{ext\text{-}sVOLE}}^{p,r}$, and running the adversary $\mathcal{A}$ as a subroutine while emulating $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ for $\mathcal{A}$. In particular, there is no communication between $\mathcal{P}$ and $\mathcal{V}$. Thus, $\mathcal{S}$ can emulate $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ and record all the values sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, and then compute the output value for the corrupted party following the protocol specification, and send it to functionality $\mathcal{F}_{\mathsf{ext\text{-}sVOLE}}^{p,r}$. It is trivial to see that the simulation is perfect.

Below, we show that the output of the honest party is statistically indistinguishable between the real-world execution and ideal-world execution. We first prove if both parties

Protocol $\Pi^{p,r}_{\mathsf{ext\text{-}sVOLE}}$

Initialize. $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns a uniform $\Delta \in \mathbb{F}_{p^r}$ to $\mathcal{V}$.

Generate sVOLE correlations. On input (extend, ℓ), two parties $\mathcal{P}$ and $\mathcal{V}$ call $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$ to directly generate ℓ sVOLE correlations.

Generate VOPE correlations. On input (VOPE, d), two parties $\mathcal{P}$ and $\mathcal{V}$ execute the following:

(1) For each $i \in [2d-1]$, two parties perform as follows:
   (a) Both parties send (extend, r) to $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$, which returns $\{(m_h, u_h)\}_{h\in[r]}$ to $\mathcal{P}$ and $\{k_h\}_{h\in[r]}$ to $\mathcal{V}$ such that $k_h = m_h + u_h \cdot \Delta \in \mathbb{F}_{p^r}$ and $u_h \in \mathbb{F}_p$ for $h \in [r]$.
   (b) $\mathcal{P}$ computes $M_i := \sum_{h\in[r]} m_h \cdot X^{h-1} \in \mathbb{F}_{p^r}$ and $U_i := \sum_{h\in[r]} u_h \cdot X^{h-1} \in \mathbb{F}_{p^r}$, and $\mathcal{V}$ computes $K_i := \sum_{h\in[r]} k_h \cdot X^{h-1} \in \mathbb{F}_{p^r}$, where $K_i = M_i + U_i \cdot \Delta$, and recall that each element in $\mathbb{F}_{p^r} \cong \mathbb{F}_p[X]/f(X)$ is denoted by a polynomial.

(2) $\mathcal{P}$ defines $g_1(x) = M_1 + U_1 \cdot x$, and $\mathcal{V}$ sets $B_1 = K_1$. If $d > 1$, from $i = 1$ to $d-1$, two parties execute as follows:
   (a) $\mathcal{P}$ computes the following univariate polynomial:
   $$g_{i+1}(x) = g_i(x) \cdot (M_{i+1} + U_{i+1} \cdot x) + (M_{d+i} + U_{d+i} \cdot x).$$
   (b) $\mathcal{V}$ computes $B_{i+1} := B_i \cdot K_{i+1} + K_{d+i}$.

   Then, $\mathcal{P}$ computes the coefficients $\{A_i\}_{i\in[0,d]}$ locally such that $g_d(x) = \sum_{i\in[0,d]} A_i \cdot x^i$, and $\mathcal{V}$ defines $B := B_d$.

(3) $\mathcal{P}$ outputs $\{A_i\}_{i\in[0,d]}$; $\mathcal{V}$ outputs B.

Figure 3.3. Protocol for extended subfield VOLE in the $\mathcal{F}^{p,r}_{\mathsf{sVOLE}}$-hybrid model.

compute their output *locally* following the protocol specification, then their outputs satisfy the correct VOPE correlation. Specifically, from $k_h = m_h + u_h \cdot \Delta$ for $h \in [r]$, we easily obtain that for each $i \in [d]$,

$$\begin{aligned} K_i &= \sum_{h\in[r]} k_h \cdot X^{h-1} = \sum_{h\in[r]} (m_h + u_h \cdot \Delta) \cdot X^{h-1} \\ &= \sum_{h\in[r]} m_h \cdot X^{h-1} + \Big(\sum_{h\in[r]} u_h \cdot X^{h-1} \Big) \cdot \Delta \\ &= M_i + U_i \cdot \Delta. \end{aligned}$$

It is easy to see that $B_1 = K_1 = M_1 + U_1 \cdot \Delta = g_1(\Delta)$. Below, we prove by induction. In the i-th iteration with $i \in [d-1]$, assuming that $B_i = g_i(\Delta)$, we have the following holds:

$$\begin{aligned} B_{i+1} &= B_i \cdot K_{i+1} + K_{d+i} \\ &= g_i(\Delta) \cdot (M_{i+1} + U_{i+1} \cdot \Delta) + (M_{d+i} + U_{d+i} \cdot \Delta) \\ &= g_{i+1}(\Delta). \end{aligned}$$

Therefore, we obtain that $B = B_d = g_d(\Delta) = \sum_{i \in [0,d]} A_i \cdot \Delta^i$.

If $\mathcal{V}$ is honest, then its output is always defined by the output of malicious party $\mathcal{P}$ and Δ (i.e., $B = \sum_{i \in [0,d]} A_i \cdot \Delta^i$) in both worlds. In the following, we consider the case that $\mathcal{P}$ is honest but $\mathcal{V}$ is malicious. The output values $A_0, \ldots, A_d$ for $\mathcal{P}$ are uniformly random such that $B = \sum_{i \in [0,d]} A_i \cdot \Delta^i$ in the ideal-world execution, where B is the output of malicious party $\mathcal{V}$. In the real protocol execution, A_i for each $i \in [0, d]$ is computed as the coefficient of item x^i for polynomial $g_d(x)$. According to the definition of $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$, u_h for $h \in [r]$ is uniform in $\mathbb{F}_p$. Therefore, for $i \in [2d-1]$, we have that $U_i := \sum_{h \in [r]} u_h \cdot X^{h-1}$ is uniformly random in $\mathbb{F}_{p^r}$. For $i \in [d]$, we prove by induction that each coefficient of $g_i(x)$ except for constant term is uniformly distributed in $\mathbb{F}_{p^r}$, except with probability at most $1/p^r$. This holds for $g_1(x) = M_1 + U_1 \cdot x$ with probability 1. In the i-th iteration with $i \in [d-1]$, we have that the coefficients $A_{i,1}, \ldots, A_{i,i}$ of degree-i polynomial $g_i(x)$ are uniform by the induction assumption. From the definition of $g_{i+1}(x)$, we obtain the

following holds:

$$\begin{aligned} g_{i+1}(x) &= g_i(x) \cdot (M_{i+1} + U_{i+1} \cdot x) + (M_{d+i} + U_{d+i} \cdot x) \\ &= \left(\sum_{h=0}^{i} A_{i,h} \cdot x^h \right) \cdot (M_{i+1} + U_{i+1} \cdot x) + (M_{d+i} + U_{d+i} \cdot x) \\ &= (A_{i,0} \cdot M_{i+1} + M_{d+i}) + (A_{i,1} \cdot M_{i+1} + A_{i,0} \cdot U_{i+1} + U_{d+i}) \cdot x \\ &\quad \sum_{h=2}^{i} (A_{i,h} \cdot M_{i+1} + A_{i,h-1} \cdot U_{i+1}) \cdot x^h + A_{i,i} \cdot U_{i+1} \cdot x^{i+1}. \end{aligned}$$

From the uniformity of U_{d+i}, we directly obtain that the 1-degree term of $g_{i+1}(x)$ is uniform. If $U_{i+1} \neq 0$ except with probability $1/p^r$, then the h-degree term of $g_{i+1}(x)$ for $h \in [2, i+1]$ is uniform from the uniformity of $A_{i,h-1}$. Overall, except with probability $1/p^r$, each coefficient of $g_{i+1}(x)$ except for constant term is uniformly random. Therefore, the coefficients $A_1, \ldots, A_d$ of polynomial $g_d(x)$ are uniform over $\mathbb{F}_{p^r}$, except with probability at most $(d-1)/p^r$. Together with $B = g_d(\Delta)$, we have that $A_0 = B - \sum_{i \in [d]} A_i \cdot \Delta^i$, which completes the proof. □

3.2.2. ZKP From Improved Triple Verification

We describe the details of the protocol in Figure 3.4. The online phase of the ZK protocol requires three rounds of communication. At the end of this section, we will show that the online phase can be made *non-interactive* in the random oracle model. Functionality $\mathcal{F}_{\mathsf{sVOLE}}^{p,r}$ can be securely realized in constant rounds using known protocols (e.g., [87, 25, 104, 95]), and thus $\mathcal{F}_{\mathsf{ext\text{-}sVOLE}}^{p,r}$ is able to be instantiated using constant-round protocols.

Protocol $\Pi^{p,r}_{\mathsf{ZK}}$

Inputs: The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ hold a circuit $\mathcal{C}$ over any field $\mathbb{F}_p$ with t multiplication gates. Prover $\mathcal{P}$ also holds a witness $\boldsymbol{w}$ such that $\mathcal{C}(\boldsymbol{w}) = 1$ and $|\boldsymbol{w}| = n$ (i.e., $|\mathcal{I}_{\mathsf{in}}| = n$).

Preprocessing phase: Both the circuit and witness are unknown.

(1) $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$, which returns a uniform $\Delta \in \mathbb{F}_{p^r}$ to $\mathcal{V}$.

(2) $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{extend}, n+t)$ to $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$, which returns authenticated values $\{[\mu_i]\}_{i\in[n]}$ and $\{[\nu_i]\}_{i\in[t]}$ to the parties.

(3) $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{VOPE}, 1)$ to $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$, which returns uniform (A^*_0, A^*_1) to $\mathcal{P}$ and B^* to $\mathcal{V}$, such that $B^* = A^*_0 + A^*_1 \cdot \Delta$.

Online phase: Now the circuit and witness are known by the parties.

(4) For $i \in \mathcal{I}_{\mathsf{in}}$, $\mathcal{P}$ sends $\delta_i := w_i - \mu_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_i] := [\mu_i] + \delta_i$.

(5) For each gate $(\alpha, \beta, \gamma, T) \in \mathcal{C}$, in a topological order:
- If $T = \mathsf{Add}$, then two parties locally compute $[w_\gamma] := [w_\alpha] + [w_\beta]$.
- If $T = \mathsf{Mult}$ and this is the i-th multiplication gate, $\mathcal{P}$ sends $d_i := w_\alpha \cdot w_\beta - \nu_i \in \mathbb{F}_p$ to $\mathcal{V}$, and then both parties compute $[w_\gamma] := [\nu_i] + d_i$ (with $w_\gamma = w_\alpha \cdot w_\beta$ in the honest case).

(6) For the i-th multiplication gate, two parties hold an authenticated triple $([w_\alpha], [w_\beta], [w_\gamma])$ (with $k_i = m_i + w_i \cdot \Delta$ for $i \in \{\alpha, \beta, \gamma\}$) from the previous step and execute the following:
- $\mathcal{P}$ computes $A_{0,i} := m_\alpha \cdot m_\beta \in \mathbb{F}_{p^r}$ and $A_{1,i} := w_\alpha \cdot m_\beta + w_\beta \cdot m_\alpha - m_\gamma \in \mathbb{F}_{p^r}$.
- $\mathcal{V}$ computes $B_i := k_\alpha \cdot k_\beta - k_\gamma \cdot \Delta \in \mathbb{F}_{p^r}$.

(7) $\mathcal{P}$ and $\mathcal{V}$ perform the following check to verify that $B_i = A_{0,i} + A_{1,i} \cdot \Delta$ for all $i \in [t]$.
   (a) $\mathcal{V}$ samples $\chi \leftarrow \mathbb{F}_{p^r}$ and sends it to $\mathcal{P}$.
   (b) $\mathcal{P}$ computes $U := \sum_{i\in[t]} A_{0,i} \cdot \chi^i + A^*_0$ and $V := \sum_{i\in[t]} A_{1,i} \cdot \chi^i + A^*_1$, and sends (U, V) to $\mathcal{V}$.
   (c) Then $\mathcal{V}$ computes $W := \sum_{i\in[t]} B_i \cdot \chi^i + B^*$ and checks that $W = U + V \cdot \Delta$. If the check fails, $\mathcal{V}$ outputs false and aborts.

(8) For the single output wire h in the circuit $\mathcal{C}$, both parties hold $[w_h]$ with $k_h = m_h + w_h \cdot \Delta$, and check that $w_h = 1$ as follows:
- In parallel with the previous step, $\mathcal{P}$ sends m_h to $\mathcal{V}$.
- $\mathcal{V}$ checks that $k_h = m_h + \Delta$. If the check fails, then $\mathcal{V}$ outputs false. Otherwise, $\mathcal{V}$ outputs true.

Figure 3.4. Zero-knowledge protocol for circuit satisfiability over any field in the $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$-hybrid model.

Overall, the ZK protocol shown in Figure 3.4 has constant rounds. In the following, we prove the security of our ZK protocol.

Proof of security. When both parties are honest, we easily see that the verifier will output true with probability 1. In particular, the check in protocol $\Pi^{p,r}_{\mathsf{ZK}}$ always passes for an honest execution. For an honest protocol execution, we always have that $w_h = 1$ (and thus $k_h = m_h + \Delta$) for the single output wire h. Overall, our ZK protocol $\Pi^{p,r}_{\mathsf{ZK}}$ shown in Figure 3.4 achieves perfect completeness.

Theorem 6. *Protocol $\Pi^{p,r}_{\mathsf{ZK}}$ UC-realizes functionality $\mathcal{F}_{\mathsf{ZK}}$ that proves the circuit satisfiability in the $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$-hybrid model with soundness error $(t+3)/p^r$ and information-theoretic security.*

Proof. We first consider the case of a malicious prover (i.e., soundness and knowledge extraction) and then consider the case of a malicious verifier (i.e., zero knowledge). In each case, we construct a simulator $\mathcal{S}$ given access to $\mathcal{F}_{\mathsf{ZK}}$, and running the adversary $\mathcal{A}$ as a subroutine while emulating $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ for $\mathcal{A}$. We always implicitly assume that $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

Malicious prover. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ for $\mathcal{A}$ by choosing uniform $\Delta \in \mathbb{F}_{p^r}$, and recording all the values $\{\mu_i\}_{i\in[n]}$ and $\{\nu_i\}_{i\in[N]}$ and their corresponding MAC tags, which are received by $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ from adversary $\mathcal{A}$. These values define the corresponding keys in the natural way. When emulating $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$, $\mathcal{S}$ also receives $(A_0^*, A_1^*) \in (\mathbb{F}_{p^r})^2$ from $\mathcal{A}$ and defines B^* accordingly.

(2) When $\mathcal{A}$ sends $\{\delta_i\}_{i\in\mathcal{I}_{\mathsf{in}}}$ in step 4, $\mathcal{S}$ computes $w_i := \delta_i + \mu_i \in \mathbb{F}_p$ for $i \in \mathcal{I}_{\mathsf{in}}$.

(3) $\mathcal{S}$ executes the rest of the protocol as an honest verifier, using Δ and the keys defined in the first step. If the honest verifier outputs false, then $\mathcal{S}$ sends $\boldsymbol{w} = \perp$

and $\mathcal{C}$ to $\mathcal{F}_{\mathsf{ZK}}$ and aborts. If the honest verifier outputs true, then $\mathcal{S}$ sends $\boldsymbol{w}$ and $\mathcal{C}$ to $\mathcal{F}_{\mathsf{ZK}}$ where $\boldsymbol{w} = (w_1, \ldots, w_n)$ is defined as above.

Clearly, the view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ has the identical distribution as its view in the real-world execution. Whenever the verifier in the real-world execution outputs false, the verifier in the ideal-world execution outputs false as well (since $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathsf{ZK}}$ in this case). Thus, it only remains to bound the probability that the verifier in the real-world execution outputs true but the witness $\boldsymbol{w}$ sent by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{ZK}}$ satisfies $\mathcal{C}(\boldsymbol{w}) = 0$. In the following, we show that if $\mathcal{C}(\boldsymbol{w}) = 0$ then the probability that the honest verifier in the real-world execution outputs true is at most $(t+3)/p^r$.

By induction, we prove that all the values on the wires in the circuit are correct. It is trivial that the values associated with the input wires and the output wires of Add gates are computed correctly. Therefore, we focus on analyzing the correctness of the values related to the output wires of Mult gates. When we analyze the correctness of the output value with respect to the i-th multiplication gate, we always assume that the output values associated with the first $(i-1)$ multiplication gates are correct by induction. For the i-th multiplication gate, two parties hold an authenticated triple $([w_\alpha], [w_\beta], [w_\gamma])$ with $w_\gamma = w_\alpha \cdot w_\beta + e_i$, where $e_i \in \mathbb{F}_p$ is an error chosen by adversary $\mathcal{A}$ by sending an incorrect value d_i' in step 5 of protocol $\Pi_{\mathsf{ZK}}^{p,r}$. Thus, we have $k_\gamma = m_\gamma + w_\gamma \cdot \Delta = m_\gamma + (w_\alpha \cdot w_\beta) \cdot \Delta + e_i \cdot \Delta$.

Further, we have:

$$\begin{aligned} B_i &= k_\alpha \cdot k_\beta - k_\gamma \cdot \Delta \\ &= (m_\alpha + w_\alpha \cdot \Delta) \cdot (m_\beta + w_\beta \cdot \Delta) \\ &\quad - (m_\gamma + w_\alpha \cdot w_\beta \cdot \Delta + e_i \cdot \Delta) \cdot \Delta \\ &= m_\alpha \cdot m_\beta + (w_\alpha \cdot m_\beta + w_\beta \cdot m_\alpha - m_\gamma) \cdot \Delta - e_i \cdot \Delta^2 \\ &= A_{0,i} + A_{1,i} \cdot \Delta - e_i \cdot \Delta^2. \end{aligned}$$

In step 7 of the ZK protocol, $\mathcal{A}$ sends $U' = U + E_u$ and $V' = V + E_v$ to the honest verifier, where $U, V \in \mathbb{F}_{p^r}$ are computed following the protocol description, and $E_u, E_v \in \mathbb{F}_{p^r}$ are the adversarially chosen errors. Furthermore, we have the following:

$$\begin{aligned} W &= \sum_{i\in[t]} B_i \cdot \chi^i + B^* \\ &= \sum_{i\in[t]} \left(A_{0,i} + A_{1,i} \cdot \Delta - e_i \cdot \Delta^2\right) \cdot \chi^i + A_0^* + A_1^* \cdot \Delta \\ &= U + V \cdot \Delta - \left(\sum_{i\in[t]} e_i \cdot \chi^i\right) \cdot \Delta^2 \\ &= (U' - E_u) + (V' - E_v) \cdot \Delta - \left(\sum_{i\in[t]} e_i \cdot \chi^i\right) \cdot \Delta^2. \end{aligned}$$

If the check passes in step 7, then we have that $W = U' + V' \cdot \Delta$. Therefore, we obtain that

$$E_u + E_v \cdot \Delta + \left(\sum_{i\in[t]} e_i \cdot \chi^i\right) \cdot \Delta^2 = 0.$$

If $\sum_{i\in[t]} e_i \cdot \chi^i \neq 0$, then the above equation holds with probability at most $2/p^r$, as $\Delta \in \mathbb{F}_{p^r}$ is uniformly random and kept secret from the adversary's view. Below, we consider that $\sum_{i\in[t]} e_i \cdot \chi^i = 0$. If there exists some $i \in [t]$ such that $e_i \neq 0$, the probability that $\sum_{i\in[t]} e_i \cdot \chi^i = 0$ is at most t/p^r, as χ is sampled uniformly at random after e_i for all $i \in [t]$ have been determined. Overall, all the values on the wires in the circuit are correct, except with probability at most $(t+2)/p^r$.

Now, we assume that all the values on the wires in the circuit are correct. If $\mathcal{C}(\boldsymbol{w}) = 0$ but the honest verifier outputs true in step 8, then adversary $\mathcal{A}$ must send $m_h + \Delta$ to the honest verifier where m_h is an MAC tag on output wire h known by $\mathcal{A}$. In other words, $\mathcal{A}$ learns Δ, which occurs with probability at most $1/p^r$.

In conclusion, any unbounded environment $\mathcal{Z}$ cannot distinguish between the real-world execution and ideal-world execution, except with probability $(t+3)/p^r$.

Malicious verifier. If $\mathcal{S}$ receives false from $\mathcal{F}_{\mathsf{ZK}}$, then it simply aborts. Otherwise, $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) In the preprocessing phase, $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ by recording the global key Δ and the keys for all the authenticated values, that are sent to this functionality by $\mathcal{A}$. $\mathcal{S}$ also receives $B^* \in \mathbb{F}_{p^r}$ from $\mathcal{A}$ when emulating $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$.

(2) $\mathcal{S}$ executes steps 4–5 of protocol $\Pi^{p,r}_{\mathsf{ZK}}$ by sending uniformly random δ_i for each $i \in \mathcal{I}_{\mathsf{in}}$ and d_i for the i-th multiplication gate to adversary $\mathcal{A}$.

(3) $\mathcal{S}$ executes steps 6–7 of the protocol as an honest prover, except that sampling $V \leftarrow \mathbb{F}_{p^r}$ and computing $U := W - V \cdot \Delta$ where W is computed using Δ, B^* and the keys received from $\mathcal{A}$ following the protocol specification.

(4) In step 8 of the protocol, $\mathcal{S}$ computes k_h (based on the keys sent to $\mathcal{F}_{\mathsf{ext\text{-}sVOLE}}^{p,r}$ by $\mathcal{A}$) and then sets $m_h := k_h + \Delta$, where h is the single output wire. Then, $\mathcal{S}$ sends m_h to $\mathcal{A}$.

Since $\{\mu_i\}, \{\nu_i\}$ and A_1^* are uniformly random and perfectly hidden against the view of adversary $\mathcal{A}$, we easily obtain that the view of $\mathcal{A}$ simulated by $\mathcal{S}$ is distributed identically to its view in the real protocol execution. This completes the proof. □

In the protocol $\Pi_{\mathsf{ZK}}^{p,r}$ shown in Figure 3.4, if we set $p = 2^{61} - 1$ and $r = 1$, then the computation of χ^i for $i \in [t]$ is expensive (especially for large t). We can replace χ^i for $i \in [t]$ with independent uniform coefficient χ_i for $i \in [t]$ to obtain better computational efficiency. In this case, the verifier can send a random seed in $\{0,1\}^\lambda$ to the prover, and then both parties compute $\chi_1, \ldots, \chi_t$ using the seed and a random oracle. Now, the soundness error is bounded by $q/2^\lambda + 4/p^r$, where q is an upper bound of the number of random oracle queries made by the adversary. [2] When using the random oracle, the security is guaranteed in the computational sense.

Non-interactive online phase. In the online phase of our protocol $\Pi_{\mathsf{ZK}}^{p,r}$, the verifier only sends a random coefficient χ to the prover. Thus, the communication cost is one field element per multiplication gate even without random oracle. But the online phase needs communication of three rounds.

We can use the Fiat-Shamir heuristic to make the online phase *non-interactive* at the cost of that the information-theoretic security is degraded to the computation security. Specifically, both parties can compute $\chi \in \mathbb{F}_{p^r}$ as $\mathsf{H}(d_1, \ldots, d_t)$, where $\mathsf{H} : \{0,1\}^* \to \mathbb{F}_{p^r}$ is

[2] This bound can be easily obtained by adapting the proof of Theorem 6 and computing the probability that the adversary succeeds to guess the seed.

a cryptographic hash function modeled as a random oracle and $p^r \geq 2^\lambda$. In this case, the soundness error for the batch check of multiplication gates together with the correctness of the single output is now bounded by $(q_{\mathsf{H}} + t + 3)/p^r \leq (q_{\mathsf{H}} + t + 3)/2^\lambda$, where q_{H} is an upper bound of the number of H queries made by the adversary. When we set $p = 2$ and $r = 128$, we can obtain a non-interactive online phase with a blazing-fast computation given hardware-instruction support.

3.3. Zero-Knowledge For Polynomial Sets

The previous verification of a multiplication gate can also be viewed as the verification of a simple degree-2 polynomial $f(x_0, x_1) := x_0 \cdot x_1$. It can be simply generalized to verify a inner product defined as $f(\boldsymbol{x}_0, \boldsymbol{x}_1) := \sum_i \boldsymbol{x}_0[i] \cdot \boldsymbol{x}_1[i]$, in which $(\boldsymbol{x}_0, \boldsymbol{x}_1)$ are committed vectors. In this section, we discuss how to prove a set of arbitrary degree-d polynomial $(f_1(\boldsymbol{x}), \ldots, f_t(\boldsymbol{x}))$ with communication overhead $O(|\boldsymbol{x}| + d)$.

We show the detailed ZKP protocol for polynomial sets in Figure 3.5. Similar to the circuit-based ZK protocol described in Section 3.2.2, our polynomial-based ZK protocol is also constant-round. In Section 3.3.1, we provide the formal security proof of the polynomial-based ZK protocol. In Section 3.3.2, we also present some practical applications of the polynomial-based ZK protocol, including how to optimize the zero-knowledge proofs for proving matrix multiplication, proving knowledge of a solution to an SIS problem, proving integer multiplication over a ring and proving the circuits with some level of weak uniformity.

Protocol $\Pi^{p,r}_{\mathsf{polyZK}}$

Inputs: The prover $\mathcal{P}$ and verifier $\mathcal{V}$ hold t number of d-degree polynomials $f_1, \ldots, f_t$ all over n variables. Each polynomial f_i is represented as $f_i = \sum_{h\in[0,d]} f_{i,h}$ where all terms in $f_{i,h}$ have degree h. $\mathcal{P}$ also holds a witness $\boldsymbol{w} \in \mathbb{F}_p^n$, such that $f_i(\boldsymbol{w}) = 0$ for all $i \in [t]$.

Preprocessing phase: Both polynomials and witness are unknown.

(1) $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$, which returns $\Delta \in \mathbb{F}_{p^r}$ to $\mathcal{V}$.
(2) $\mathcal{P}$ and $\mathcal{V}$ send (extend, n) to $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$, which returns authenticated values $\{[s_i]\}_{i\in[n]}$ to the parties.
(3) $\mathcal{P}$ and $\mathcal{V}$ also send $(\mathsf{VOPE}, d-1)$ to $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$, which returns $\{A^*_h\}_{h\in[0,d-1]}$ to $\mathcal{P}$, B^* to $\mathcal{V}$, such that $\sum_{h\in[0,d-1]} A^*_h \cdot \Delta^h = B^*$.

Online phase: Now the polynomials and witness are known.

(4) For $i \in [n]$, $\mathcal{P}$ sends $\delta_i := w_i - s_i \in \mathbb{F}_p$ to $\mathcal{V}$, and both parties compute $[w_i] := [s_i] + \delta_i$. Note that $k_i = m_i + w_i \cdot \Delta$ for $i \in [n]$.
(5) From $i = 1$ to t, for the i-th polynomial f_i, two parties perform the following:
   - $\mathcal{V}$ computes $B_i := \sum_{h\in[0,d]} f_{i,h}(k_1, \ldots, k_n) \cdot \Delta^{d-h}$.
   - $\mathcal{P}$ defines a univariate d-degree polynomial over field $\mathbb{F}_{p^r}$ as $g_i(x) = \sum_{h\in[0,d]} f_{i,h}(m_1 + w_1 \cdot x, \ldots, m_n + w_n \cdot x) \cdot x^{d-h}$, and computes the coefficients $\{A_{i,h}\}_{h\in[0,d]}$ such that $g_i(x) = \sum_{h\in[0,d]} A_{i,h} \cdot x^h$. Since $A_{i,d} = f_i(w_1, \ldots, w_n) = 0$, $g_i(x)$ can be written as $\sum_{h\in[0,d-1]} A_{i,h} \cdot x^h$.
(6) Two parties perform the following to check that $\sum_{h\in[0,d-1]} A_{i,h} \cdot \Delta^h = B_i$ for all $i \in [t]$:
   (a) $\mathcal{V}$ samples $\chi \leftarrow \mathbb{F}_{p^r}$ and sends it to $\mathcal{P}$.
   (b) For all $h \in [0, d-1]$, $\mathcal{P}$ computes $U_h := \sum_{i\in[t]} A_{i,h} \cdot \chi^i + A^*_h$ and sends it to $\mathcal{V}$.
   (c) $\mathcal{V}$ computes $W := \sum_{i\in[t]} B_i \cdot \chi^i + B^*$ and checks that $W = \sum_{h\in[0,d-1]} U_h \cdot \Delta^h$. If the check fails, $\mathcal{V}$ outputs false; otherwise it outputs true.

Figure 3.5. Zero-knowledge for polynomial satisfiability over any field in the $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$-hybrid model.

Computing polynomial coefficients. In the ZK protocol shown in Figure 3.5, prover $\mathcal{P}$ can compute the coefficients $\{A_{i,h}\}_{h\in[0,d-1]}$ of polynomial $g_i(x)$ for $i \in [t]$ in the following *generic* way.

- $\mathcal{P}$ computes $y_{i,j} := g_i(\alpha_j)$ for $j \in [d+1]$, where $\alpha_1, \ldots, \alpha_{d+1}$ are any $d+1$ different fixed points over extension field $\mathbb{F}_{p^r}$.

- Then $\mathcal{P}$ computes $g_i(x) := \sum_{j\in[d+1]} y_{i,j} \cdot \delta_j(x)$, where $\delta_j(x) = \prod_{k\neq j} \frac{x-\alpha_k}{\alpha_j-\alpha_k}$ is a fixed d-degree polynomial that can be precomputed in the preprocessing phase.

In a lot of practical applications, the polynomials $\{g_i(x)\}$ are usually simple, and thus the coefficients can be computed efficiently without the need of using the above Lagrange interpolation approach.

Computational complexity. In the $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$-hybrid model, the computational cost of protocol $\Pi^{p,r}_{\mathsf{polyZK}}$ is dominated by polynomial evaluation (i.e., computing $\{B_i\}$ and $\{A_{i,h}\}$). We easily bound the computational complexities of prover $\mathcal{P}$ and verifier $\mathcal{V}$ by $O(tdc+dn)$ and $O(tc)$ respectively, where c is the maximum cost to evaluate any polynomial on a single point, and $O(dn)$ is the cost to compute $\{m_i + w_i \cdot \alpha_j\}_{i\in[n]}$ for $j \in [d+1]$. Here, we assume that the polynomial coefficients $\{A_{i,h}\}_{h\in[0,d]}$ for $i \in [t]$ are computed using the generic Lagrange interpolation approach described as above. For many practical applications, the computational complexity of the prover may be lower without using the generic approach. Let z be the maximum number of terms in all t polynomials. Then we have that $c = O(dz)$, as each term in any polynomial has a degree at most d. Therefore, the computational complexities of $\mathcal{P}$ and $\mathcal{V}$ can be bounded by $O(td^2z + dn)$ and $O(tdz)$, respectively.

3.3.1. Proof of Security

When both parties are honest, it is not hard to see that the verifier will always output true with probability 1. Specifically, from $k_i = m_i + w_i \cdot \Delta$ for $i \in [n]$, we have that $B_i = \sum_{h\in[0,d-1]} A_{i,h} \cdot \Delta^h$ for all $i \in [t]$. Together with $\sum_{h\in[0,d-1]} A^*_h \cdot \Delta^h = B^*$, we obtain

that the following holds:

$$\begin{aligned} W &= \sum_{i\in[t]} B_i \cdot \chi^i + B^* \\ &= \sum_{i\in[t]} \Big(\sum_{h\in[0,d-1]} A_{i,h} \cdot \Delta^h \Big) \cdot \chi^i + \sum_{h\in[0,d-1]} A^*_h \cdot \Delta^h \\ &= \sum_{h\in[0,d-1]} \Big(\sum_{i\in[t]} A_{i,h} \cdot \chi^i + A^*_h \Big) \cdot \Delta^h = \sum_{h\in[0,d-1]} U_h \cdot \Delta^h. \end{aligned}$$

Thus, our protocol shown in Figure 3.5 achieves perfect completeness.

Theorem 7. *Protocol* $\Pi^{p,r}_{\mathsf{polyZK}}$ *UC-realizes functionality* $\mathcal{F}_{\mathsf{ZK}}$ *that proves polynomial satisfiability in the* $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$*-hybrid model with soundness error* $(d+t)/p^r$ *and information-theoretic security.*

Proof. We first consider the case of a malicious prover (i.e., soundness and knowledge extraction) and then consider the case of a malicious verifier (i.e., zero knowledge). In each case, we construct a simulator $\mathcal{S}$, which is given access to $\mathcal{F}_{\mathsf{ZK}}$, runs the adversary $\mathcal{A}$ as a subroutine while emulating $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ for $\mathcal{A}$. We always implicitly assume that $\mathcal{S}$ passes all communication between adversary $\mathcal{A}$ and environment $\mathcal{Z}$.

Malicious prover. $\mathcal{S}$ emulates functionality $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ and interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ for $\mathcal{A}$ by choosing uniform $\Delta \in \mathbb{F}_{p^r}$, and recording all the values $\{s_i\}_{i\in[n]}$ and their corresponding MAC tags that are received by $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ from adversary $\mathcal{A}$. These values define the corresponding keys in the natural

way. When emulating $\mathcal{F}_{\mathsf{ext\text{-}sVOLE}}^{p,r}$, $\mathcal{S}$ also receives $\{A_h^*\}_{h\in[0,d-1]}$ from $\mathcal{A}$ and defines $B^* = \sum_{h\in[0,d-1]} A_h^* \cdot \Delta^h$.

(2) When $\mathcal{A}$ sends $\{\delta_i\}_{i\in[n]}$ in step 4, $\mathcal{S}$ extracts the witness as $w_i := \delta_i + s_i$ for $i \in [n]$.

(3) $\mathcal{S}$ executes the remaining part of protocol $\Pi_{\mathsf{polyZK}}^{p,r}$ as an honest verifier, using Δ and the keys defined in the first step. If the honest verifier outputs false, then $\mathcal{S}$ sends $\boldsymbol{w} = \perp$ and $\mathcal{C}$ to $\mathcal{F}_{\mathsf{ZK}}$ and aborts. If the honest verifier outputs true, $\mathcal{S}$ sends $\boldsymbol{w}$ and $\mathcal{C}$ to $\mathcal{F}_{\mathsf{ZK}}$ where $\boldsymbol{w} = (w_1, \ldots, w_n)$ is extracted by $\mathcal{S}$ as above.

It is easy to see that the view of the adversary simulated by $\mathcal{S}$ has the identical distribution as its view in the real-world execution. Whenever the honest verifier in the real-world execution outputs false, the honest verifier in the ideal-world execution outputs false as well (since $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathsf{ZK}}$ in this case). Therefore, we only need to bound the probability that the verifier in the real-world execution outputs true but the witness $\boldsymbol{w}$ sent by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{ZK}}$ satisfies that $f_i(\boldsymbol{w}) \neq 0$ for some $i \in [t]$. Below, we show that this happens with probability at most $(d+t)/p^r$.

Let $f_i(\boldsymbol{w}) = f_i(w_1, \ldots, w_n) = y_i$ with some $y_i \in \mathbb{F}_p$ for each $i \in [t]$, where $\boldsymbol{w} = (w_1, \ldots, w_n)$ is a vector extracted by $\mathcal{S}$. According to the definition of B_i for $i \in [t]$, we

have the following:

$$\begin{aligned} B_i &= \sum_{h\in[0,d]} f_{i,h}(k_1,\ldots,k_n)\cdot\Delta^{d-h} \\ &= \sum_{h\in[0,d]} f_{i,h}(m_1+w_1\cdot\Delta,\ldots,m_n+w_n\cdot\Delta)\cdot\Delta^{d-h} \\ &= \sum_{h\in[0,d-1]} A_{i,h}\cdot\Delta^h + y_i\cdot\Delta^d. \end{aligned}$$

In step 6, $\mathcal{S}$ receives $U'_h = U_h + E_h$ for $h \in [0, d-1]$ from adversary $\mathcal{A}$, where U_h is computed with $\boldsymbol{w}$ and the corresponding MACs following the protocol specification, and $E_h \in \mathbb{F}_{p^r}$ is an adversarially chosen error. Together with $B^* = \sum_{h\in[0,d-1]} A^*_h \cdot \Delta^h$, we obtain that the following equation holds:

$$\begin{aligned} W &= \sum_{i\in[t]} B_i\cdot\chi^i + B^* \\ &= \sum_{i\in[t]}\Big(\sum_{h\in[0,d-1]} A_{i,h}\cdot\Delta^h + y_i\cdot\Delta^d\Big)\cdot\chi^i + \sum_{h\in[0,d-1]} A^*_h\cdot\Delta^h \\ &= \Big(\sum_{i\in[t]} y_i\cdot\chi^i\Big)\cdot\Delta^d + \sum_{h\in[0,d-1]}\Big(\sum_{i\in[t]} A_{i,h}\cdot\chi^i + A^*_h\Big)\cdot\Delta^h \\ &= \Big(\sum_{i\in[t]} y_i\cdot\chi^i\Big)\cdot\Delta^d + \sum_{h\in[0,d-1]} U'_h\cdot\Delta^h - \sum_{h\in[0,d-1]} E_h\cdot\Delta^h. \end{aligned}$$

If the honest verifier outputs true, then we have $W = \sum_{h\in[0,d-1]} U'_h\cdot\Delta^h$. Therefore, we have the following:

$$\Big(\sum_{i\in[t]} y_i\cdot\chi^i\Big)\cdot\Delta^d - E_{d-1}\cdot\Delta^{d-1} - \cdots - E_1\cdot\Delta - E_0 = 0.$$

If $\sum_{i\in[t]} y_i \cdot \chi^i \neq 0$, the probability that the above equation holds is at most d/p^r, as $\Delta \in \mathbb{F}_{p^r}$ is uniformly random and kept secret from the adversary's view. In the following, we assume that $\sum_{i\in[t]} y_i \cdot \chi^i = 0$. If there exists some $i \in [t]$ such that $y_i \neq 0$, then that probability that $\sum_{i\in[t]} y_i \cdot \chi^i = 0$ is at most t/p^r, since $\chi \in \mathbb{F}_{p^r}$ is sampled uniformly at random after y_i for all $i \in [t]$ have been defined. Overall, the probability that the honest verifier outputs true but $f_i(\boldsymbol{w}) \neq 0$ for some $i \in [t]$ is bounded by $(d+t)/p^r$. In conclusion, any unbounded environment $\mathcal{Z}$ cannot distinguish between the real-world execution and ideal-world execution, except with probability $(d+t)/p^r$.

Malicious verifier. If $\mathcal{S}$ receives false from $\mathcal{F}_{\mathsf{ZK}}$, then it simply aborts. Otherwise, $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) In the preprocessing phase, $\mathcal{S}$ emulates $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$ by recording the global key Δ and the keys for all the authenticated values, which are received from adversary $\mathcal{A}$. Additionally, $\mathcal{S}$ also receives $B^* \in \mathbb{F}_{p^r}$ from $\mathcal{A}$ by emulating $\mathcal{F}^{p,r}_{\mathsf{ext\text{-}sVOLE}}$.

(2) $\mathcal{S}$ executes the step 4 of protocol $\Pi^{p,r}_{\mathsf{polyZK}}$ by sending uniform $\delta_i \in \mathbb{F}_p$ for $i \in [n]$ to adversary $\mathcal{A}$.

(3) For steps 5–6 of the ZK protocol, $\mathcal{S}$ computes W by using Δ, the keys and B^* received from $\mathcal{A}$ following the protocol description, and then samples $U_1, \ldots, U_{d-1} \leftarrow \mathbb{F}_{p^r}$ and computing $U_0 := W - \sum_{h\in[d-1]} U_h \cdot \Delta^h$. Then, $\mathcal{S}$ sends $U_0, \ldots, U_{d-1}$ to $\mathcal{A}$.

Note that $\{s_i\}_{i\in[n]}$ and $\{A^*_h\}_{h\in[d-1]}$ are uniform and kept secret from the view of adversary $\mathcal{A}$. Therefore, we easily obtain that the view of $\mathcal{A}$ simulated by $\mathcal{S}$ is distributed identically to its view in the real-world execution, which completes the proof. □

3.3.2. Optimizing Practical Applications

In the following applications, for the sake of simplicity, we always assume that $p^r \approx 2^\kappa$ as the Fiat-Shamir heuristic is assumed to be implicitly used in the applications. For the interactive case, we can also extend the applications to smaller extension fields, as long as the soundness error is assured negligible in ρ. In this section, the communication cost is computed in the sVOLE-hybrid model.[3]

Optimizing matrix multiplication. The prover wants to prove that $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$, where $\mathbf{A}, \mathbf{B} \in \mathbb{F}_p^{n \times n}$ are two secret matrices and $\mathbf{C} \in \mathbb{F}_p^{n \times n}$ is a public matrix known by the verifier. Using the circuit-based ZK protocol shown in Figure 3.1, this will need communication of $(2n^2 + n^3) \log p + 2\lambda$ bits.

Using the polynomial-based ZK protocol described in Figure 3.5, we can directly obtain a ZK protocol for inner product of two n-length vectors with communication of $2n \log p + 2\lambda$ bits, by defining a polynomial $f(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i \in [n]} x_i \cdot y_i$ for two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_p^n$. The communication complexity remains unchanged, even if the inner product of t vector pairs needs to be proved. This immediately gives us a ZK protocol for proving matrix multiplication with communication of $2n^2 \log p + 2\lambda$ bits, since a matrix multiplication can be written as the inner-product of n^2 vector pairs, where the communication of $2n^2 \log p$ bits is used to commit the entries in matrices $\mathbf{A}$ and $\mathbf{B}$ using sVOLE.

Proving solutions to lattice problems. Here, we assume the prover has a binary vector $\boldsymbol{s} \in \{0,1\}^m$ and intends to prove that $\mathbf{A} \cdot \boldsymbol{s} = \boldsymbol{t}$, with public matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and vector $\boldsymbol{t} \in \mathbb{Z}_q^n$. Here we assume that q is a prime. The SIS problem has been considered in

[3] We note that the communication for generating sVOLE correlations is *sublinear* to the number of resulting sVOLE correlations, using the recent LPN-based protocols [87, 25, 104, 95].

prior work such as [10, 95]. Our ZK protocol for polynomial sets can be used to prove the statements more efficiently. To commit to all input bits $s_1, \ldots, s_m$, the ZK protocol takes $m \log q$ bits. Then we need to show that 1) the prover indeed commits bits and that 2) the above linear equation holds. All of the above can be modeled as a set of $m+n$ polynomials with degree at most 2. In particular, we need to show that $s_i \cdot (s_i - 1) = s_i^2 - s_i = 0$ for all $i \in [m]$ and that $\sum_{j \in [m]} a_{i,j} \cdot s_j - t_i = 0$ for $i \in [n]$ where $a_{i,j}$ is the entry in the i-th row and j-th column of matrix $\mathbf{A}$. Since the polynomial degree is at most 2, the communication cost would be 2 elements over the extension field, each of size roughly κ bits. Therefore, the *total* communication cost is $m \log q + 2\kappa$ bits.

If the secret vector $\boldsymbol{s}$ is in $[-B, B]^m$ (with a small integer B) instead of a binary vector, which has also been addressed by prior work [20, 21, 48, 95], we would need a degree-$(2B+1)$ polynomial to prove that $f(s_i) = 0$ for all $i \in [m]$ where $f(x) = \Pi_{j \in [-B,B]}(x - j)$, with the total communication cost of $m \log q + (2B+1)\kappa$ bits.

In Section 3.4.2, we evaluate the concrete performance to demonstrate that our polynomial-based ZK protocol significantly outperforms prior work for proving knowledge of solutions to SIS.

Optimizing integer operations over a ring. Arithmetic operations over a field may often be sufficient for some applications. However, for applications where matching cleartext computation is crucial, the statement to be proven may require native computation over a ring $\mathbb{Z}_{2^n}$ such as $\mathbb{Z}_{2^{32}}$. In this case, one may naturally think about ring operations. Here we explore an alternative approach.

Our idea is to view integer multiplication over $\mathbb{Z}_{2^n}$ as a set of n polynomials that take $2n$ variables as input. In this case, the maximum degree for these polynomials is

$8n^2$, since the Boolean circuit for integer multiplication has a depth $2\log n + 3$ [33]. The communication cost for proving a set of integer multiplications would become linear to n, when the number of integer multiplications to be proven is large. In particular, if there are t integer multiplications to be proven with $t \approx 8n\lambda$, the amortized communication cost for each multiplication will be $3n + \frac{8n^2\lambda}{t} \approx 4n$ bits.

Optimizing for circuits with weak uniformity. Inspired by the above concrete examples, we summarize a blueprint to optimize circuits with some level of weak uniformity (i.e., the polynomial representations of sub-circuits are all bounded by some degree d).

Assume that the circuit to be proven is C, which contains t multiplication gates. We let $C_1, \ldots, C_k$ be k non-overlapping sub-circuits of C, such that for sub-circuit C_i, it has t_i multiplication gates without counting the multiplication gates that include the output wires of C_i. Each sub-circuit can be represented as a set of polynomials with the degree at most d. In a nutshell, our ZK protocol can be constructed as follows:

(1) Use sVOLE to commit to all the wire values in $C \setminus \{C_1, \ldots, C_k\}$, including the input wires go into the sub-circuits $C_1, \ldots, C_k$ and the output wires go out of these sub-circuits.

    This step takes $t - \sum_{i=1}^{k} t_i$ elements over $\mathbb{F}_p$ for communication.

(2) Prove that all multiplication gates in $C \setminus \{C_1, \ldots, C_k\}$ are computed correctly using our ZK protocol for circuit satisfiability shown in Figure 3.1.

    This step takes 2κ bits of communication.

(3) For each sub-circuit C_i, represent it as a set of polynomials, one for each output of C_i. Prove that all the polynomials with respect to all sub-circuits are computed correctly using our ZK protocol for polynomial satisfiability shown in Figure 3.5.

This step takes $d\kappa$ bits of communication, because all input and output wire values have already been committed with sVOLE.

In summary, the communication of the above protocol is essentially $(t - \sum_{i\in[k]} t_i) \log p + (d+2)\kappa$ bits. Now the task is really about how to "dig" as many "holes" as possible from C, while keeping all holes relatively simple. In practice, this is fairly common, as the real-life computations are written in succinct libraries, which means the same subroutine is often called for many times. We leave it as a future work to fully explore its potential and build an automated optimizer to maximize the practical efficiency.

3.4. Implementation and Benchmarking

We implemented our ZK protocols and report their performance. Unless otherwise specified, our evaluation results are reported over two Amazon EC2 machines of type `m5.2xlarge` with throttled network bandwidth (with latency about 0.1 ms) and one thread. Each machine has 8 virtual CPUs, which means 4 CPU cores. We instantiate the COT protocol (i.e., sVOLE with $p = 2$ and $r = \lambda$) and the VOLE protocol over a 61-bit field by using the recent protocols [**104**, **95**], and use SHA-256 as the cryptographic hash function modeled as a random oracle. We take advantage of hardware AES-NI and binary-field multiplication when applicable. All our implementations achieve computational security parameter $\kappa = 128$ and statistical security parameter $\rho \approx 100$ for Boolean circuits, and $\kappa = 128$ and $\rho \geq 40$ for arithmetic circuits over a 61-bit field where Mersenne prime $p = 2^{61} - 1$ is used as in prior work. The implementation is openly available at EMP [**94**].

Threads	Boolean Circuits				
	10 Mbps	20 Mbps	30 Mbps	50 Mbps	Local-host
1	4.4 M	6.2 M	7.0 M	7.5 M	7.6 M
2	5.3 M	8.1 M	9.9 M	11.8 M	11.8 M
3	5.7 M	9.1 M	11.4 M	13.9 M	14.3 M
4	5.8 M	9.9 M	12.2 M	14.9 M	15.8 M
Threads	Arithmetic Circuits				
	100 Mbps	500 Mbps	1 Gbps	2 Gbps	Local-host
1	1.2 M	3.4 M	4.2 M	4.8 M	4.8 M
2	1.3 M	4.4 M	6.1 M	7.0 M	7.1 M
3	1.4 M	4.9 M	7.2 M	8.4 M	8.4 M
4	1.4 M	5.0 M	7.5 M	8.9 M	8.9 M

Table 3.1. **Benchmark the performance of our circuit-based ZK protocol.** The benchmark results are the number of AND/MULT gates per second that can be proven using our protocol, where "M" means "million". Benchmark was obtained with different network settings and number of threads.

3.4.1. Benchmarking Our Circuit-based ZK Proof

We benchmarked the performance of our ZK protocol by proving circuits with 3×10^8 AND/MULT gates. Similar to prior work [95, 8], we observe that the performance does not depend on the shape of the circuit and is linear to the circuit size; and thus we focus on the speed in terms of "million gates per second". In Table 3.1, we benchmarked the performance of our circuit-based ZK protocol under different network settings and number of threads. The performance of our protocol ranges from 4.4 million to 15.8 million AND gates per second (or from 1.2 million to 8.9 million multiplication gates per second), depending on the network setting and number of threads. When we increase the number of threads and/or the network bandwidth, we could see an increase in the performance. The computation becomes the efficiency bottleneck of our protocol for Boolean circuits (resp., arithmetic circuits) when the network bandwidth is increased to 50 Mbps (resp., 2 Gbps), and thus the performance is not improved much beyond that.

Protocol	Boolean Circuit		Arithmetic Circuit	
	Size	Speed	Size	Speed
Wolverine [95]	7	1.25 M/sec	4	0.66 M/sec
Mac'n'Cheese [8]	–	–	3	0.4 M/sec
LPZK [45]	–	–	1	–
QS	1	7.7 M/sec	1	4.8 M/sec

Table 3.2. **Comparing our work (QS) with prior related work.** Size represents the number of field elements to send for each multiplication gate, which is also the number of (s)VOLEs. Speed represents the number of multiplication gates that can be executed per second with unlimited bandwidth and a single thread.

Comparison with prior work. We compared the performance of our ZK protocol and prior related work in Table 3.2. Since Mac'n'Cheese [8] only reported the performance of their protocol with one thread and local-host, we compare the performance of all protocols using this setup. In the Boolean setting, we observe 6× improvement in computation and 7× improvement in communication compared to the state-of-the-art protocol Wolverine [95]. For arithmetic circuits, our protocol improves by at least 7× in computation and 3×–4× in communication compared to Wolverine and Mac'n'Cheese. Note that Wolverine studied the performance of their ZK protocol when used for DECO [107] and Blind CA [93], as well as other applications like Merkle trees, and proving bugs in a set of code snippets [66]. Our performance improvement directly translates to the improvements for all of these applications.

Stress-testing of our ZK protocol. We stress-test our circuit-based ZK protocol on the cheapest instance of Amazon EC2 that only costs 2 to 5 cents per hour, and

Instance Information			Boolean Circuits		Arithmetic Circuits	
Type	Price cents/hour	CPU	Speed gates/sec	Cost gates/cent	Speed gates/sec	Cost gates/cent
`c6g.medium`	1.9	ARM	5.3 M	10.0 B	2.2 M	4.1 B
`c5.large`	4.7	Intel	5.9 M	4.5 B	2.9 M	2.2 B
`c5a.large`	4.2	AMD	7.3 M	6.3 B	3.0 M	2.6 B

Table 3.3. **Performance of stress-testing our ZK protocol on different Amazon EC2 instances.** All instances have 2 vCPUs and 1 GB memory.

	Binary field $\mathbb{F}_2$			Large field $\mathbb{F}_{2^{61}-1}$		
Length of vectors	10^6	10^7	10^8	10^6	10^7	10^8
Process witness (s)	0.24	2.4	23.8	0.39	3.9	39.2
Prove inner product (ms)	36.6	69.3	423.5	42.8	100.3	703.8

Table 3.4. **Performance of our ZK protocol for inner product.** We report separately the cost to process the witness and the cost to prove the inner product after the witness was processed.

summarize the experimental results in Table 3.3.[4] For all protocol executions, we use only a single thread. The Boolean circuits (resp., arithmetic circuits) are tested under the network bandwidth of 20 Mbps (resp., 500 Mbps). Although the computational power and memory are limited, our protocol still achieves high throughput. The speed for computing Boolean circuits ranges from 5.3 million to 7.3 million gates per second and the speed for arithmetic circuits ranges from 2.2 million to 3 million gates per second. Taking the low cost into consideration, our ZK protocol is very *affordable*. The lowest cost to prove Boolean circuits is about 10 billion gates per cent; and roughly 2.2–4.1 billion gates per cent for arithmetic circuits.

[4]Price is based on AWS defined-duration spot instances. There are cheaper t3.medium, t3a.medium burstable instances, but the cost is higher than the instances in Table 3.3 unless the average CPU usage is kept below 20%.

Protocol	Execution Time	Communication
Spartan [89]	$\geq$ 5000 s	$\leq$ 100 KB
Virgo [108]	357 s	221 KB
Wolverine [95]	1627 s	34 GB
Mac′n′Cheese [8]	2684 s	25.8 GB
QuickSilver (Circuit)	316 s	8.6 GB
QuickSilver (Polynomial)	10 s	25.2 MB

Table 3.5. **Performance of proving matrix multiplication using various protocols.** All numbers are based on proving knowledge of two 1024×1024 matrices over a 61-bit field, whose product is a public matrix. The execution time for Wolverine and Mac′n′Cheese is based on local-host, while our protocols and Virgo are based on a 500 Mbps network. Spartan consumed 600 GB memory before crash, and thus we extrapolate the execution time based on a smaller proving instance. Our protocols use just 1 GB of memory, but Virgo needs 148 GB of memory.

3.4.2. Benchmarking Our Polynomial-based ZK Proof

While our ZK protocol for polynomial satisfiability is generic, here we focus on some useful applications that use low-degree polynomials to demonstrate how powerful it can be. We leave exploration of compiler-based optimization and more complicated examples as the future work. In all of the experiments below, we use the network bandwidth of 20 Mbps for a binary field and 500 Mbps for a 61-bit field, and always use a single thread.

Inner product. In this benchmark, the witness consists of two vectors of n field elements (namely $\boldsymbol{x}$ and $\boldsymbol{y}$), and the prover wants to prove that the inner product of two vectors $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i \in [n]} x_i \cdot y_i$ equals to some public value. We report the cost of processing the witness and the cost of proving the inner product separately in Table 3.4. We found that processing the witness could be free in a larger computation. This is because when using inner product as a sub-circuit in a larger circuit, the input witness of this subcircuit is the output of some prior computation and thus need not be processed again. In this case,

Protocol	ENS [48]	Wolverine [95]	QuickSilver
Communication	53 KB	32.8 KB	4.1 KB
Execution time	–	220 ms	2 ms

Table 3.6. **Performance comparison of our ZK protocol** *Wand* **vs. prior work for proving knowledge of an SIS solution.** The solution is assumed to be a ternary-vector and $n = 2048, m = 1024, \log q = 32$.

the cost of the ZK proof for inner product is simply the second line. We can see that even for proving inner product of two vectors of length 10^8, the cost is very small.

Matrix multiplication. We report the performance of our ZK protocol for proving matrix multiplication, and compare it with prior work in Table 3.5. We observe that our polynomial-based ZK protocol is 31× faster than our own circuit-based protocol, which is already faster than prior protocols. It also uses 340× less communication than our circuit-based protocol. Our proof size is still significantly larger than Spartan and Virgo, but our ZK protocols (*Wand*) benefit in other aspects including execution time and memory usage. We note that the prover time of GKR-style protocols like Virgo could be further improved based on the technique in interactive proofs [92]. We did not find any ZK proof that implements this technique, but anticipate that the prover time will be of the same order of magnitude when incorporating this technique into Virgo.

Proving knowledge of solutions to lattice-based problems. Here we focus on proving knowledge of a solution to a *short integer solution* (SIS) problem. We assume that the prover knows a vector $\boldsymbol{s} \in [-B, B]^m$, such that $\mathbf{A} \cdot \boldsymbol{s} = \boldsymbol{t}$, where both parties know the public matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and vector $\boldsymbol{t} \in \mathbb{Z}_q^n$ (here we assume that q is a prime). Checking the matrix multiplication is easy since the matrix $\mathbf{A}$ is public and thus the main

work is to check that all coordinates in $\boldsymbol{s}$ are bounded. This can be done by proving that $\Pi_{j\in[-B,B]}(s_i - j) = 0$ for all $i \in [m]$. In typical SIS problems, e.g., the one studied in the recent work [48], B is set to 1, resulting in a degree-3 polynomial. The checking procedure of our ZK protocol is essentially *free* compared to the cost of obtaining the committed input to the polynomial. We show the performance comparison in Table 3.6, where the execution time for [48] is not available from their paper. Due to our improved protocol for low-degree polynomials, our protocol outperforms prior work. When the solution is restricted to a binary vector, our ZK protocol $Wand$ is still very faster than the state-of-the-art protocol Wolverine, which outperforms other protocol [10].

CHAPTER 4

VOLE-ZK with Sublinear Communication

The VOLE-based ZKPs mentioned in Chapter 3 demonstrate concrete efficiency and scalability. However, they still require communication linear to the circuit size. In this section, we show how to reduce the communication cost of VOLE-ZK by generalizing the underlying IT-PAC to a polynomial commitment and adapt the verification protocol for it. In this chapter, we first show how to achieve sublinear communication for SIMD circuits, then generalize it to arbitrary circuits. We also discuss the implementation and performance evaluation.

4.1. Information-Theoretic Polynomial Authentication Codes

In this section, we present the notion of information-theoretic polynomial authentication codes (IT-PACs). In our ZK protocol, IT-PACs are used as commitments on polynomials. We also describe a useful procedure to check consistency of the evaluation of two sets of polynomials at multiple points. Then, we present a concretely efficient protocol that generates a batch of IT-PACs.

4.1.1. Definition of IT-PACs

For the sake of simplicity, we define IT-PACs over a large field $\mathbb{F}$. Nevertheless, one can extend the definition of IT-PACs to a more general case in a straightforward manner, where the values are defined over a small field $\mathbb{F}$ (e.g., $\mathbb{F} = \mathbb{F}_2$) and authenticated over

a large extension field $\mathbb{K}$. Specifically, the definition of IT-PACs over a large field $\mathbb{F}$ is described as follows:

- **Commitments with IT-PACs.** A verifier $\mathcal{V}$ holds a uniform *global key* $\Delta \in \mathbb{F}$ and another uniform key $\Lambda \in \mathbb{F}$ referred to as a *polynomial key*. Both keys are reused across different IT-PACs. An IT-PAC commitment on a degree-k polynomial $f(\cdot) = \sum_{i=0}^{k} c_i \cdot X^i \in \mathbb{F}[X]$ is denoted by $[f(\cdot)]$, where $\mathcal{P}$ holds a polynomial $f(\cdot) \in \mathbb{F}[X]$ and an MAC $\mathsf{M} \in \mathbb{F}$, while $\mathcal{V}$ holds keys $\mathsf{K}, \Delta, \Lambda \in \mathbb{F}$, such that $\mathsf{M} = \mathsf{K} + f(\Lambda) \cdot \Delta$. As in IT-MACs, the key K is also called as a *local key*. We could also consider an IT-PAC on polynomial $f(\cdot)$ as an IT-MAC on value $f(\Lambda)$.

- **Opening.** To open a commitment $[f(\cdot)]$, $\mathcal{P}$ sends $(f(\cdot), \mathsf{M})$ to $\mathcal{V}$, who checks whether $\mathsf{M} = \mathsf{K} + f(\Lambda) \cdot \Delta$.

  If the malicious $\mathcal{P}$ cheats, it must either forge an MAC $\mathsf{M}' = \mathsf{K} + f'(\Lambda) \cdot \Delta$ on message $f'(\Lambda) \neq f(\Lambda)$ with probability at most $1/|\mathbb{F}|$, or find a different polynomial $f'(\cdot) \neq f(\cdot)$ such that $f'(\Lambda) = f(\Lambda)$. If the second case occurs, then $f'(\cdot) - f(\cdot)$ is a non-zero polynomial of degree at most k, and thus the probability that it is equal to 0 at a random point Λ is at most $k/|\mathbb{F}|$, according to the Schwartz–Zippel lemma. Therefore, the probability that $\mathcal{P}$ succeeds to cheat is at most $(k+1)/|\mathbb{F}|$.

- **Linear combination.** Similar to IT-MACs, IT-PACs are also additively homomorphic. Specifically, given the public coefficients $c_0, c_1, \ldots, c_\ell \in \mathbb{F}$ and IT-PACs $[f_1(\cdot)], \ldots, [f_\ell(\cdot)]$, $\mathcal{P}$ and $\mathcal{V}$ can *locally* compute $[g(\cdot)] = \sum_{i=1}^{\ell} c_i \cdot [f_i(\cdot)] + c_0$, where $g(\cdot) = \sum_{i=1}^{\ell} c_i \cdot f_i(\cdot) + c_0$, $\mathsf{M}_g = \sum_{i=1}^{\ell} c_i \cdot \mathsf{M}_{f_i}$ and $\mathsf{K}_g = \sum_{i=1}^{\ell} c_i \cdot \mathsf{K}_{f_i} - c_0 \cdot \Delta$.

Procedures $\mathsf{BatchCheck}_{k,m,t}$

Inputs. Two parties $\mathcal{P}$ and $\mathcal{V}$ hold the following inputs:

- Two sets of IT-PACs $\{[f_1(\cdot)], \ldots, [f_\ell(\cdot)]\}$ and $\{[g_1(\cdot)], \ldots, [g_\ell(\cdot)]\}$ where $f_i()$ is a degree-k polynomial and $g_i()$ is a degree-m polynomial for $i \in [1, \ell]$.
- Let $\{\alpha_1, \ldots, \alpha_t\}$ and $\{\beta_1, \ldots, \beta_t\}$ be two sets of public elements over $\mathbb{F}$. Let $\mathsf{H} : \{0,1\}^\lambda \to \mathbb{F}^\ell$ be a random oracle.

Consistency check. $\mathcal{P}$ and $\mathcal{V}$ check $f_j(\alpha_i) = g_j(\beta_i)$ for all $i \in [1, t], j \in [1, \ell]$ as follows.

Linear combination phase: Before the polynomial key Λ is opened, $\mathcal{P}$ and $\mathcal{V}$ execute as follows.

(1) $\mathcal{P}$ samples two random polynomials $r(\cdot)$ and $s(\cdot)$ of respective degrees k and m in $\mathbb{F}[X]$ such that $r(\alpha_i) = s(\beta_i)$ for $i \in [1, t]$. Then, $\mathcal{P}$ and $\mathcal{V}$ generate the corresponding IT-PACs $[r(\cdot)]$ and $[s(\cdot)]$.

(2) $\mathcal{V}$ samples $\mathsf{seed} \leftarrow \{0,1\}^\lambda$ and sends it to $\mathcal{P}$. Then, two parties compute $(\chi_1, \ldots, \chi_\ell) := \mathsf{H}(\mathsf{seed}) \in \mathbb{F}^\ell$.

(3) $\mathcal{P}$ and $\mathcal{V}$ locally compute $[f(\cdot)] := \sum_{j=1}^{\ell} \chi_j \cdot [f_j(\cdot)] + [r(\cdot)]$ and $[g(\cdot)] := \sum_{j=1}^{\ell} \chi_j \cdot [g_j(\cdot)] + [s(\cdot)]$. Then, $\mathcal{P}$ sends the polynomial pair $(f(\cdot), g(\cdot))$ to $\mathcal{V}$, who checks that $f(\cdot), g(\cdot)$ have the degrees k and m respectively and $f(\alpha_i) = g(\beta_i)$ for all $i \in [1, t]$. If the check fails, $\mathcal{V}$ aborts.

Check phase:

(4) $\mathcal{P}$ and $\mathcal{V}$ locally compute $[\mu] := [f(\Lambda)] - f(\Lambda)$ and $[\nu] := [g(\Lambda)] - g(\Lambda)$. Then, two parties run $\mathsf{CheckZero}([\mu], [\nu])$ to check that $\mu = 0$ and $\nu = 0$. If the check fails, then $\mathcal{V}$ aborts, else $\mathcal{V}$ accepts.

Figure 4.1. Procedure for checking the consistency of polynomial evaluation for two sets of IT-PACs.

For a public polynomial $f(\cdot) \in \mathbb{F}[X]$, $\mathcal{P}$ and $\mathcal{V}$ can directly define the MAC and key in the IT-PAC $[f(\cdot)]$ as 0 and $-f(\Lambda) \cdot \Delta$ respectively without any interaction. Given $k+1$ distinct elements $\alpha_1, \ldots, \alpha_{k+1} \in \mathbb{F}$, an IT-PAC $[f(\cdot)]$ also commits to the values $f(\alpha_1), \ldots, f(\alpha_{k+1})$, since $k+1$ values uniquely determine a degree-k polynomial and vice versa.

4.1.2. Batch Check of Polynomial Evaluation

We present an interactive procedure to check the consistency of polynomial evaluation of two sets of IT-PACs at a single point or multiple points. The consistency check is done in a batch with communication independent of the number of IT-PACs. Specifically, given two sets of public field elements $\{\alpha_i\}_{i\in[1,t]}$ and $\{\beta_i\}_{i\in[1,t]}$ and two sets of IT-PACs $\{[f_j(\cdot)]\}_{j\in[1,\ell]}$ and $\{[g_j(\cdot)]\}_{j\in[1,\ell]}$ as input, this procedure allows to check that $f_j(\alpha_i) = g_j(\beta_i)$ for each $i \in [1,t], j \in [1,\ell]$, where $f_j(\cdot)$ is a degree-k polynomial and $g_j(\cdot)$ is a degree-m polynomial for $j \in [1,\ell]$. We require that the procedure does *not* reveal any secret information on these polynomials except that the equalities hold. This is realized by first generating two random IT-PAC commitments $[r(\cdot)]$ and $[s(\cdot)]$ such that $r(\alpha_i) = s(\beta_i)$ for $i \in [1,t]$ if $\mathcal{P}$ is honest, and then opening $[f(\cdot)] = \sum_{i=1}^{\ell} \chi_i \cdot [f_i(\cdot)] + [r(\cdot)]$ and $[g(\cdot)] = \sum_{i=1}^{\ell} \chi_i \cdot [g_i(\cdot)] + [s(\cdot)]$ where $\chi_1, \ldots, \chi_\ell$ are public coefficients sampled at random by $\mathcal{V}$. To achieve better communication efficiency, we can use a random oracle to compress these public coefficients into a random seed. We denote the consistency-check procedure by $\mathsf{BatchCheck}$, which is described in Figure 4.1. We let $\mathcal{P}$ send two polynomials $f(\cdot)$ and $g(\cdot)$ to $\mathcal{V}$, meaning that $\mathcal{P}$ sends the coefficients of the two polynomials to $\mathcal{V}$. The communication complexity of this procedure is $O(k+m)$.

In the following, we give an important lemma, which will be used in the security proof of our ZK protocol.

Lemma 4. *Let $\mathcal{P}$ be a malicious party who interacts with an honest verifier $\mathcal{V}$ during the execution of* $\mathsf{BatchCheck}$. *Let* H *be a random oracle. If there exists some $i \in [1,t], j \in$*

$[1,\ell]$ *such that* $f_j(\alpha_i) \neq g_j(\beta_i)$*, then the probability that* $\mathcal{V}$ *accepts at the end of* BatchCheck *is at most* $\frac{\max\{k,m\}+2}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

Proof. Let $f(\cdot)$ and $g(\cdot)$ be the polynomials committed in IT-PACs $[f(\cdot)]$ and $[g(\cdot)]$ respectively, where $[f(\cdot)] = \sum_{j\in[1,\ell]} \chi_j \cdot [f_j(\cdot)] + [r(\cdot)]$ and $[g(\cdot)] = \sum_{j\in[1,\ell]} \chi_j \cdot [g_j(\cdot)] + [s(\cdot)]$. This means that $f(\cdot) = \sum_{j\in[1,\ell]} \chi_j \cdot f_j(\cdot) + r(\cdot) \in \mathbb{F}[X]$ and $g(\cdot) = \sum_{j\in[1,\ell]} \chi_j \cdot g_j(\cdot) + s(\cdot) \in \mathbb{F}[X]$. Let $f'(\cdot)$ and $g'(\cdot)$ be two polynomials opened by the malicious $\mathcal{P}$ in the step 3 of the BatchCheck procedure.

Let E_1 be the event that $f'(\Lambda) \neq f(\Lambda)$ or $g'(\Lambda) \neq g(\Lambda)$ but $\mathcal{V}$ accepts in the CheckZero procedure. According to the security of CheckZero, we have that $\Pr[E_1] \leq \frac{1}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

Let E_2 be the event that $f'(\Lambda) = f(\Lambda)$ and $g'(\Lambda) = g(\Lambda)$ but $f'(\cdot) \neq f(\cdot)$ or $g'(\cdot) \neq g(\cdot)$. Note that the polynomials $f'(\cdot)$ and $g'(\cdot)$ are opened by malicious $\mathcal{P}$ before the polynomial key Λ known by $\mathcal{P}$. At least one of two polynomials $f'(\cdot) - f(\cdot)$ and $g'(\cdot) - g(\cdot)$ is non-zero, and the degrees of $f'(\cdot) - f(\cdot)$ and $g'(\cdot) - g(\cdot)$ are bounded by k and m respectively. Therefore, for a uniform $\Lambda \in \mathbb{F}$, the probability that $f'(\Lambda) - f(\Lambda) = 0$ and $g'(\Lambda) - g(\Lambda) = 0$ is at most $\frac{\max\{k,m\}}{|\mathbb{F}|}$. In other words, $\Pr[E_2] \leq \frac{\max\{k,m\}}{|\mathbb{F}|}$.

Let E_3 be the event that there exists some $i \in [1,t], j \in [1,\ell]$ such that $f_j(\alpha_i) \neq g_j(\beta_i)$ but $\mathcal{V}$ accepts at the end of BatchCheck. We assume that both E_1 and E_2 do not happen. Thus, we have that $f'(\cdot) = \sum_{j\in[1,\ell]} \chi_j \cdot f_j(\cdot) + r(\cdot)$ or $g'(\cdot) = g(\cdot) = \sum_{j\in[1,\ell]} \chi_j \cdot g_j(\cdot) + s(\cdot)$. Since $\mathcal{V}$ accepts, we obtain that $f'(\alpha_i) = g'(\alpha_i)$ for $i \in [1,t]$. Therefore, $\sum_{j\in[1,\ell]} \chi_j \cdot (f_j(\alpha_i) - g_j(\alpha_i)) + (r(\alpha_i) - s(\alpha_i)) = 0$ for each $i \in [1,t]$. If the malicious $\mathcal{P}$ does not make a query seed to random oracle H before receiving seed, then $\chi_1, \ldots, \chi_\ell$ are determined after the polynomials $\{f_j(\cdot), g_j(\cdot)\}_{j=1}^{\ell}$, $r(\cdot)$ and $s(\cdot)$ have already been defined, and thus are independent of these polynomials. In this case we have that $\Pr[E_3 \,|\, \neg(E_1 \cup E_2)] \leq \frac{1}{|\mathbb{F}|}$.

The probability that $\mathcal{P}$ queried seed to random oracle H before $\mathcal{V}$ sends seed to $\mathcal{P}$ is at most $\frac{q}{2^\lambda}$, where q is the number of queries to random oracle H. Together, we obtain that $\Pr[E_3 \,|\, \neg(E_1 \cup E_2)] \leq \frac{1}{|\mathbb{F}|} + \frac{q}{2^\lambda} = \frac{1}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

Overall, we have the following:

$$\begin{aligned}
\Pr[E_3] &= \Pr[E_3 \,|\, E_1 \cup E_2] \cdot \Pr[E_1 \cup E_2] + \Pr[E_3 \,|\, \neg(E_1 \cup E_2)] \cdot \Pr[\neg(E_1 \cup E_2)] \\
&\leq \Pr[E_1 \cup E_2] + \Pr[E_3 \,|\, \neg(E_1 \cup E_2)] \\
&\leq \Pr[E_1] + \Pr[E_2] + \Pr[E_3 \,|\, \neg(E_1 \cup E_2)] \\
&\leq \frac{\max\{k, m\} + 2}{|\mathbb{F}|} + \mathsf{negl}(\lambda),
\end{aligned}$$

which completes the proof. □

We can easily extend the $\mathsf{BatchCheck}$ procedure shown in Figure 4.1 to check the correctness of opening ℓ degree-k polynomials to the values of these polynomials at t different points. Specifically, $\mathcal{P}$ can send $(f_j(\alpha_1), \ldots, f_j(\alpha_t))$ for each $j \in [1, \ell]$ to $\mathcal{V}$. Both parties locally compute an IT-PAC $[g_j(\cdot)]$ for each $j \in [1, \ell]$, where $g_j(\cdot)$ is a public degree-$(t-1)$ polynomial reconstructed from the values $f_j(\alpha_1), \ldots, f_j(\alpha_t)$ using Lagrange interpolation. Then, $\mathcal{P}$ and $\mathcal{V}$ run the $\mathsf{BatchCheck}_{k,(t-1),t}$ procedure to check $f_j(\alpha_i) = g_j(\alpha_i)$ for all $i \in [1, t], j \in [1, \ell]$. In the special case of $t = k + 1$, the above procedure is to check the correctness of opening the whole polynomials. In this case, it is unnecessary to mask the linear combination of input IT-PACs $[f_1(\cdot)], \ldots, [f_\ell(\cdot)]$ with a random IT-PAC. The extended procedure as described above may be useful in other ZK protocols and applications.

Protocol $\Pi^k_{\mathsf{IT\text{-}PAC}}$

Let $\mathsf{AHE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an additively homomorphic encryption scheme. Suppose that two parties $\mathcal{P}$ and $\mathcal{V}$ have already agreed a set of public parameters $\mathsf{par} = \mathsf{Setup}(1^\lambda)$. Let PRG be a PRG. Let ℓ be the number of IT-PACs to be generated in one execution and k be the maximum degree of the polynomials committed in each IT-PAC.

Initialize. Two parties $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{VOLE}}$, which returns a uniform $\Delta \in \mathbb{F}$ to $\mathcal{V}$.

Create and encrypt polynomial keys.

(1) $\mathcal{V}$ samples $\mathsf{seed} \leftarrow \{0,1\}^\lambda$, and then $\mathcal{V}$ and $\mathcal{P}$ call the (Commit) command of $\mathcal{F}_{\mathsf{Com}}$ with input seed, which returns a handle τ_1 to $\mathcal{P}$.

(2) $\mathcal{V}$ samples $\Lambda \leftarrow \mathbb{F}$ and runs $\langle \Lambda^i \rangle \leftarrow \mathsf{Enc}(\mathsf{sk}, \Lambda^i; r_i)$ for all $i \in [1,k]$ where $(r_0, r_1, \ldots, r_k) = \mathsf{PRG}(\mathsf{seed})$ and $\mathsf{sk} \leftarrow \mathsf{KeyGen}(\mathsf{par}; r_0)$. Then, $\mathcal{V}$ sends the AHE ciphertexts $\langle \Lambda^1 \rangle, \ldots, \langle \Lambda^k \rangle$ to $\mathcal{P}$.

Pre-generation of IT-PACs.

(3) $\mathcal{P}$ and $\mathcal{V}$ sends (extend, ℓ) to $\mathcal{F}_{\mathsf{VOLE}}$, which returns $\boldsymbol{u}, \boldsymbol{w} \in \mathbb{F}^\ell$ to $\mathcal{P}$ and $\boldsymbol{v} \in \mathbb{F}^\ell$ to $\mathcal{V}$, such that $\boldsymbol{w} = \boldsymbol{v} + \boldsymbol{u} \cdot \Delta$.

(4) For each $j \in [1,\ell]$, on input the j-th polynomial $f_j(\cdot) = \sum_{i=0}^k f_{j,i} \cdot X^i \in \mathbb{F}[X]$, $\mathcal{P}$ computes a ciphertext $\langle b_j \rangle$ with $u_j + b_j = f_j(\Lambda)$ via $\langle b_j \rangle = \sum_{i=1}^k f_{j,i} \cdot \langle \Lambda^i \rangle + f_{j,0} - u_j$.

(5) $\mathcal{P}$ and $\mathcal{V}$ call the (Commit) command of $\mathcal{F}_{\mathsf{Com}}$ with inputs $\langle b_1 \rangle, \ldots, \langle b_\ell \rangle$, which returns a handle τ_2 to $\mathcal{V}$.

Generation of IT-PACs and opening polynomial keys.

(6) $\mathcal{V}$ and $\mathcal{P}$ call the (Open) command of $\mathcal{F}_{\mathsf{Com}}$ on input τ_1, which returns (seed, τ_1) to $\mathcal{P}$. In parallel, $\mathcal{V}$ sends Λ to $\mathcal{P}$. Then, $\mathcal{P}$ computes $(r_0, r_1, \ldots, r_k) := \mathsf{PRG}(\mathsf{seed})$ and runs $\mathsf{sk} \leftarrow \mathsf{KeyGen}(\mathsf{par}; r_0)$. $\mathcal{P}$ checks that $\langle \Lambda^i \rangle = \mathsf{Enc}(\mathsf{sk}, \Lambda^i; r_i)$ for all $i \in [1,k]$, and aborts if the check fails. For each $j \in [1,\ell]$, $\mathcal{P}$ sets $\mathsf{M}_j := w_j$.

(7) $\mathcal{P}$ and $\mathcal{V}$ call the (Open) command of $\mathcal{F}_{\mathsf{Com}}$ on input τ_2, which returns $(\langle b_1 \rangle, \ldots, \langle b_\ell \rangle, \tau_2)$ to $\mathcal{V}$. Then, for each $j \in [1,\ell]$, $\mathcal{V}$ runs $b_j \leftarrow \mathsf{Dec}(\mathsf{sk}, \langle b_j \rangle)$, and then computes $\mathsf{K}_j := v_j - b_j \cdot \Delta \in \mathbb{F}$.

(8) For each $j \in [1,\ell]$, two parties obtain an IT-PAC $[f_j(\cdot)]$, where $\mathcal{P}$ holds $(f_j(\cdot), \mathsf{M}_j)$ and $\mathcal{V}$ holds K_j.

Figure 4.2. Protocol for generating IT-PACs without ZK proofs in the $(\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{Com}})$-hybrid model.

4.1.3. Efficient Protocol to Generate IT-PACs

Below, we present a concretely efficient protocol of generating IT-PACs. This protocol works in the $(\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{Com}})$-hybrid model, and adopts AHE to generate the additive shares

of the polynomial-evaluation values at the secret point Λ. Using the additive shares, a VOLE correlation can be locally transformed to a batch of IT-PACs.

For the AHE ciphertexts sent by a verifier $\mathcal{V}$, one can adopt a ZK proof to prove the validity of these ciphertexts. However, the usage of ZK proofs introduces significantly communication overhead (particularly, the size of ciphertexts need to be significantly larger to cover the so-called slack brought about by the ZK proofs). To reduce the communication overhead, we replace the ZK proof with the "commit-then-open" approach. In particular, the correctness of the ciphertexts produced by a verifier $\mathcal{V}$ is guaranteed by committing the randomness to generate the ciphertexts and then opening the randomness at some later point. The randomness can be generated with a random seed and a pseudo-random generator (PRG) to reduce the communication cost. When the ciphertexts sent by $\mathcal{V}$ may be *incorrect* before the randomness is opened, we let the party $\mathcal{P}$ first commit to its homomorphically computed ciphertexts and then open these ciphertexts after checking the correctness of the ciphertexts received from $\mathcal{V}$. This allows us to remove the possible leakage of secret polynomials, which is incurred by homomorphically performing polynomial evaluation upon incorrect ciphertexts. The "commit-then-open" approach is sufficient, as the polynomial key Λ will be always opened.

Based on the "commit-then-open" approach, the concretely efficient protocol for generating IT-PACs is described in Figure 4.2. While the initialization phase to generate a global key Δ needs to be run only once, other phases can be executed multiple times where every execution creates a fresh polynomial key Λ and a batch of IT-PACs under the key Λ. For generating ℓ IT-PACs on degree-k polynomials, the total communication complexity is $O(k + \ell)$, where the communication complexity for generating ℓ random

VOLE correlations is either $O(\sqrt{\ell}\log\ell)$ or $O(\log^2\ell)$ depending on the choices of LPN assumptions.

4.2. Zero-Knowledge Proofs with Sublinear Communication

4.2.1. Sublinear ZK Proof for SIMD Circuits

In Figure 4.3 and 4.4, we describe the details of our ZK protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ on proving satisfiability of SIMD circuits in the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model. This protocol also invokes $\Pi_{\mathsf{PAC}}^{(2B-2)}$ as a sub-protocol, and thus needs to call a commitment functionality $\mathcal{F}_{\mathsf{Com}}$ as well. When executing sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ to generate IT-PACs, the generation of the local keys held by the verifier $\mathcal{V}$ are delayed to the time point after the polynomial key Λ is opened. Thus, the computation of the local keys on the output IT-PACs of addition gates also has to be postponed. While prover $\mathcal{P}$ can execute the check phase of the underlying $\mathsf{BatchCheck}$ procedure *before* Λ is opened, the values from $\mathcal{P}$ could be checked by verifier $\mathcal{V}$ *after* Λ is opened and the local keys on the IT-PACs are computed.

When $\mathcal{F}_{\mathsf{VOLE}}$ is instantiated by the recent LPN-based VOLE protocol with sublinear communication, protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ has the communication complexity of $O(B + |\mathcal{C}|)$ for proving $(B, |\mathcal{C}|)$-SIMD circuits, where note that addition gates are *free* for our protocol. If the underlying DVZK proof has at most two rounds (e.g., DVZK is instantiated by [46, 100, 44]), the protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ has 6 rounds in the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model. Note that all invocations of sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ can be made in parallel and all $\mathsf{CheckZero}$ executions can be combined into one execution. Since the LPN-based VOLE protocol realizing $\mathcal{F}_{\mathsf{VOLE}}$ has constant rounds, our ZK protocol has also constant rounds.

Protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ (Part 1)

Inputs. The prover $\mathcal{P}$ and verifier $\mathcal{V}$ hold a generic circuit $\mathcal{C}$ over a large field $\mathbb{F}$, where $\mathcal{C}$ contains $n = |\mathcal{C}|$ multiplication gates and m input gates. $\mathcal{P}$ holds B witnesses $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_B \in \mathbb{F}^m$ such that $\mathcal{C}(\boldsymbol{w}_i) = 0$ for all $i \in [1, B]$. Let $\alpha_1, \ldots, \alpha_B \in \mathbb{F}$ be B distinct elements that are fixed for the whole protocol execution. Let $\delta_i(X) = \prod_{j \in [1,B], j \neq i} (X - \alpha_j)/(\alpha_i - \alpha_j) \in \mathbb{F}[X]$ be a degree-$(B-1)$ polynomial for each $i \in [1, B]$, which is referred to as a Lagrange polynomial.

Initialization. $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{VOLE}}$, which returns a uniform $\Delta \in \mathbb{F}$ to $\mathcal{V}$.

Circuit evaluation. For B executions of circuit $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{V}$ pack B same-type gates into a group in a straightforward way. In particular, for an index i, the parties pack the i-th input/output/multiplication/addition gates from all B executions of circuit $\mathcal{C}$ into a group.

(1) $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ shown in Figure 4.2 to create a uniform key $\Lambda \in \mathbb{F}$.

(2) The parties execute the following steps to process the inputs:
   (a) $\mathcal{P}$ and $\mathcal{V}$ send (extend, mB) to $\mathcal{F}_{\mathsf{VOLE}}$, which returns IT-MACs $\{[a_{i,j}]\}_{i \in [1,B], j \in [1,m]}$ to the parties.
   (b) For $i \in [1, B], j \in [1, m]$ (i.e., corresponding to the j-th circuit-input wire of the i-th execution of circuit $\mathcal{C}$), $\mathcal{P}$ sends $b_{i,j} := w_{i,j} - a_{i,j}$ to $\mathcal{V}$, and then both parties locally compute $[w_{i,j}] := [a_{i,j}] + b_{i,j}$.
   (c) For $j \in [1, m]$, for the j-th group of B input gates with input vector $(w_{1,j}, \ldots, w_{B,j})$, $\mathcal{P}$ defines a degree-$(B-1)$ polynomial $u_j(\cdot)$ such that $u_j(\alpha_i) = w_{i,j}$ for $i \in [1, B]$.
   (d) $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ to generate IT-PACs $[u_1(\cdot)], \ldots, [u_m(\cdot)]$.

(3) Following a predetermined topological order for circuit $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{V}$ execute as follows:
   (a) For each group of addition gates with input IT-PACs $[f(\cdot)]$ and $[g(\cdot)]$, both parties locally compute an output IT-PAC $[h(\cdot)] := [f(\cdot)] + [g(\cdot)]$.
   (b) For the j-th group of multiplication gates with input IT-PACs $[f_j(\cdot)]$ and $[g_j(\cdot)]$ where $j \in [1, n]$, $\mathcal{P}$ computes a degree-$(2B-2)$ polynomial $\widetilde{h}_j(\cdot) := f_j(\cdot) \cdot g_j(\cdot) \in \mathbb{F}[X]$ and a degree-$(B-1)$ polynomial $h_j(\cdot)$ such that $h_j(\alpha_i) = \widetilde{h}_j(\alpha_i)$ for all $i \in [1, B]$. Then, $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ to generate two IT-PACs $[h_j(\cdot)]$ and $[\widetilde{h}_j(\cdot)]$.

Figure 4.3. Sublinear zero-knowledge protocol for SIMD circuits in the $(\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{Com}})$-hybrid model (Part 1).

4.2.1.1. Proof of Security. For the security of protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$, we prove the following theorem. In this theorem, we assume that the soundness error ϵ_{dvzk} of the underlying ZK proof DVZK is at most $3/|\mathbb{F}| + \mathsf{negl}(\lambda)$, e.g., it is instantiated by QuickSilver [100].

Protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ (Part 2)

Correctness check of multiplication gates. $\mathcal{P}$ convinces $\mathcal{V}$ that the multiplication gates are evaluated correctly.

(4) $\mathcal{P}$ and $\mathcal{V}$ execute the linear-combination phase of the $\mathsf{BatchCheck}_{(B-1),(2B-2),B}$ procedure on inputs $\{[h_1(\cdot)], \ldots, [h_n(\cdot)]\}$ and $\{[\widetilde{h}_1(\cdot)], \ldots, [\widetilde{h}_n(\cdot)]\}$ to check that $h_j(\alpha_i) = \widetilde{h}_j(\alpha_i)$ for all $i \in [1, B], j \in [1, n]$.

(5) $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ to open Λ to $\mathcal{P}$, and then $\mathcal{V}$ can compute the local keys on all IT-PACs.

(6) Then, $\mathcal{P}$ and $\mathcal{V}$ execute the check phase of $\mathsf{BatchCheck}_{(B-1),(2B-2),B}$ to complete the above check. If the check fails, $\mathcal{V}$ aborts.

(7) $\mathcal{P}$ and $\mathcal{V}$ run a VOLE-based zero-knowledge proof

$$\mathsf{DVZK}\left\{([f_j(\Lambda)], [g_j(\Lambda)], [\widetilde{h}_j(\Lambda)])_{j\in[1,n]} \mid \forall j \in [1, n],\ \widetilde{h}_j(\Lambda) = f_j(\Lambda) \cdot g_j(\Lambda)\right\}.$$

Consistency check of input gates and output gates. $\mathcal{P}$ convinces the verifier $\mathcal{V}$ that $[u_j(\cdot)]$ is consistent to $([w_{1,j}], \ldots, [w_{B,j}])$ for $j \in [1, m]$, and the values on all output gates are 0.

(8) For each $j \in [1, m]$, $\mathcal{P}$ and $\mathcal{V}$ locally compute an IT-MAC $[z_j] := \sum_{i\in[1,B]} \delta_i(\Lambda) \cdot [w_{i,j}]$, and then run $\mathsf{CheckZero}([u_j(\Lambda)] - [z_j])$ to check $u_j(\Lambda) = z_j$. If the check fails, $\mathcal{V}$ aborts.

(9) Let $[v(\cdot)]$ be the IT-PAC associated with the output values of B executions of circuit $\mathcal{C}$. The parties $\mathcal{P}$ and $\mathcal{V}$ run $\mathsf{CheckZero}([v(\Lambda)])$ to check $v(\Lambda) = 0$, and $\mathcal{V}$ aborts if the check fails.

(10) If $\mathcal{V}$ will abort in some step, then $\mathcal{V}$ outputs false and aborts, else it outputs true.

Figure 4.4. Sublinear zero-knowledge protocol for SIMD circuits in the $(\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{Com}})$-hybrid model (Part 2).

Theorem 8. *If the* AHE *scheme satisfies the CPA security, degree-restriction and circuit privacy along with* PRG *is a pseudorandom generator, protocol* $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ *shown in Figure 4.3 and 4.4 UC-realizes functionality* $\mathcal{F}_{\mathsf{ZK}}^{B}$ *on* $(B, |\mathcal{C}|)$*-SIMD circuits in the* $(\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{Com}})$*-hybrid model and the random oracle model with soundness error at most* $\frac{2B+3}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

In this section, we give the formal proof of Theorem 8. First of all, we extend Lemma 4 from the information-theoretic setting to the computational setting. While Lemma 4 assumes that the polynomial key Λ is information-theoretically secure, we prove that the

result claimed in the lemma still holds, even if the malicious $\mathcal{P}$ is given the AHE ciphertexts $\mathsf{Enc}(\Lambda), \ldots, \mathsf{Enc}(\Lambda^h)$ with $h = \max\{k, m\}$ and the input IT-PACs of the $\mathsf{BatchCheck}$ procedure are generated by executing the protocol $\Pi^h_{\mathsf{IT\text{-}PAC}}$. In particular, we have the following lemma:

Lemma 5. *Let the IT-PACs used in the* $\mathsf{BatchCheck}$ *procedure be generated by two parties* $\mathcal{P}$ *and* $\mathcal{V}$ *by running protocol* $\Pi^h_{\mathsf{IT\text{-}PAC}}$ *shown in Figure 4.2 where* $h = \max\{k, m\}$. *Let* H *be a random oracle and* PRG *be a pseudorandom generator. If the* AHE *scheme is CPA secure and satisfies the degree-restriction property, then the probability that there exists some* $i \in [1, t], j \in [1, \ell]$ *such that* $f_j(\alpha_i) \neq g_j(\beta_i)$ *but an honest verifier* $\mathcal{V}$ *accepts at the end of* $\mathsf{BatchCheck}$ *is bounded by* $\frac{\max\{k,m\}+2}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

Proof. We consider that $\mathcal{P}$ is corrupted by a PPT malicious adversary $\mathcal{A}$ and $\mathcal{V}$ is honest. In this proof, we will reuse the notation used in the proof of Lemma 4. For example, the events E_1, E_2, E_3 and the polynomials $f'(\cdot), g'(\cdot)$ opened by $\mathcal{P}$ during the $\mathsf{BatchCheck}$ procedure. In the protocol $\Pi^h_{\mathsf{IT\text{-}PAC}}$, if $\mathcal{A}$ sends a ciphertext evaluated on a polynomial of degree $h' > h$ to the honest verifier $\mathcal{V}$, then it is easy to construct an algorithm who breaks the degree-restriction property of the AHE scheme.[1] In the following proof, we always assume that the degree of the polynomials committed in the IT-PACs generated by protocol $\Pi^h_{\mathsf{IT\text{-}PAC}}$ is bounded by $h = \max\{k, m\}$.

We consider a hybrid-check procedure denoted by $\mathsf{HybridCheck}$, which is the same as $\mathsf{BatchCheck}$ except that the verification of $\mathsf{CheckZero}([\mu], [\nu])$ is modified as follows:

[1] If the stronger notion of linear targeted malleability [16] is used, it is unnecessary to construct such an algorithm, and the simulation-based security excludes the computation on the ciphertexts of powers of Λ other than affine linear maps.

- Let M_μ and M_ν be the MACs involved in the IT-MACs $[\mu]$ and $[\nu]$ respectively, where $[\mu] = [f(\Lambda)] - f'(\Lambda)$ and $[\nu] = [g(\Lambda)] - g'(\Lambda)$. Use M_μ and M_ν to check the correctness of the hash value sent by $\mathcal{P}$.
- Check that $z_1 = f'(\Lambda)$ and $z_2 = g'(\Lambda)$, where z_1, z_2 are the values committed in the IT-MACs $[f(\Lambda)]$ and $[g(\Lambda)]$.

It is easy to see that the output of $\mathsf{HybridCheck}$ is identical to that of $\mathsf{CheckZero}$, unless event E_1 occurs with probability at most $\frac{1}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$. Note that global key $\Delta \in \mathbb{F}$ is uniformly random in the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model. Below, we assume that E_1 does not occur.

Based on the $\mathsf{HybridCheck}$ procedure, we construct the following hybrid protocol that is executed between adversary $\mathcal{A}$ and a PPT simulator $\mathcal{S}_{\mathsf{hybrid}}$:

(1) On behalf of an honest verifier, $\mathcal{S}_{\mathsf{hybrid}}$ simulates the phase to create and encrypt a uniform polynomial key Λ following the protocol description, except that the secret key sk and randomness used in the AHE ciphertexts are now sampled at random rather than being generated from a random seed. The resulting ciphertexts $\langle \Lambda \rangle, \ldots, \langle \Lambda^h \rangle$ are sent to $\mathcal{A}$.

(2) $\mathcal{S}_{\mathsf{hybrid}}$ emulates functionality $\mathcal{F}_{\mathsf{VOLE}}$ by recording the vectors $\boldsymbol{u}, \boldsymbol{w} \in \mathbb{F}^{2\ell+2}$ sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{VOLE}}$, where there are $2\ell + 2$ IT-PACs that need to be generated. Then, $\mathcal{S}_{\mathsf{hybrid}}$ sets $\mathsf{M}_i = w_i$ for $i \in [1, 2\ell + 2]$ as the MACs in the IT-PACs $\{([f_j(\cdot)], [g_j(\cdot)])\}_{j \in [1,\ell]}$ and $(r[\cdot], s[\cdot])$.

(3) $\mathcal{S}_{\mathsf{hybrid}}$ emulates functionality $\mathcal{F}_{\mathsf{Com}}$ by receiving the ciphertexts $\langle b_1 \rangle, \ldots, \langle b_{2\ell+2} \rangle$ sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{Com}}$ and sending a handle τ_2 to $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{hybrid}}$ obtains b_i by running $\mathsf{Dec}(\mathsf{sk}, \langle b_i \rangle)$ for $i \in [1, 2\ell + 2]$. Simulator $\mathcal{S}_{\mathsf{hybrid}}$ computes $y_i := b_i + u_i$

for all $i \in [1, 2\ell + 2]$ as the values $\{f_j(\Lambda), g_j(\Lambda)\}_{j\in[1,\ell]}$, $r(\Lambda)$ and $s(\Lambda)$, where the underlying polynomials are unknown for $\mathcal{S}_{\mathsf{hybrid}}$.

(4) Using (y_i, M_i) for all $i \in [1, 2\ell + 2]$, $\mathcal{S}_{\mathsf{hybrid}}$ runs the linear-combination phase of the $\mathsf{HybridCheck}$ procedure with $\mathcal{A}$, and then obtains the values $\mathsf{M}_\mu, \mathsf{M}_\nu, z_1, z_2 \in \mathbb{F}$ and the opened polynomials $f'(\cdot), g'(\cdot)$. Then, $\mathcal{S}_{\mathsf{hybrid}}$ checks that $f'(\alpha_i) = g'(\beta_i)$ for $i \in [1, t]$ and aborts if the check fails.

(5) $\mathcal{S}_{\mathsf{hybrid}}$ sends Λ to $\mathcal{A}$, and emulates $\mathcal{F}_{\mathsf{Com}}$ by receiving a handle τ_2 from $\mathcal{A}$.

(6) Using the key Λ, $(\mathsf{M}_\mu, \mathsf{M}_\nu, z_1, z_2)$ and $(f'(\cdot), g'(\cdot))$, $\mathcal{S}_{\mathsf{hybrid}}$ runs the check phase of the $\mathsf{HybridCheck}$ procedure with $\mathcal{A}$.

For the case of a malicious prover $\mathcal{P}$, we only need to consider the soundness. In this case, it is unnecessary to commit and open a random seed that is used to generate the secret key and randomness, as the verifier is always honest. Together with that the output of PRG is pseudorandom, the above hybrid protocol is computationally indistinguishable from the protocol $\Pi^h_{\mathsf{IT\text{-}PAC}}$ combining with $\mathsf{BatchCheck}$, when only considering the soundness.

In the following, we assume that event E_2 happens in the above hybrid protocol, meaning that $z_1 = f'(\Lambda)$ and $z_2 = g'(\Lambda)$ but $f'(\cdot) \neq f(\cdot)$ or $g'(\cdot) \neq g(\cdot)$ always hold, where z_1 and z_2 are computed by $\mathcal{S}_{\mathsf{hybrid}}$ as described above. Specifically, for a PPT adversary $\mathcal{A}$ who makes E_2 happen, we construct a PPT algorithm $\mathcal{B}$ who breaks the CPA security of the AHE scheme. Algorithm $\mathcal{B}$ interacts with $\mathcal{A}$ as follows:

(1) $\mathcal{B}$ samples two random values $\Lambda, \Lambda^* \leftarrow \mathbb{F}$, and then sends h message pairs $(\Lambda^i, (\Lambda^*)^i)$ for $i \in [1, h]$ to the CPA game. Then, $\mathcal{B}$ obtains h challenge ciphertexts $c_1, \ldots, c_h$ from the CPA game, where c_i is the encryption of either Λ^i or $(\Lambda^*)^i$ for $i \in [1, h]$. Algorithm $\mathcal{B}$ sends $c_1, \ldots, c_h$ to $\mathcal{A}$.

(2) $\mathcal{B}$ emulates functionality $\mathcal{F}_{\mathsf{VOLE}}$ by recording the vectors $\boldsymbol{u}, \boldsymbol{w}$ sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{VOLE}}$, and then sets $\mathsf{M}_i = w_i$ for $i \in [1, 2\ell + 2]$.

(3) $\mathcal{B}$ emulates $\mathcal{F}_{\mathsf{Com}}$ by receiving the ciphertexts $\langle b_1 \rangle, \ldots, \langle b_{2\ell+2} \rangle$ sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{Com}}$ and sending a handle τ_2 to $\mathcal{A}$.

(4) Using $\{\mathsf{M}_i\}_{i\in[1,2\ell+2]}$, $\mathcal{B}$ runs the linear-combination phase of the HybridCheck procedure with $\mathcal{A}$, and then obtains the values $\mathsf{M}_\mu, \mathsf{M}_\nu \in \mathbb{F}$ and the opened polynomials $f'(\cdot), g'(\cdot)$. Then, $\mathcal{B}$ checks that $f'(\alpha_i) = g'(\beta_i)$ for $i \in [1, t]$ and aborts if the check fails.

(5) Using Λ, $(\mathsf{M}_\mu, \mathsf{M}_\nu)$ and $(f'(\cdot), g'(\cdot))$, $\mathcal{B}$ runs the check phase of the HybridCheck procedure with $\mathcal{A}$. In particular, $\mathcal{B}$ always uses Λ to check that $z_1 = f'(\Lambda)$ and $z_2 = g'(\Lambda)$, ignoring which key (Λ or Λ^*) is used to compute $\{c_i\}_{i\in[1,h]}$. Actually, this check is *implicit* and assumed to be passed as E_2 occurs. Additionally, $\mathcal{B}$ uses the MACs M_μ and M_ν to check the correctness of the hash value sent by $\mathcal{A}$ in the CheckZero procedure.

In the case that event E_2 occurs, $\mathcal{B}$ behaves just like as $\mathcal{S}_{\mathsf{hybrid}}$, except for the encryption of a polynomial key. If the challenge ciphertexts $c_1, \ldots, c_h$ are the encryption of the powers of Λ, then the protocol execution simulated by $\mathcal{B}$ is the same as the above hybrid protocol. Otherwise, the key Λ used in the HybridCheck procedure is independent of the adversary's view, and thus is information-theoretically secure. From the proof of Lemma 4, we have that the event E_2 occurs with probability at most $\frac{\max\{k,m\}}{|\mathbb{F}|}$ in this case. Therefore, the successful probability of adversary $\mathcal{A}$ in the above hybrid protocol is bounded by $\frac{\max\{k,m\}}{|\mathbb{F}|} + \epsilon_{\mathsf{cpa}}$, where ϵ_{cpa} is the advantage of a PPT adversary to break the CPA security of the AHE scheme. Overall, the probability, that event E_2 occurs

in the $\mathsf{BatchCheck}$ procedure for the IT-PACs generated by protocol $\Pi^{h}_{\mathsf{IT\text{-}PAC}}$, is at most $\frac{\max\{k,m\}}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

Given the probabilities that the events E_1 and E_2 happen, we can easily obtain the probability that event E_3 occurs following the proof of Lemma 4. In particular, we have that $\Pr[E_3] \leq \frac{\max\{k,m\}+2}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$, which completes the proof. □

Theorem 9 (Theorem 8, restated)**.** *If the* AHE *scheme satisfies the CPA security, degree-restriction and circuit privacy along with* PRG *is a pseudorandom generator, protocol* $\Pi^{\mathsf{SIMD}}_{\mathsf{ZK}}$ *shown in Figure 4.3 and 4.4 UC-realizes functionality* $\mathcal{F}^{B}_{\mathsf{ZK}}$ *on* $(B, |\mathcal{C}|)$*-SIMD circuits in the* $(\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{Com}})$*-hybrid model and the random oracle model with soundness error at most* $\frac{2B+3}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

Proof. We first consider the case of a malicious prover (i.e., soundness) and then consider the case of a malicious verifier (i.e., zero knowledge). In each case, we construct a PPT simulator $\mathcal{S}_{\mathsf{ZK}}$ given access to functionality $\mathcal{F}^{B}_{\mathsf{ZK}}$, and running a PPT adversary $\mathcal{A}$ as a subroutine while emulating $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{Com}}$ for $\mathcal{A}$. For each case, we show that no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution. We always implicitly assume that $\mathcal{S}_{\mathsf{ZK}}$ passes all communication between adversary $\mathcal{A}$ and environment $\mathcal{Z}$. Let $\mathcal{S}_{\mathsf{DVZK}}$ be a PPT simulator for the underlying VOLE-based ZK proof DVZK.

Malicious prover. Firstly, we construct a PPT simulator $\mathcal{S}_{\mathsf{PAC}}$ to simulate the adversary's view in the execution of sub-protocol $\Pi^{(2B-2)}_{\mathsf{IT\text{-}PAC}}$. Specifically, $\mathcal{S}_{\mathsf{PAC}}$ interacts with $\mathcal{A}$ as follows:

(1) $\mathcal{S}_{\mathsf{PAC}}$ samples a random $\mathsf{seed} \in \{0,1\}^\lambda$, and then emulates the (Commit) command of $\mathcal{F}_{\mathsf{Com}}$ by sending a handle τ_1 to $\mathcal{A}$.

(2) Following the protocol specification, $\mathcal{S}_{\mathsf{PAC}}$ samples $\Lambda \leftarrow \mathbb{F}$, and simulates the encryption of polynomial key Λ honestly where a secret key sk is derived from seed. Then, $\mathcal{S}_{\mathsf{PAC}}$ sends the ciphertexts $\langle \Lambda \rangle, \ldots, \langle \Lambda^{(2B-2)} \rangle$ to $\mathcal{A}$.

(3) $\mathcal{S}_{\mathsf{PAC}}$ emulates $\mathcal{F}_{\mathsf{VOLE}}$ by sampling uniform $\Delta \in \mathbb{F}$ and recording the vectors $\boldsymbol{u}, \boldsymbol{w} \in \mathbb{F}^\ell$ sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{VOLE}}$, where $\ell = 2n + m + 2$ when applying sub-protocol $\Pi_{\mathsf{IT\text{-}PAC}}^{(2B-2)}$ into protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$. Then, $\mathcal{S}_{\mathsf{PAC}}$ computes $\boldsymbol{v} := \boldsymbol{w} - \boldsymbol{u} \cdot \Delta$.

(4) By emulating the (Commit) command of $\mathcal{F}_{\mathsf{Com}}$, $\mathcal{S}_{\mathsf{PAC}}$ receives the ciphertexts $\langle b_1 \rangle, \ldots, \langle b_\ell \rangle$ from $\mathcal{A}$, and sends a handle τ_2 to $\mathcal{A}$. For each $j \in [1, \ell]$, $\mathcal{S}_{\mathsf{PAC}}$ computes b_j by decrypting the ciphertext $\langle b_j \rangle$ with secret key sk, and then computes $\mathsf{K}_j := v_j - b_j \cdot \Delta$ following the protocol description.

(5) $\mathcal{S}_{\mathsf{PAC}}$ emulates the (Open) command of functionality $\mathcal{F}_{\mathsf{Com}}$ by sending (seed, τ_1) to $\mathcal{A}$, and also sends Λ to $\mathcal{A}$. In addition, $\mathcal{S}_{\mathsf{PAC}}$ emulates the (Open) command of $\mathcal{F}_{\mathsf{Com}}$ by receiving τ_2 from adversary $\mathcal{A}$.

By invoking $\mathcal{S}_{\mathsf{PAC}}$ and $\mathcal{S}_{\mathsf{DVZK}}$, the simulator $\mathcal{S}_{\mathsf{ZK}}$ simulates the view of adversary $\mathcal{A}$ for the protocol execution of $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ as follows:

(1) $\mathcal{S}_{\mathsf{ZK}}$ emulates $\mathcal{F}_{\mathsf{VOLE}}$ by invoking $\mathcal{S}_{\mathsf{PAC}}$ to generate a uniform global key $\Delta \in \mathbb{F}$.

(2) $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{PAC}}$ to generate a uniform polynomial key $\Lambda \in \mathbb{F}$.

(3) $\mathcal{S}_{\mathsf{ZK}}$ emulates functionality $\mathcal{F}_{\mathsf{VOLE}}$ by recording the values and MACs on $\{[a_{i,j}]\}_{i \in [1,B], j \in [1,m]}$ received from $\mathcal{A}$, and then computes the corresponding local keys in the natural way.

(4) After receiving $b_{i,j} \in \mathbb{F}$ for $i \in [1, B], j \in [1, m]$ from $\mathcal{A}$, $\mathcal{S}_{\mathsf{ZK}}$ computes $w_{i,j} := a_{i,j}+b_{i,j} \in \mathbb{F}$ for each $i \in [1, B], j \in [1, m]$, and then defines $\boldsymbol{w}_i = (w_{i,1}, \ldots, w_{i,m}) \in \mathbb{F}^m$ as a witness for each $i \in [1, m]$.

(5) Simulator $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{PAC}}$ to simulate the execution of sub-protocol $\Pi^{(2B-2)}_{\mathsf{IT\text{-}PAC}}$ for generating the IT-PACs $\{[u_j(\cdot)]\}_{j\in[1,m]}$, $\{([h_j(\cdot)], [\widetilde{h}_j(\cdot)])\}_{j\in[1,n]}$ and $([r(\cdot)], [s(\cdot)])$. Following the topological order, $\mathcal{S}_{\mathsf{ZK}}$ computes the local key on the output wire of every addition gate. Now, $\mathcal{S}_{\mathsf{ZK}}$ holds the local keys of the IT-PAC on every wire and the local keys of the IT-MAC on every circuit-input wire.

(6) Following the protocol specification, the simulator $\mathcal{S}_{\mathsf{ZK}}$ executes the linear-combination phase of the $\mathsf{BatchCheck}_{(B-1),(2B-2),B}$ procedure with $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{PAC}}$ to open the key Λ to $\mathcal{A}$. Next, $\mathcal{S}_{\mathsf{ZK}}$ uses the Δ and local keys to execute the check phase of $\mathsf{BatchCheck}_{(B-1),(2B-2),B}$ with $\mathcal{A}$.

(7) $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{DVZK}}$ to perform the verification of the VOLE-based ZK proof DVZK using the global key Δ and local keys.

(8) Following the protocol description, $\mathcal{S}_{\mathsf{ZK}}$ executes the $\mathsf{CheckZero}$ procedure with $\mathcal{A}$ for the consistency check of input gates and output gates.

(9) If $\mathcal{S}_{\mathsf{ZK}}$ who acts as an honest verifier will abort in some check step, $\mathcal{S}_{\mathsf{ZK}}$ sends $\boldsymbol{w}_i = \perp$ for $i \in [1, B]$ along with a circuit $\mathcal{C}$ to $\mathcal{F}^B_{\mathsf{ZK}}$ and then aborts. Otherwise, $\mathcal{S}_{\mathsf{ZK}}$ sends the extracted witnesses $(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_B)$ and $\mathcal{C}$ to $\mathcal{F}^B_{\mathsf{ZK}}$.

It is easy to see that the simulation of $\mathcal{S}_{\mathsf{ZK}}$ is perfect, where recall that the DVZK simulation of $\mathcal{S}_{\mathsf{DVZK}}$ invoked by $\mathcal{S}_{\mathsf{ZK}}$ is perfect. Whenever the honest verifier outputs false in the real-world execution, the verifier also outputs false in the ideal-world execution, as $\mathcal{S}_{\mathsf{ZK}}$ sends $\perp$ to $\mathcal{F}^B_{\mathsf{ZK}}$ in this case. Therefore, it only remains to bound the probability that the

verifier in the real-world execution outputs true, but there exists some $i \in [1, B]$ such that $\mathcal{C}(\boldsymbol{w}_i) \neq 0$ where $\boldsymbol{w}_i$ is extracted by $\mathcal{S}_{\mathsf{ZK}}$. Below, we show if $\mathcal{C}(\boldsymbol{w}_i) \neq 0$ for some $i \in [1, B]$, then the probability that the verifier outputs true in the real protocol execution is at most $\frac{2B+3}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

By induction, we prove that all addition and multiplication gates in all B executions of circuit $\mathcal{C}$ are evaluated correctly. It is trivial that all addition gates in the $(B, |\mathcal{C}|)$-SIMD circuit are computed correctly from the additively homomorphic property of IT-PACs. Thus, we focus on analyzing the correctness of computing multiplication gates. From the soundness of the DVZK proof, we obtain that $\widetilde{h}_j(\Lambda) = f_j(\Lambda) \cdot g_j(\Lambda)$ for all $j \in [1, n]$, except with probability at most $\epsilon_{\mathsf{dvzk}} = \frac{3}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$. Thus, we can replace the IT-PAC $[\widetilde{h}_j(\cdot)]$ with $[f_j(\cdot) \cdot g_j(\cdot)]$ for each $j \in [1, n]$. From Lemma 5 for the soundness of $\mathsf{BatchCheck}_{(B-1),(2B-2),B}$, we have that $h_j(\alpha_i) = f_j(\alpha_i) \cdot g_j(\alpha_i)$ for all $i \in [1, B], j \in [1, n]$ hold for the IT-PACs $\{([h_j(\cdot)], [f_j(\cdot) \cdot g_j(\cdot)])\}_{j \in [1,n]}$, except with probability at most $\frac{2B}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$. Together, for each $j \in [1, n]$, we have that the j-th multiplication gate in all B executions of circuit $\mathcal{C}$ is evaluated correctly, except with probability at most $\frac{2B+3}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

In the following, we prove that the IT-PACs on the input gates and output gates in the $(B, |\mathcal{C}|)$-SIMD circuit are computed in the way consistent to the witnesses and 0 respectively. Specifically, for each $j \in [1, m]$, we define a polynomial $u_j^*(\cdot)$ such that $u_j^*(\alpha_i) = w_{i,j}$ for all $i \in [1, B]$, where $w_{i,j} = a_{i,j} + b_{i,j}$ for $i \in [1, B], j \in [1, m]$. From the definition of $[z_j] = \sum_{i \in [1,B]} \delta_i(\Lambda) \cdot [w_{i,j}]$, we have that $[z_j] = [u_j^*(\Lambda)]$. The probability that there exists some $j \in [1, m]$ such that $u_j(\Lambda) \neq u_j^*(\Lambda)$ but the verifier accepts in the $\mathsf{CheckZero}([u_j(\Lambda)] - [u_j^*(\Lambda)])$ procedure is at most $\frac{1}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$, where $[u_j(\cdot)]$ is generated

by running sub-protocol $\Pi_{\mathsf{IT\text{-}PAC}}^{(2B-2)}$. Let E_4 be the event that there exists some $j \in [1, m]$ such that $u_j(\Lambda) = u_j^*(\Lambda)$ but $u_j(\cdot) \neq u_j^*(\cdot)$. According to the proof of Lemma 5, we know that $\Pr[E_4] \leq \frac{2B-2}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$. Therefore, the witnesses are consistent to the IT-PACs on the input gates, except with probability at most $\frac{2B-1}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$. Since the verifier accepts in the $\mathsf{CheckZero}([v(\Lambda)])$ procedure, we have that $v(\Lambda) = 0$ except with probability at most $\frac{1}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$, where $[v(\cdot)]$ be the IT-PAC committing to the values on the output gates. Let E_5 be the event that $v(\Lambda) = 0$ but $v(\cdot) \neq 0$. Again, according to the proof of Lemma 5, we obtain that $\Pr[E_5] \leq \frac{2B-2}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

Overall, we conclude that if $\mathcal{C}(\boldsymbol{w}_i) \neq 0$ for some $i \in [1, B]$, then the probability that the verifier outputs true in the real-world execution is bounded by $\frac{2B+3}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$, where the repeated computation of probabilities is merged.

Malicious verifier. As such, we first construct a PPT simulator $\mathcal{S}_{\mathsf{PAC}}$ to simulate the adversary's view in the execution of sub-protocol $\Pi_{\mathsf{IT\text{-}PAC}}^{(2B-2)}$. In particular, $\mathcal{S}_{\mathsf{PAC}}$ interacts with $\mathcal{A}$ as follows:

(1) $\mathcal{S}_{\mathsf{PAC}}$ emulates $\mathcal{F}_{\mathsf{VOLE}}$ by recording $\Delta \in \mathbb{F}$ and a vector $\boldsymbol{v} \in \mathbb{F}^{\ell}$ that are sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{VOLE}}$.

(2) $\mathcal{S}_{\mathsf{PAC}}$ emulates the (Commit) command of $\mathcal{F}_{\mathsf{Com}}$ by receiving a seed from $\mathcal{A}$ and sending a handle τ_1 to $\mathcal{A}$. Then $\mathcal{S}_{\mathsf{PAC}}$ computes the secret key sk and randomness $r_0, r_1, \ldots, r_{(2B-2)}$ with seed following the protocol specification.

(3) On behalf of an honest verifier, $\mathcal{S}_{\mathsf{PAC}}$ receives the AHE ciphertexts $c_1, \ldots, c_{(2B-2)}$ from $\mathcal{A}$.

(4) For $j \in [1, \ell]$, $\mathcal{S}_{\mathsf{PAC}}$ samples $b_j \leftarrow \mathbb{F}$ and encrypts b_j as $\langle b_j \rangle$ using secret key sk. Then, $\mathcal{S}_{\mathsf{PAC}}$ emulates the (Commit) command of $\mathcal{F}_{\mathsf{Com}}$ by sending a handle τ_2 to $\mathcal{A}$.

(5) $\mathcal{S}_{\mathsf{PAC}}$ emulates the (Open) command of functionality $\mathcal{F}_{\mathsf{Com}}$ by receiving a handle τ_1 from $\mathcal{A}$. After receiving $\Lambda \in \mathbb{F}$ from $\mathcal{A}$, $\mathcal{S}_{\mathsf{PAC}}$ verifies the correctness of the ciphertexts sent by $\mathcal{A}$ by checking $c_i = \mathsf{Enc}(\mathsf{sk}, \Lambda^i; r_i)$ for $i \in [1, 2B-2]$. If the check fails, $\mathcal{S}_{\mathsf{PAC}}$ aborts. Otherwise, $\mathcal{S}_{\mathsf{PAC}}$ computes $\mathsf{K}_j := v_j - b_j \cdot \Delta$ for each $j \in [1, \ell]$.

(6) $\mathcal{S}_{\mathsf{PAC}}$ emulates the (Open) command of $\mathcal{F}_{\mathsf{Com}}$ by sending the ciphertexts $\langle b_1 \rangle, \ldots, \langle b_\ell \rangle$ as well as τ_2 to adversary $\mathcal{A}$.

For the $\mathsf{CheckZero}$ procedure, the verifier checks that the hash value sent by the prover is identical to that computed from the local keys. Therefore, the simulator can use the local keys held by the verifier to simulate $\mathsf{CheckZero}$ by computing the hash value from the local keys. If $\mathcal{S}_{\mathsf{ZK}}$ receives false from functionality $\mathcal{F}_{\mathsf{ZK}}^B$, then it simply aborts. Otherwise, by invoking $\mathcal{S}_{\mathsf{PAC}}$ and $\mathcal{S}_{\mathsf{DVZK}}$, $\mathcal{S}_{\mathsf{ZK}}$ interacts with $\mathcal{A}$ as follows:

(1) $\mathcal{S}_{\mathsf{ZK}}$ emulates $\mathcal{F}_{\mathsf{VOLE}}$ by invoking $\mathcal{S}_{\mathsf{PAC}}$ to receive a global key $\Delta \in \mathbb{F}$ from $\mathcal{A}$, and stores Δ.

(2) $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{PAC}}$ to simulate the execution about creating a polynomial key Λ.

(3) $\mathcal{S}_{\mathsf{ZK}}$ emulates $\mathcal{F}_{\mathsf{VOLE}}$ by recording the local keys on IT-MACs $[a_{i,j}]$ for $i \in [1, B], j \in [1, m]$ that are sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{VOLE}}$.

(4) For each $i \in [1, B], j \in [1, m]$, $\mathcal{S}_{\mathsf{ZK}}$ samples $b_{i,j} \leftarrow \mathbb{F}$ and sends it to $\mathcal{A}$, and then computes the local key on the IT-MAC $[w_{i,j}]$.

(5) $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{PAC}}$ to simulate the execution of $\Pi_{\mathsf{IT\text{-}PAC}}^{(2B-2)}$ for generating the IT-PACs $\{[u_j(\cdot)]\}_{j\in[1,m]}$, $\{([h_j(\cdot)],[\widetilde{h}_j(\cdot)])\}_{j\in[1,n]}$ and $([r(\cdot)],[s(\cdot)])$. $\mathcal{S}_{\mathsf{ZK}}$ also computes the local key on the output wire of every addition gate. Now, $\mathcal{S}_{\mathsf{ZK}}$ holds the local keys on all IT-PACs and IT-MACs.

(6) For the linear-combination phase of $\mathsf{BatchCheck}_{(B-1),(2B-2),B}$, $\mathcal{S}_{\mathsf{ZK}}$ receives a seed from $\mathcal{A}$, and then samples two random polynomials $f(\cdot)$ and $g(\cdot)$ in $\mathbb{F}[X]$ such that $f(\alpha_i) = g(\alpha_i)$ for all $i \in [1, B]$, where the degrees of $f(\cdot)$ and $g(\cdot)$ are $B-1$ and $2B-2$ respectively. Then, $\mathcal{S}_{\mathsf{ZK}}$ sends $f(\cdot)$ and $g(\cdot)$ to $\mathcal{A}$.

(7) $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{PAC}}$ to simulate the execution of receiving the key Λ from $\mathcal{A}$. For the check phase of $\mathsf{BatchCheck}_{(B-1),(2B-2),B}$, $\mathcal{S}_{\mathsf{ZK}}$ computes the local keys on IT-MACs $[f(\Lambda)] - f(\Lambda)$ and $[g(\Lambda)] - g(\Lambda)$, and then uses them to execute the $\mathsf{CheckZero}$ procedure with $\mathcal{A}$, where note that $f(\Lambda)$ and $g(\Lambda)$ can be computed by $\mathcal{S}_{\mathsf{ZK}}$.

(8) Using Δ and the local keys, simulator $\mathcal{S}_{\mathsf{ZK}}$ invokes $\mathcal{S}_{\mathsf{DVZK}}$ to simulate the protocol execution of DVZK without knowing the underlying witness.

(9) Following the protocol specification, $\mathcal{S}_{\mathsf{ZK}}$ computes the local keys on IT-MACs $[u_j(\Lambda)] - [z_j]$ for all $j \in [1, m]$, and then uses them to execute the $\mathsf{CheckZero}$ procedure with $\mathcal{A}$.

(10) $\mathcal{S}_{\mathsf{ZK}}$ uses the local key on $[v(\cdot)]$ to run the $\mathsf{CheckZero}([v(\Lambda)])$ procedure with $\mathcal{A}$, where the local key in IT-MAC $[v(\Lambda)]$ is equal to that in IT-PAC $[v(\cdot)]$.

Clearly, the simulation of functionalities $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{Com}}$ is perfect. In the real execution of sub-protocol $\Pi_{\mathsf{IT\text{-}PAC}}^{(2B-2)}$, the ciphertexts $\langle\Lambda\rangle, \ldots, \langle\Lambda^{(2B-2)}\rangle$ sent by $\mathcal{A}$ are guaranteed to be computed correctly by the opening and checking step. In the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model, for each $j \in [1, \ell]$, u_j is uniform in $\mathbb{F}$, and thus $b_j = f(\Lambda) - u_j$ is also uniform in $\mathbb{F}$. For

the j-th ciphertext $\langle b_j \rangle$ with $j \in [1, \ell]$ in the execution of sub-protocol $\Pi^{(2B-2)}_{\mathsf{IT\text{-}PAC}}$, while $\langle b_j \rangle$ is computed as $\langle b_j \rangle = \sum_{i=1}^{k} f_{j,i} \cdot \langle \Lambda^i \rangle + f_{j,0} - u_j$ in the real-world execution, $\langle b_j \rangle$ is computed as $\mathsf{Enc}(\mathsf{sk}, b_j)$ for a random $b_j \in \mathbb{F}$ in the ideal-world execution. For each $j \in [1, \ell]$, the message b_j has the identical distribution in both worlds, and the only difference between the real-world execution and ideal-world execution is the computation method of ciphertext $\langle b_j \rangle$. The difference is easy to be bounded by a reduction to the circuit privacy of the underlying AHE scheme, and thus is negligible in λ.

For the input processing, for each $i \in [1, B], j \in [1, m]$, while $b_{i,j} = w_{i,j} - a_{i,j} \in \mathbb{F}$ in the real-world execution, $b_{i,j} \in \mathbb{F}$ is sampled uniformly at random in the ideal-world execution. Due to the uniformity of $a_{i,j}$ for $i \in [1, B], j \in [1, m]$, the distribution of $\{b_{i,j}\}_{i \in [1,B], j \in [1,m]}$ is identical in both worlds. In the real protocol execution, the random linear combination of polynomials $h_j(\cdot), \widetilde{h}_j(\cdot)$ for $j \in [1, n]$ is masked by two random polynomials $r(\cdot), s(\cdot)$ with $r(\alpha_i) = s(\alpha_i)$ for $i \in [1, B]$. Therefore, the resulting polynomials $f(\cdot) = \sum_{j=1}^{\ell} \chi_j \cdot h_j(\cdot) + r(\cdot)$ and $g(\cdot) = \sum_{j=1}^{\ell} \chi_j \cdot \widetilde{h}_j(\cdot) + s(\cdot)$ are uniform in $\mathbb{F}[X]$ such that $f(\alpha_i) = g(\alpha_i)$ for all $i \in [1, B]$. Therefore, the polynomials $f(\cdot)$ and $g(\cdot)$ simulated by $\mathcal{S}_{\mathsf{ZK}}$ have the identical distribution as that in the real protocol execution. Since $\mathcal{S}_{\mathsf{ZK}}$ computes the local keys in the same way as that done by verifier $\mathcal{V}$, the simulation of $\mathsf{CheckZero}$ has the identical distribution as the $\mathsf{CheckZero}$ execution in the real world. From the zero-knowledge property of DVZK, we have that the simulation of protocol DVZK is indistinguishable from the real protocol execution.

Overall, any PPT environment $\mathcal{Z}$ cannot distinguish between the real-world execution and ideal-world execution, which completes the proof. □

Our protocol is also secure for a *relaxed* degree-restriction property: given ciphertexts $\langle\Lambda\rangle, \ldots, \langle\Lambda^{(2B-2)}\rangle$, it is infeasible to compute any ciphertext of $f(\Lambda)$ such that polynomial f has a degree greater than d for some $d \geq 2B-2$. In this case, the soundness error stated in Theorem 8 is bounded by $\frac{d+5}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

4.2.1.2. Streaming ZK proofs. For a large circuit, our ZK protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ is able to prove it batch-by-batch, i.e., a batch of gates are proved each time, where $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ can process the circuit gate-by-gate. For B executions of a circuit C, the memory complexity of $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ is $O(BM)$ that is independent of circuit size $|C|$, where M is the number of multiplication gates in a batch for each circuit copy.

For every protocol execution of $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$, the polynomial key Λ is opened, and thus the IT-PAC commitments, which are generated after Λ was opened, are not binding any more. Thus, for the next protocol execution, the verifier has to sample a fresh key Λ' by running the sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$. Then, two parties $\mathcal{P}$ and $\mathcal{V}$ need to convert the IT-PACs $[v_1(\cdot)]_\Lambda, \ldots, [v_\ell(\cdot)]_\Lambda$ that commit to the output values in the current batch into the IT-PACs $[v_1(\cdot)]_{\Lambda'}, \ldots, [v_\ell(\cdot)]_{\Lambda'}$ on the same polynomials, which are used as the input IT-PACs for the next batch.

To realize the conversion of IT-PACs $[v_1(\cdot)]_\Lambda, \ldots, [v_\ell(\cdot)]_\Lambda$ from the key Λ to a new key Λ', both parties generate the IT-PACs $[v_1(\cdot)]_{\Lambda'}, \ldots, [v_\ell(\cdot)]_{\Lambda'}$ by running sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ before the polynomial keys Λ and Λ' are opened, and then check the consistency of polynomials between two sets of IT-PACs $\{[v_i(\cdot)]_\Lambda\}_{i\in[1,\ell]}$ and $\{[v_i(\cdot)]_{\Lambda'}\}_{i\in[1,\ell]}$. Specifically, the consistency check of polynomial-key conversion of IT-PACs is performed as follows:

- Linear combination phase: Before both polynomial keys Λ and Λ' are opened, two parties $\mathcal{P}$ and $\mathcal{V}$ do the following:

(1) $\mathcal{P}$ and $\mathcal{V}$ generate two random IT-PACs $[r(\cdot)]_\Lambda$ and $[r(\cdot)]_{\Lambda'}$ under different polynomial keys by running sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$, where $r(\cdot)$ is a random polynomial and has the same degree as $v_i(\cdot)$ for $i \in [1, \ell]$.

(2) $\mathcal{V}$ samples $\mathsf{seed} \leftarrow \{0,1\}^\lambda$ and sends it to $\mathcal{P}$. Then, both parties compute $(\chi_1, \ldots, \chi_\ell) := \mathsf{H}(\mathsf{seed}) \in \mathbb{F}^\ell$.

(3) $\mathcal{P}$ and $\mathcal{V}$ locally compute $[v(\cdot)]_\Lambda := \sum_{i=1}^{\ell} \chi_i \cdot [v_i(\cdot)]_\Lambda + [r(\cdot)]_\Lambda$ and $[v(\cdot)]_{\Lambda'} := \sum_{i=1}^{\ell} \chi_i \cdot [v_i(\cdot)]_{\Lambda'} + [r(\cdot)]_{\Lambda'}$. Then, $\mathcal{P}$ sends the polynomial $v(\cdot) = \sum_{i=1}^{\ell} \chi_i \cdot v_i(\cdot) + r(\cdot)$ to $\mathcal{V}$.

- Check phase:

(4) $\mathcal{P}$ and $\mathcal{V}$ locally compute $[\epsilon] := [v(\Lambda)] - v(\Lambda)$ and $[\sigma] := [v(\Lambda')] - v(\Lambda')$. Then, both parties run $\mathsf{CheckZero}([\epsilon], [\sigma])$ to check that $\epsilon = 0$ and $\sigma = 0$. If the check fails, $\mathcal{V}$ aborts.

$\mathcal{V}$ is able to execute the above check phase after both keys Λ and Λ' were opened in which the local keys are obtained by $\mathcal{V}$. The linear combination of $v_1(\cdot), \ldots, v_\ell(\cdot)$ is masked by a random polynomial $r(\cdot)$, which assures the zero-knowledge property. Following the proof of Lemma 4, the soundness error is at most $\frac{2B}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$, as all IT-PACs are generated before the public coefficients $\chi_1, \ldots, \chi_\ell$ are determined and both polynomial keys are opened, and *a single* polynomial $v(\cdot)$ is opened for guaranteeing consistency.

4.2.1.3. Integrating AntMan with VOLE-based ZK proofs. While the recent VOLE-based ZK proofs [96, 46, 9, 100, 97, 6, 44] have a communication *linear* to the circuit size for proving a single generic circuit, our ZK protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ shown in Figure 4.3 and 4.4 achieves the *sublinear* communication for proving SIMD circuits. We

can seamlessly integrate $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ into a VOLE-based ZK protocol to obtain better communication efficiency for proving a single generic circuit. In particular, $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ is used to prove the sub-circuits that have the SIMD structure, and the VOLE-based ZK protocol is used to prove the remaining parts of the circuit. While protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ evaluates the circuit using IT-PACs, the VOLE-based ZK protocol adopts IT-MACs to perform the circuit evaluation. Therefore, we need to give efficient procedures that allow to convert between IT-PACs and IT-MACs. Protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ shown in Figure 4.3 and 4.4 has implied the conversion procedure from IT-MACs to IT-PACs. We only need to present the conversion procedure from IT-PACs to IT-MACs, which is totally similar. Specifically, let $[v_1(\cdot)], \ldots, [v_\ell(\cdot)]$ be the IT-PACs generated by protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$, and $y_{j,i} = v_j(\alpha_i)$ for $i \in [1, B], j \in [1, \ell]$ be the output values that need to be committed by IT-MACs. For each $i \in [1, B], j \in [1, \ell]$, $\mathcal{P}$ and $\mathcal{V}$ can generate an IT-MAC $[y_{j,i}]$ before the polynomial key Λ is opened. In particular, for $i \in [1, B], j \in [1, \ell]$, the parties call functionality $\mathcal{F}_{\mathsf{VOLE}}$ to generate a random IT-MAC $[r_{j,i}]$, and then $\mathcal{P}$ sends $d_{j,i} = y_{j,i} - r_{j,i}$ to $\mathcal{V}$ and both parties compute $[y_{j,i}] := [r_{j,i}] + d_{j,i}$. After Λ was opened, $\mathcal{P}$ and $\mathcal{V}$ execute the verification procedure described in Figure 4.3 and 4.4 to check the consistency between IT-PAC $[v_j(\cdot)]$ and IT-MACs $([y_{j,1}], \ldots, [y_{j,B}])$ for $j \in [1, \ell]$. That is, for each $j \in [1, \ell]$, $\mathcal{P}$ and $\mathcal{V}$ locally compute an IT-MAC $[y_j] := \sum_{i \in [1,B]} \delta_i(\Lambda) \cdot [y_{j,i}]$, and then run $\mathsf{CheckZero}([v_j(\Lambda)] - [y_j])$ to check $v_j(\Lambda) = y_j$.

4.2.2. Sublinear ZK Proof for Generic Circuits

Below, we show how to extend the ZK protocol on SIMD circuits (as shown in Figure 4.3 and 4.4) into a ZK protocol that proves satisfiability of a single generic circuit with *sublinear* communication. To do this, we first compile a generic circuit $\mathcal{C}$ into another equivalent circuit $\mathcal{C}'$ with $|\mathcal{C}'| = |\mathcal{C}| + O(B)$, where B is the number of wire values committed by a single IT-PAC. The new circuit $\mathcal{C}'$ needs to satisfy the following properties:

(1) For each input $\boldsymbol{w}$, $\mathcal{C}(\boldsymbol{w}) = \mathcal{C}'(\boldsymbol{w})$.

(2) The number of input gates, addition gates and multiplication gates is the multiple of B. There are at least B output gates.

(3) Every B same-type gates are divided into a group, where the B related wire values in a group will be committed by an IT-PAC.

This is easy to be realized by adding "dummy" wires and gates following the approach [**102**], where the values on the "dummy" wires are set as 0.

Now we need to deal with the case that the input wires of B gates in a group are *not* corresponding to the output wires of B gates in any group of previous layers. In this case, the values on these input wires have *not* been committed by existing IT-PACs. Thus, we let the prover and verifier generate such an IT-PAC by running sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$. However, for a malicious prover, the values committed by the IT-PAC may be *inconsistent* with the values on the output wires of B gates from different groups in previous layers. Therefore, we give an efficient consistency check to detect such malicious behavior based on the BatchCheck procedure shown in Figure 4.1. Specifically, for each input IT-PAC $[\hat{g}(\cdot)]$, if the j-th wire value $\hat{g}(\alpha_j)$ comes from the i-th wire value $\hat{f}(\alpha_i)$ committed by an output IT-PAC $[\hat{f}(\cdot)]$, we need to check $\hat{f}(\alpha_i) = \hat{g}(\alpha_j)$. This corresponds to the wire that

carries the value $\hat{g}(\alpha_j) = \hat{f}(\alpha_i)$ in the circuit. Following the previous work [61, 102], we refer to a tuple $([\hat{f}(\cdot)], [\hat{g}(\cdot)], i, j)$ as a wire tuple.

In detail, the sublinear ZK protocol $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$ for proving a single generic circuit can be constructed by extending the protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$ on SIMD circuits (as described in Figure 4.3 and 4.4) in the following way.

- **Preprocess circuit.** $\mathcal{P}$ and $\mathcal{V}$ preprocess the input circuit $\mathcal{C}$ and obtain an equivalent SIMD-friendly circuit $\mathcal{C}'$ with the same output. Let $\boldsymbol{w} \in \mathbb{F}^{mB}$ be the input of circuit $\mathcal{C}'$ that consists of an actual witness and "dummy" zero values.
- **Circuit evaluation.** The input preprocessing is the same as that of protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$, except for the following differences:
  - $\boldsymbol{w}$ is written as $(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_B)$ where $w_{1,j}, \ldots, w_{B,j}$ for each $j \in [1, m]$ are packed in a group and will be committed by an IT-PAC. Meanwhile, $\boldsymbol{w}$ will also be committed by mB IT-MACs. Note that $(w_{1,j}, \ldots, w_{B,j})$ for $j \in [1, m]$ corresponds to B input gates in the j-th group.
  - If $w_{i,j}$ for $i \in [1, B], j \in [1, m]$ corresponds to a "dummy" zero value, the IT-MAC $[w_{i,j}] = [0]$ can be generated locally by $\mathcal{P}$ and $\mathcal{V}$. If $(w_{1,j}, \ldots, w_{B,j})$ for $j \in [1, m]$ corresponds to B "dummy" zero values, the IT-PAC $[u_j(\cdot)]$ with $u_j(\alpha_i) = w_{i,j} = 0$ for $i \in [1, B]$ can also be generated locally by both parties.

  Following a predetermined topological order, $\mathcal{P}$ and $\mathcal{V}$ can evaluate the addition and multiplication gates as in protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$, except for the following differences:

– If the input wires of B gates in a group are not corresponding to the output wires of B gates in any group of previous layers, then $\mathcal{P}$ computes a degree-$(B-1)$ polynomial $\hat{g}(\cdot)$ such that $\hat{g}(\alpha_i) = z_i$ for all $i \in [1, B]$, where $\{z_i\}_{i\in[1,B]}$ are the values on these input wires.

– Then, $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ shown in Figure 4.2 to generate an IT-PAC $[\hat{g}(\cdot)]$.

- **Consistency check of wire tuples.** Let $L(i, j)$ be a set of all wire tuples $\{([\hat{f}_h(\cdot)], [\hat{g}_h(\cdot)], i, j)\}$ such that $\hat{f}_h(\alpha_i) = \hat{g}_h(\alpha_j)$. For each $i, j \in [1, B]$, $\mathcal{P}$ and $\mathcal{V}$ check the consistency of the wire tuples in $L(i, j)$ as follows:

  (1) Let ℓ be the size of $L(i, j)$, and $([\hat{f}_1(\cdot)], [\hat{g}_1(\cdot)], i, j), \ldots, ([\hat{f}_\ell(\cdot)], [\hat{g}_\ell(\cdot)], i, j)$ be the wire tuples in $L(i, j)$.

  (2) The parties $\mathcal{P}$ and $\mathcal{V}$ execute the linear-combination phase of the $\mathsf{BatchCheck}_{(B-1),(B-1),1}$ procedure (as shown in Figure 4.1) on inputs $\{[\hat{f}_1(\cdot)], \ldots, [\hat{f}_\ell(\cdot)]\}$ and $\{[\hat{g}_1(\cdot)], \ldots, [\hat{g}_\ell(\cdot)]\}$ to check that $\hat{f}_h(\alpha_i) = \hat{g}_h(\alpha_j)$ for all $h \in [1, \ell]$ before the polynomial key Λ is opened.

  (3) $\mathcal{P}$ and $\mathcal{V}$ execute the check phase of $\mathsf{BatchCheck}_{(B-1),(B-1),1}$ to complete the above check, where $\mathcal{V}$ can perform the check after the key Λ was opened and local keys on the IT-PACs were computed. If the check fails, $\mathcal{V}$ aborts.

A direct optimization for the consistency check as describe above is that $\mathcal{V}$ sends only one random seed to $\mathcal{P}$ for all B^2 executions of $\mathsf{BatchCheck}$, and then both parties can use the seed and a random oracle to generate the public coefficients for all $\mathsf{BatchCheck}$ executions. Note that all B^2 executions of $\mathsf{BatchCheck}$ can be run in parallel. The above consistency check can be executed in parallel with the correctness check of multiplication

gates. Therefore, the ZK protocol $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$ has the same rounds as protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$, and thus is constant-round. The communication complexity of protocol $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$ is $O(|\mathcal{C}|/B + B^3)$, and becomes $O(|\mathcal{C}|^{3/4})$ sublinear to the circuit size if $B = |\mathcal{C}|^{1/4}$. The communication for addition gates is only from the possible IT-PAC generation when the values on the input wires of these gates are not committed by any existing IT-PAC, and thus depends on the circuit structure.

The security proof of protocol $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$ is similar to that of protocol $\Pi_{\mathsf{ZK}}^{\mathsf{SIMD}}$, where the simulation of sub-protocol $\Pi_{\mathsf{PAC}}^{(2B-2)}$ for generating IT-PACs $\{[\hat{g}(\cdot)]\}$ is the same as that of $\Pi_{\mathsf{PAC}}^{(2B-2)}$ for generating IT-PACs $\{([h_j(\cdot)], [\widetilde{h}_j(\cdot)])\}$. Furthermore, the consistency-check procedure for wire tuples is easy to be simulated by invoking the simulator for the $\mathsf{BatchCheck}$ procedure involved in the proof of Theorem 8. Therefore, for the security proof of protocol $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$, we only need to analyze the soundness error of the consistency-check procedure on wire tuples, and the other part of the proof just follows the proof of Theorem 8. In particular, we have the following lemma.

Lemma 6. *For a malicious prover $\mathcal{P}$ and an honest verifier $\mathcal{V}$, if there exists an inconsistent wire tuple, $\mathcal{V}$ aborts in the execution of protocol $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$ except with probability at most $\frac{2B}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.*

Based on Lemma 4 and Lemma 5, the proof of the above lemma is straightforward. Particularly, if there exists some $h \in [1, \ell]$ such that $\hat{f}_h(\alpha_i) \neq \hat{g}_h(\alpha_j)$ for some $i, j \in [1, B]$, the $\mathsf{BatchCheck}$ procedure will abort except with probability $\frac{2B}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$. Combining the proof of Theorem 8 with Lemma 6, we can obtain the following corollary.

Corollary 1. *Protocol* $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$ *UC-realizes functionality* $\mathcal{F}_{\mathsf{ZK}}^{1}$ *on a single generic circuit in the* $(\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{Com}})$*-hybrid model with soundness error at most* $\frac{4B+3}{|\mathbb{F}|} + \mathsf{negl}(\lambda)$.

We are not aware of how to stream our ZK protocol $\Pi_{\mathsf{ZK}}^{\mathsf{generic}}$ in a gate-by-gate way. However, if a circuit of size $|C|$ can be divided into small chunks, each of size M (so that each can fit into the memory), then each chunk can be processed in communication $O(M^{3/4})$, where our protocol ensures the consistency of wire values across chunks. The total communication would be $O(|C|/M^{1/4})$, a memory-communication trade-off.

4.3. Implementation and Benchmarking

4.3.1. Practical Optimizations

Faster polynomial interpolation and multiplication. For every multiplication gate, we have the input polynomials $f(\cdot), g(\cdot) \in \mathbb{F}[X]$ of degree $(B-1)$, and need to compute the polynomial $\widetilde{h}(\cdot) = f(\cdot) \cdot g(\cdot)$ and a degree-B polynomial $h(\cdot)$ such that $h(\alpha_i) = \widetilde{h}(\alpha_i)$ for $i \in [1, B]$. To maximize the performance, we represent every degree-$(B-1)$ polynomial by the polynomial values evaluated at all B-th roots of unity in $\mathbb{F}$. This representation brings a lot of benefits: 1) we can directly use number theoretic transformation (NTT) to switch between the representation of polynomial values and the representation of polynomial coefficients; 2) the set of polynomial values representing $\widetilde{h}(\cdot)$ evaluated at all $(2B-1)$-th roots of unity contains the set of polynomial values representing $h(\cdot)$ evaluated at all B-th roots of unity. In this way, the polynomial-value representation of $h(\cdot)$ can be directly computed by B field multiplications. The computation of polynomial $\widetilde{h}(\cdot)$ only requires additionally applying inverse NTT and NTT to switch between different representations of the polynomials $f(\cdot)$ and $g(\cdot)$, followed by B field multiplications.

Better utilization of plaintext slots in AHE. The AHE schemes based on ring-LWE such as [30, 29, 49] support the efficient computation over multiple independent slots, and allow to manipulate multiple plaintexts at once using SIMD operations. Taking advantage of multiple plaintext slots is the key to obtain high efficiency in the AHE schemes. In our IT-PAC generation protocol with simplified notation, the verifier has a vector $\boldsymbol{a} \in \mathbb{F}^k$ (where $k = 2B - 2$ when applying this protocol into our ZK protocol), and the prover has a matrix $\mathbf{M} \in \mathbb{F}^{\ell \times k}$ and is desired to get a vector $\mathbf{M} \cdot \boldsymbol{a} \in \mathbb{F}^\ell$. Let S be the number of slots (which is 8192 in our implementation). One way to accomplish this task is to have the prover obtaining the ciphertexts on plaintext vectors $\hat{\boldsymbol{a}}_i = (a_i, \ldots, a_i) \in \mathbb{F}^S$ for all $i \in [1, k]$. However, the communication cost in this case is $O(BS)$. To reduce the communication, we adopt the "diagonal method" in prior work [64]. In particular, the verifier fills S slots with copies of $\boldsymbol{a}$ and sends a rotation key to the prover. Then, the prover uses the key to rotate the ciphertexts on vector $\boldsymbol{a}$ and obtains the encryption of a cyclic matrix $\mathbf{A}$ defined by $\boldsymbol{a}$. The rest of the computation remains the same except for homomorphically operating on the ciphertext of matrix $\mathbf{A}$.

4.3.2. Parameters and Testbed Configuration

We implemented our ZK protocol AntMan for proving SIMD circuits. We use two Amazon EC2 `c5.9xlarge` instances located in the same region to act as the prover and verifier. We use 4 threads and throttle the bandwidth to 1 Gbps unless otherwise specified. Our protocol AntMan adopts the BGV homomorphic encryption with a single level [30] to instantiate the AHE scheme, and uses the LPN-based VOLE protocol [96] to realize

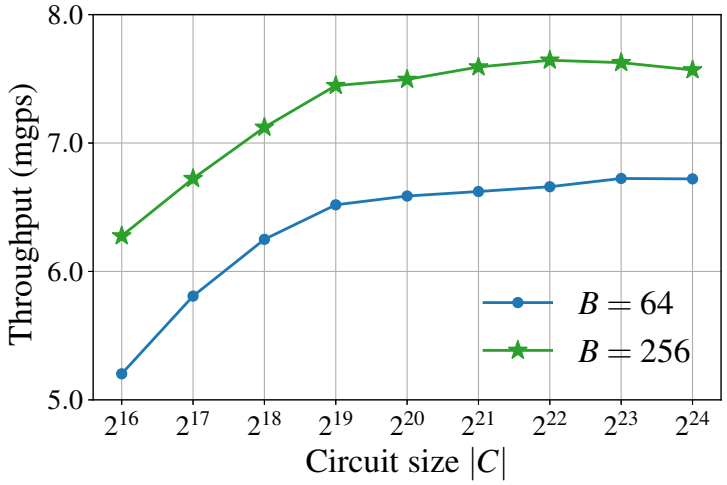

Figure 4.5. **The throughput of our ZK protocol with respect to the circuit size.** The performance is measured as million MULT gates per second (mgps), and is shown for proving $(B, |\mathcal{C}|)$-SIMD circuits where $B \in \{64, 256\}$ and $|\mathcal{C}| \in \{2^{16}, \ldots, 2^{24}\}$.

functionality $\mathcal{F}_{\mathsf{VOLE}}$. In our ZK protocol, addition gates are free and the performance is dominated by evaluating and checking multiplication gates.

There are two implementation versions that depend on the underlying homomorphic encryption libraries: one relies on private library TOTA [105] and the other depends on an open-sourced library SEAL [88]. Compared to SEAL, TOTA supports faster rotation operations, and reduces the size of rotation keys by a factor of about 8×. Thus, TOTA allows AntMan to achieve less setup cost. In the following, we evaluate the performance of our ZK protocol implemented using SEAL. The source code will be available in the EMP-toolkit library [94].

We use a finite field $\mathbb{F}$ defined by a prime $p = 2^{59} - 2^{28} + 1$, which guarantees that $2N$-th roots of unity exist for any $N = 2^k$ and $k < 28$. We set the computational security parameter $\lambda = 128$ and statistical security parameter $\rho = \log|\mathbb{F}| - \log(2B + 3) > 40$, where B is the number of executions of a circuit and chosen from 16 to 2048. We often

refer to B as a batch size as well. In our implementation, we consider a setup phase to generate the AHE ciphertexts on the powers of a polynomial key Λ and rotate these ciphertexts as described in Section 4.3.1. The setup cost is independent of the circuit size and only depends on the parameter B. Our experimental results involve the running time to execute the setup phase. For performance evaluation, we compare AntMan with the state-of-the-art VOLE-based ZK implementation QuickSilver [100].

4.3.3. Performance for Circuit Size and Batch Size

The performance of our ZK protocol AntMan does not depend on the circuit structure, and is only related to the circuit size $|\mathcal{C}|$ and the batch size B. Therefore, we evaluate the performance using random circuits.

Performance vs. circuit size. In Figure 4.5, we analyze how the throughput of AntMan changes over different circuit sizes. We study the performance on proving $(B, |\mathcal{C}|)$-SIMD circuits with $B \in \{64, 256\}$ and that $|\mathcal{C}|$ is chosen from 2^{16} to 2^{24}. From this figure, we observe that the overall performance of our ZK protocol increases as the circuit size becomes larger. This is because the setup cost of $O(B)$ is amortized over the circuit, and thus a larger circuit leads to a smaller amortized cost. The highest throughput is reached when the circuit size is larger than 2^{20}, after that the throughput is stable. Indeed, the throughput in this case is roughly the throughput without counting the setup cost.

Performance vs. batch size. Now we fix the circuit size $|\mathcal{C}|$ to be 2^{21} and study how different batch sizes (i.e., B) impacts the performance of our ZK protocol. In Table 4.1, we report the running time and communication cost of AntMan with batch sizes B changed from 16 to 2048.

B	Running time		Communication
	Setup (ms)	Per gate (μs)	Per gate (field elements)
16	449	0.259	0.78
32	472	0.186	0.39
64	525	0.151	0.19
128	645	0.139	0.098
256	936	0.134	0.049
512	1877	0.143	0.0242
1024	5480	0.146	0.0122
2048	18505	0.152	0.0061
QuickSilver	0	0.169	1

Table 4.1. **The communication and running time of our ZK protocol.** The running time is benchmarked with 4 threads and 1 Gbps bandwidth. The circuit size $|\mathcal{C}| = 2^{21}$ for the whole table. The setup communication cost for all B in the table is 40 MB.

Because our AHE parameters support up to 8192 slots, the communication cost for the setup phase is 40 MB for all choices of B. This mainly consists of rotation keys of which the size is 35.5 MB. When B is greater than 2048, more ciphertexts need to be sent but it does not impact the overall setup communication by much. The setup computation cost is mostly brought from rotation operations.

As for the performance without the setup cost, we observe that we are able to achieve better than one-field-element per multiplication gate once $B \geq 16$. The amortized running time per multiplication gate of AntMan decreases as the number of circuit executions increases from 16 to 256, and deteriorates when batch size is greater than 256. [2] Meanwhile, it is faster than QuickSilver under the 1 Gbps network bandwidth when $B \geq 64$. In terms of communication cost, the communication per multiplication gate for AntMan

[2]The running time of polynomial multiplication increases fast as batch size B increases. When $B > 256$, this leads to the decrease of throughput of AntMan.

is reduced by half every time B doubles. When $B = 2048$, the communication cost per multiplication gate of our ZK protocol is about 0.0061 field elements, which improves the state-of-the-art ZK implementation QuickSilver by a factor of more than 164×. This means that when the network bandwidth is low, our ZK protocol is significantly better.

Memory comparison. Our ZK protocol will require more memory cost than QuickSilver. For example, for circuit size $|C| = 2^{21}$, AntMan needs about 0.88 GB, 1.97 GB and 6.13 GB memory for $B = 64$, $B = 256$ and $B = 1024$ respectively, while QuickSilver only needs about 400 MB memory for any B.

4.3.4. Performance for Threads and Bandwidthes

In Table 4.2, we show how multi-threading and different bandwidthes affect the performance of the ZK protocol AntMan. In particular, we fix the batch size $B = 1024$ and the circuit size $|\mathcal{C}| = 2^{21}$. Our ZK protocol is computationally heavy, and thus the multi-threading boosts the efficiency significantly. In a high bandwidth setting (1 Gbps), the throughput of AntMan increases from 2.04 to 17.85 mgps when the number of threads increases from 1 to 16. On the other hand, AntMan is highly communication-efficient, and thus performs well in low bandwidth settings. The throughput of AntMan does not significantly benefit from the increase of network bandwidthes when it is higher than 50 Mbps. It is due to the fact that the communication cost is reduced by a factor of B asymptotically, compared to the ZK protocol QuickSilver [100].

Protocol-thread	Network Bandwidth				
	10 Mbps	50 Mbps	100 Mbps	500 Mbps	1 Gbps
AntMan-1	1.76	2.00	2.00	2.01	2.04
AntMan-2	3.04	3.70	3.74	3.83	3.84
AntMan-4	4.57	6.27	6.54	6.74	6.84
AntMan-8	6.33	10.04	10.85	11.57	11.71
AntMan-16	7.78	14.31	15.98	17.57	17.85
QuickSilver-∞	0.17	0.85	1.7	8.47	16.95

Table 4.2. **The performance of our ZK protocol subject to the bandwidth and the number of threads.** The benchmark results are the number of million MULT gates per second (mgps). QuickSilver-∞ refers to the theoretical performance of QuickSilver with infinity computational power and thus the running time is solely determined by the communication.

Operations	Running Time (ns)
BGV homomorphic evaluation	106.46
BGV decryption	3.17
Polynomial multiplication	9.92
Hashing (SHA-256)	0.84
Check of MULT gates	3.69
Others	10.65
Total	134.73

Table 4.3. **The microbenchmark of our ZK protocol.** The running time is the amortized time for proving one multiplication gate. The communication time is involved in the "Others" part.

4.3.5. Microbenchmarking

To understand the slowest part of our ZK protocol, we conduct a microbenchmark of the protocol execution in Table 4.3. In this experiment, we synthesize a random circuit with $|\mathcal{C}| = 2^{22}$ multiplication gates, and prove $B = 256$ executions of a circuit (and thus 2^{30} multiplication gates in total). In this table, 5 major expensive operations are counted. The computation cost mainly comes from the homomorphic operations of AHE

and NTT operations. The homomorphic evaluation of BGV ciphertexts dominates the whole computation cost, and takes around 79% of the total running time. The polynomial multiplication (involving the operations of NTT and inverse NTT) takes about 7.4% of the total running time. Note that the BGV encryption and rotation operations are performed only once at the setup phase, and the setup cost can be amortized to negligible when proving a large circuit.

CHAPTER 5

Efficient Conversions for Zero-Knowledge Proofs

In previous chapters, we discussed ZKP schemes that respectively work for Boolean or arithmetic circuits. However, some statements can be better compiled into the composition of these two. For example, the circuit for the neural network inference can be split into linear and non-linear layers. The former is better represented as arithmetic circuits and the latter can be more efficiently translated to Boolean circuits. In this chapter, we study the conversion in VOLE-based ZKPs that allows to compose different components of a statement with different representations. We start with the conversion between Boolean and arithmetic circuits, and then discuss the conversion between the public commitments and private authenticated values. In the end, we demonstrate their power by prototype experiments that prove the correct inference of a ResNet-101 neural network model.

5.1. Arithmetic-Boolean Conversion for Zero-Knowledge Proofs

In this section, we provide full details on how to construct *ZK-friendly extended doubly authenticated bits* (zk-edaBits) efficiently, and then show how to use them to securely realize conversions between arithmetic and Boolean circuits. Our construction is based on the previously described VOLE-ZK. In our implementation, we make use of the most efficient generic VOLE-ZK protocol Quicksilver [100]. We use a macro shown in Figure 5.1 to describe the commitment procedure.

Macro $\mathsf{Auth}(x, q)$

On input $x \in \mathbb{F}_q$ and $q \in \{2, p\}$, this subroutine interacts with two parties $\mathcal{P}$ and $\mathcal{V}$, and generates an authenticated value $[x]$ for the parties. Let $k = \lambda$ and $\Phi = \Delta$ if $q = 2$. Let $k \in \mathbb{N}$ such that $q^k \geq 2^\rho$ and $\Phi = \Gamma$ if $q = p$.

(1) If $\mathcal{V}$ is honest, sample $\mathsf{K} \leftarrow \mathbb{F}_{q^k}$. Otherwise, receive $\mathsf{K} \in \mathbb{F}_{q^k}$ from the adversary.
(2) If $\mathcal{P}$ is honest, compute $\mathsf{M} := \mathsf{K} + \Phi \cdot x \in \mathbb{F}_{q^k}$. Otherwise, receive $\mathsf{M} \in \mathbb{F}_{q^k}$ from the adversary and recompute $\mathsf{K} := \mathsf{M} - \Phi \cdot x \in \mathbb{F}_{q^k}$.
(3) Output $[x]$ to the parties, i.e., send (x, M) to $\mathcal{P}$ and K to $\mathcal{V}$.

Figure 5.1. **Macro used by functionalities $\mathcal{F}_{\mathsf{authZK}}$ and $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ to generate authenticated values.**

Functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$

This functionality is parameterized by a prime $p > 2$ and an integer $k \geq 1$ with $p^k \geq 2^\rho$. Let $m = \lceil \log p \rceil$.

Initialize: On input (init) from $\mathcal{P}$ and $\mathcal{V}$, sample $\Delta \leftarrow \mathbb{F}_{2^\lambda}$ and $\Gamma \leftarrow \mathbb{F}_{p^r}$ if $\mathcal{V}$ is honest, and receive $\Delta \in \mathbb{F}_{2^\lambda}$ and $\Gamma \in \mathbb{F}_{p^r}$ from the adversary otherwise. Store two global keys (Δ, Γ) and send them to $\mathcal{V}$, and ignore all subsequent (init) commands.

Generate random ZK-friendly edaBits: On input $(\mathsf{random}, \mathsf{id}, \mathsf{id}_0, \ldots, \mathsf{id}_{m-1})$ from two parties $\mathcal{P}$ and $\mathcal{V}$ where $\mathsf{id}, \mathsf{id}_0, \ldots, \mathsf{id}_{m-1}$ are fresh identifiers, generate a random $\mathsf{zk\text{-}edaBit}$ $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ with $r_i \in \mathbb{F}_2$ for $i \in [0, m)$ and $r = \sum_{i=0}^{m-1} r_i \cdot 2^i \in \mathbb{F}_p$ as follows:

(1) If $\mathcal{P}$ is honest, sample $r \leftarrow \mathbb{F}_p$. Otherwise, receive $r \in \mathbb{F}_p$ from the adversary. Decompose r to $(r_0, \ldots, r_{m-1})$ such that $r = \sum_{i=0}^{m-1} r_i \cdot 2^i \mod p$.
(2) Execute $[r_i]_2 \leftarrow \mathsf{Auth}(r_i, 2)$ for $i \in [0, m)$ and $[r]_p \leftarrow \mathsf{Auth}(r, p)$, where the macro $\mathsf{Auth}(\cdot)$ is described in Figure 5.1. Thus, the two parties obtain $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$.
(3) Store $(\mathsf{id}, p, [r]_p)$ and $(\mathsf{id}_i, 2, [r_i]_2)$ for $i \in [0, m)$.

Figure 5.2. **Functionality for ZK-friendly extended doubly authenticated bits.**

5.1.1. Extended Doubly Authenticated Bits for Zero-Knowledge Proofs

$\mathsf{zk\text{-}edaBit}$ is a key tool in this work to efficiently perform conversions between arithmetic and Boolean circuits. A $\mathsf{zk\text{-}edaBit}$ consists of a set of m authenticated bits $([r_0]_2, \ldots, [r_{m-1}]_2)$

Protocol $\Pi_{\mathsf{zk\text{-}edaBits}}$

Parameters: Let $p > 2$ be a prime, $m = \lceil \log p \rceil$ and $k \in \mathbb{N}$ with $p^k \geq 2^\rho$. Two parties want to generate N zk-edaBits. Let B, c be some parameters to be specified later and $\ell = NB + c$.

Initialize: $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{authZK}}$, which returns two uniform global keys to $\mathcal{V}$.

Generating random zk-edaBits:

(1) The parties generate random authenticated values $[r^i]_p$ for $i \in [1, \ell]$. Then, for $i \in [1, \ell]$, $\mathcal{P}$ decomposes r^i to $(r^i_0, \ldots, r^i_{m-1})$ such that $r^i = \sum_{h=0}^{m-1} r^i_h \cdot 2^h \mod p$.

(2) For $i \in [1, \ell]$, $\mathcal{P}$ inputs $(r^i_0, \ldots, r^i_{m-1}) \in \mathbb{F}_2^m$ to $\mathcal{F}_{\mathsf{authZK}}$, which returns $([r^i_0]_2, \ldots, [r^i_{m-1}]_2)$ to the parties.

(3) Place the first N zk-edaBits into N buckets in order, where each bucket has exactly one zk-edaBit. Then, $\mathcal{V}$ samples a random permutation $\pi : [N+1, \ell] \to [N+1, \ell]$ and sends it to $\mathcal{P}$. Use π to permute the remaining $\ell - N$ zk-edaBits.

(4) The parties check that the last c zk-edaBits are correctly computed and abort if not. Divide the remaining $N(B-1)$ (unopened) zk-edaBits into N buckets accordingly, such that each bucket has B zk-edaBits.

(5) For each bucket, both parties choose the first zk-edaBit $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ (that is placed into the bucket in the step 3), and for every other zk-edaBit $([s_0]_2, \ldots, [s_{m-1}]_2, [s]_p)$ in the same bucket, execute the following check:
   (a) Compute $[t]_p := [r]_p + [s]_p$, and then execute $([t_0]_2, \ldots, [t_{m-1}]_2) := \mathsf{AdderModp}([r_0]_2, \ldots, [r_{m-1}]_2, [s_0]_2, \ldots, [s_{m-1}]_2)$ by calling functionality $\mathcal{F}_{\mathsf{authZK}}$, where AdderModp is the modular-addition circuit, and $\sum_{h=0}^{m-1} t_h \cdot 2^h = \sum_{h=0}^{m-1} r_h \cdot 2^h + \sum_{h=0}^{m-1} s_h \cdot 2^h \mod p$.
   (b) Execute the BatchCheck procedure on $([t_0]_2, \ldots, [t_{m-1}]_2)$ to obtain $(t_0, \ldots, t_{m-1})$, and then compute $t' := \sum_{h=0}^{m-1} t_h \cdot 2^h \mod p$.
   (c) Execute the CheckZero procedure on $[t]_p - t'$ to verify that $t = t'$.

(6) If any check fails, $\mathcal{V}$ aborts. Otherwise, the parties output the first zk-edaBit from each of the N buckets.

Figure 5.3. **Protocol for generating ZK-friendly edaBits in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model.**

along with a *random* authenticated value $[r]_p$ such that $r = \sum_{h=0}^{m-1} r_h \cdot 2^h \in \mathbb{F}_p$. We provide the ideal functionality for zk-edaBits in Figure 5.2.

A prover $\mathcal{P}$ and a verifier $\mathcal{V}$ can generate faulty zk-edaBits by calling functionality $\mathcal{F}_{\mathsf{authZK}}$, and then use a "cut-and-bucketing" technique to check the consistency of resulting zk-edaBits. We provide the details of our zk-edaBits protocol in Figure 5.3. In this protocol, the prover and verifier use $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands to compute a Boolean

circuit AdderModp, which efficiently realizes the module-addition computation that adds two m-bit integers and then modules a prime p.

Theorem 10. *Protocol* $\Pi_{\mathsf{zk\text{-}edaBits}}$ *shown in Figure 5.3 UC-realizes functionality* $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ *in the presence of a static, malicious adversary with statistical error at most* $\binom{N(B-1)+c}{B-1}^{-1} + \frac{1}{p^k}$ *in the* $\mathcal{F}_{\mathsf{authZK}}$*-hybrid model.*

Given the number N of zk-edaBits, we can choose suitable parameters B and c such that $\binom{N(B-1)+c}{B-1}^{-1} \leq 2^{-\rho}$. For example, when $N = 10^6$, we can choose $B = 3$ and $c = 2$, and achieve at least 40-bit statistical security.

5.1.1.1. Proof of Security. The security of protocol $\Pi_{\mathsf{zk\text{-}edaBits}}$ shown in Figure 5.3 in the presence of a malicious prover crucially depends on the "cut-and-bucketing" procedure. As in previous work [51, 3, 95], we model this procedure as the "balls-and-buckets" game:

(1) Adversary $\mathcal{A}$ prepares $\ell = NB + c$ balls denoted by $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$. Every ball corresponds to a zk-edaBit, and can be either good or bad depending on whether it is honestly generated by $\mathcal{A}$.

(2) The first N balls $\{\mathcal{B}_1, \ldots, \mathcal{B}_N\}$ are placed into N buckets, where $\mathcal{B}_i$ is assigned into the i-th bucket.

(3) In the set $\{\mathcal{B}_{N+1}, \ldots, \mathcal{B}_\ell\}$, c balls are randomly chosen and opened. If one of the chosen balls is bad, then $\mathcal{A}$ loses the game. Otherwise, the game proceeds to the next step.

(4) The remaining $N(B-1)$ balls are randomly divided into the N buckets, such that each bucket has an equal size B.

(5) We define that a bucket is fully good (resp., fully bad) if all the balls inside it are good (resp., bad). $\mathcal{A}$ wins if and only if there exists at least one bucket that is fully bad, and all other buckets are either fully good or fully bad.

Lemma 7. *Assume $c \geq -1$ $B-1$, then adversary $\mathcal{A}$ wins the above game with probability at most* $\binom{N(B-1)+c}{B-1}$.

Proof. Assume that $\mathcal{A}$ makes m buckets bad for $1 \leq m \leq N$. $\mathcal{A}$ wins if exactly mB balls are bad, and they are placed into the m buckets. The probability of this event is computed below. In the step 2, N balls of ℓ balls are first placed in the N buckets, and thus m bad balls of N balls defines the m bad buckets. Let $B^* = B-1$ and $\ell^* = NB^* + c$. At the step 3 of the game, the probability that none of mB^* bad balls is chosen is

$$p_1 = \frac{\binom{\ell^* - mB^*}{c}}{\binom{\ell^*}{c}} = \frac{(NB^* + c - mB^*)!(NB^*)!}{(NB^* + c)!(NB^* - mB^*)!}.$$

Assuming that this occurs. We are left with $\ell^* - c = NB^*$ balls that have not been placed in the buckets, of which mB^* balls are bad. In the step 4, the probability that mB^* bad balls are exactly put in the m bad buckets is

$$p_2 = \frac{(NB^* - mB^*)!(mB^*)!}{(NB^*)!}.$$

Overall, the probability that adversary $\mathcal{A}$ wins the game is

$$p = p_1 \cdot p_2 = \frac{(NB^* + c - mB^*)!(mB^*)!}{(NB^* + c)!} = \binom{NB^* + c}{mB^*}^{-1}.$$

When $c \geq B - 1 = B^*$ and $1 \leq m \leq N$, the probability p is maximized in the case of $m = 1$. So the probability that $\mathcal{A}$ wins the game is bounded by $\binom{N(B-1)+c}{B-1}^{-1}$. □

Below, we give the formal proof of Theorem 10.

Proof. We first consider the case of a malicious prover, and then consider the case of a malicious verifier. In each case, we construct a PPT simulator $\mathcal{S}$ given access to functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, which runs a PPT adversary $\mathcal{A}$ as a subroutine when emulating $\mathcal{F}_{\mathsf{authZK}}$. In both cases, we show that no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution.

Malicious prover. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) In the process of generating faulty zk-edaBits, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ for $\mathcal{A}$ by sampling uniform "dummy" global keys, and recording all the values $\{r^i\}_{i\in[1,\ell]}$ and $\{(r_0^i, \ldots, r_{m-1}^i)\}_{i\in[1,\ell]}$ and their corresponding MAC tags received from adversary $\mathcal{A}$. Note that these values and MAC tags naturally define corresponding local keys.

(2) Simulator $\mathcal{S}$ plays the role of an honest verifier to perform the consistency-check procedure with $\mathcal{A}$, using the "dummy" global keys and the local keys. In the process of computing modular-addition circuit AdderModp, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ by sending the bits on the output wires of all gates to $\mathcal{A}$, and recording the

corresponding MAC tags sent to $\mathcal{F}_{\mathsf{authZK}}$ by $\mathcal{A}$. Then, $\mathcal{S}$ uses the output bits of circuit AdderModp and their MAC tags to define corresponding local keys.

(3) $\mathcal{S}$ acts as an honest verifier and executes the consistency-check procedure with adversary $\mathcal{A}$, where $\mathcal{S}$ uses the "dummy" global keys and local keys recorded by itself to execute the BatchCheck and CheckZero procedures. If the check fails, $\mathcal{S}$ sends abort to functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ and aborts. Otherwise, $\mathcal{S}$ sends $r^1, \ldots, r^N$ to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, and also sends their corresponding MAC tags and the MAC tags of $\{(r_0^i, \ldots, r_{m-1}^i)\}_{i\in[1,N]}$ to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$.

It is clear that the view of adversary $\mathcal{A}$ is perfectly simulated by $\mathcal{S}$, since the "dummy" global keys sampled by $\mathcal{S}$ have the same distribution as the real global keys, global keys are only used to check validity of opened values, and the local keys are totally determined by the values and MAC tags chosen by $\mathcal{A}$. If the honest verifier aborts in the real protocol execution, then the verifier also aborts in the ideal-world execution (as $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$). Therefore, it remains to bound the probability of the event BadEvent that the honest verifier accepts in the real protocol execution, but there exists one outputting zk-edaBit $(r_0^i, \ldots, r_{m-1}^i, r^i)$ for some $i \in [1, N]$ such that $r^i \neq \sum_{h=0}^{m-1} r_h^i \cdot 2^h \mod p$. In the following, we show that BadEvent occurs with probability at most $\binom{N(B-1)+c}{B-1}^{-1} + \frac{1}{p^k} + \mathsf{negl}(\lambda)$.

In the BatchCheck procedure, the probability which the honest verifier does not abort but there exists some value that is opened incorrectly is bounded by $\mathsf{negl}(\lambda)$. Below, we assume that this does not happen. In the consistency-check procedure, if there are

a correct zk-edaBit and an incorrect zk-edaBit in the same bucket, then the honest verifier would abort unless $\mathcal{A}$ breaks the security of CheckZero with probability at most $1/p^k + \mathsf{negl}(\lambda)$. Now, we assume that there is no mixed zk-edaBits in the same bucket. Therefore, according to Lemma 7, we have the probability that BadEvent occurs is at most $\binom{N(B-1)+c}{B-1}^{-1}$.

If BadEvent doest not occur except with probability at most $\binom{N(B-1)+c}{B-1}^{-1} + \frac{1}{p^k} + \mathsf{negl}(\lambda)$, the output distributions of the honest verifier in the real-world and ideal-world executions are identical, as global keys are uniform, and the local keys are uniquely determined by the values and MAC tags known by $\mathcal{A}$ and the independent global keys.

Malicious verifier. Simulator $\mathcal{S}$ has access to functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, and interacts with $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ by recording global keys Δ, Γ and the local keys for all authenticated values, which are sent to $\mathcal{F}_{\mathsf{authZK}}$ by $\mathcal{A}$. Then $\mathcal{S}$ sends Δ and Γ to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$.

(2) $\mathcal{S}$ receives a permutation π from $\mathcal{A}$, and places the zk-edaBits into N buckets following the protocol specification. In the opening procedure of c zk-edaBits (step 4), for $i \in [1, c]$, $\mathcal{S}$ samples $r^i \leftarrow \mathbb{F}_p$ and decomposes it as $(r^i_0, \ldots, r^i_{m-1})$, and defines corresponding MAC tags using these values as well as global keys Δ, Γ and the related local keys. Then, $\mathcal{S}$ executes the BatchCheck procedure with $\mathcal{A}$ using $\{(r^i_0, \ldots, r^i_{m-1}, r^i)\}_{i\in[1,c]}$ and these MAC tags.

(3) For each bucket, in the checking procedure between the first zk-edaBit and every other zk-edaBit, $\mathcal{S}$ simulates as follows:

   (a) $\mathcal{S}$ samples $t \leftarrow \mathbb{F}_p$ and computes its corresponding MAC tag using Γ and the associated local key defined as above, and also defines $(t_0, \ldots, t_{m-1})$ such that $t = \sum_{i=0}^{m-1} t_i \cdot 2^i \mod p$.

   (b) In the process of executing circuit AdderModp, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ by recording the local keys on t_i for all $i \in [0, m)$, which are computed using the keys sent to $\mathcal{F}_{\mathsf{authZK}}$ by $\mathcal{A}$. Then $\mathcal{S}$ defines the MAC tags on $t_0, \ldots, t_{m-1}$ using Δ and the corresponding local keys.

   (c) Simulator $\mathcal{S}$ uses $t_0, \ldots, t_{m-1}$ along with their corresponding MAC tags to execute the BatchCheck procedure with adversary $\mathcal{A}$.

   (d) $\mathcal{S}$ uses the MAC tag on $[t]_p$ to run the CheckZero procedure with $\mathcal{A}$.

In the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model, the arithmetic values over $\mathbb{F}_p$ on all zk-edaBits are uniformly random from the view of adversary $\mathcal{A}$. Therefore, in the real protocol execution, for each check in the step 5, the value t is uniform in $\mathbb{F}_p$, where r is always masked by a uniform value s. Besides, $\mathcal{S}$ will pass the checks in the BatchCheck and CheckZero procedures, as it always uses the correct MAC tags and the opened bits $t_0, \ldots, t_{m-1}$ satisfy the relation $t = \sum_{i=0}^{m-1} t_i \cdot 2^i \mod p$ for every pairwise check. Overall, the view of adversary $\mathcal{A}$ is perfectly simulated by $\mathcal{S}$. It is clear that the output distribution of the honest prover in the real-world execution is identical to that in the ideal-world execution, since the output values over $\mathbb{F}_2$ or $\mathbb{F}_p$ by the honest prover are uniform under the zk-edaBit condition in both worlds, and the MAC tags output by the honest prover are totally determined by the keys chosen by $\mathcal{A}$ and these output values. This completes the proof. □

Protocol $\Pi^{\mathsf{A2B}}_{\mathsf{Convert}}$

Let $p > 2$ be a prime and $m = \lceil \log p \rceil$.

Initialize: $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, which returns two uniform global keys to $\mathcal{V}$.

Input: The parties have an authenticated value $[x]_p$.

Convert: $\mathcal{P}$ and $\mathcal{V}$ convert an authenticated value over field $\mathbb{F}_p$ into m authenticated bits as follows:

(1) Call funcationality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, which returns $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ to the parties.
(2) Compute $[z]_p := [x]_p - [r]_p$, and then execute the $\mathsf{BatchCheck}$ procedure on $[z]_p$ to obtain z.
(3) Decompose $z \in \mathbb{F}_p$ as $(z_0, \ldots, z_{m-1}) \in \mathbb{F}_2^m$ such that $z = \sum_{h=0}^{m-1} z_h \cdot 2^h \mod p$, and then compute $([x_0]_2, \ldots, [x_{m-1}]_2) := \mathsf{AdderModp}(z_0, \ldots, z_{m-1}, [r_0]_2, \ldots, [r_{m-1}]_2)$ by calling $\mathcal{F}_{\mathsf{authZK}}$ where $z_0, \ldots, z_{m-1}$ are public constants.
(4) Output $([x_0]_2, \ldots, [x_{m-1}]_2)$.

Figure 5.4. **Protocol for conversion from arithmetic to Boolean in the $(\mathcal{F}_{\mathsf{zk\text{-}edaBits}}, \mathcal{F}_{\mathsf{authZK}})$-hybrid model.**

5.1.2. Arithmetic-Boolean Conversion Protocols

Using functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ efficiently realized in the previous sub-section, we propose two efficient protocols to convert authenticated wire values from an arithmetic circuit to a Boolean circuit and to convert in another direction. In the two protocols, the prover and verifier would also use functionality $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands to compute a Boolean circuit $\mathsf{AdderModp}$. In both protocols, we assume that $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ shares the same initialization procedure with $\mathcal{F}_{\mathsf{authZK}}$, and thus the same global keys are used in the two functionalities. This is the case, when we use the protocol $\Pi_{\mathsf{zk\text{-}edaBits}}$ shown in Figure 5.3 to UC-realize $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model. We also assume that $\mathcal{F}_{\mathsf{authZK}}$ can use authenticated values generated by $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$. This is easy to be realized by viewing $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ as a part of $\mathcal{F}_{\mathsf{authZK}}$.

Protocol $\Pi^{\mathsf{B2A}}_{\mathsf{Convert}}$

Let $p > 2$ be a prime and $m = \lceil \log p \rceil$.

Initialize: $\mathcal{P}$ and $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, which returns two uniform global keys to $\mathcal{V}$.

Input: Two parties $\mathcal{P}$ and $\mathcal{V}$ hold m authenticated bits $[x_0]_2, \ldots, [x_{m-1}]_2$.

Convert: $\mathcal{P}$ and $\mathcal{V}$ convert m authenticated bits into one authenticated value over field $\mathbb{F}_p$ as follows:

(1) Call funcationality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$, which returns $([r_0]_2, \ldots, [r_{m-1}]_2, [r]_p)$ to the parties.
(2) Compute $([z_0]_2, \ldots, [z_{m-1}]_2) := \mathsf{AdderMod}p([x_0]_2, \ldots, [x_{m-1}]_2, [r_0]_2, \ldots, [r_{m-1}]_2)$ by calling functionality $\mathcal{F}_{\mathsf{authZK}}$, such that $\sum_{h=0}^{m-1} z_h \cdot 2^h = \sum_{h=0}^{m-1} x_h \cdot 2^h + \sum_{h=0}^{m-1} r_h \cdot 2^h \mod p$.
(3) Execute the BatchCheck procedure on $([z_0]_2, \ldots, [z_{m-1}]_2)$ to obtain $(z_0, \ldots, z_{m-1})$, and then compute $z := \sum_{h=0}^{m-1} z_h \cdot 2^h \mod p$.
(4) Compute and output $[x]_p := z - [r]_p$.

Figure 5.5. **Protocol for conversion from Boolean to arithmetic in the $(\mathcal{F}_{\mathsf{zk\text{-}edaBits}}, \mathcal{F}_{\mathsf{authZK}})$-hybrid model.**

We provide the full details about the conversion from arithmetic to Boolean circuits in Figure 5.4. In Figure 5.5, we describe in details how to perform an efficient conversion from Boolean to arithmetic circuits.

5.1.2.1. Proof of Security. Below, we prove the security of the two protocols in the following theorems.

Theorem 11. *Protocol $\Pi^{\mathsf{A2B}}_{\mathsf{Convert}}$ UC-realizes the* convertA2B *command of functionality $\mathcal{F}_{\mathsf{authZK}}$ in the presence of a static, malicious adversary with statistical error $1/p^k$ in the $(\mathcal{F}_{\mathsf{zk\text{-}edaBits}}, \mathcal{F}_{\mathsf{authZK}})$-hybrid model.*

Proof. We consider two cases of a malicious prover or a malicious verifier. In each case, we construct a PPT simulator $\mathcal{S}$, which runs a PPT adversary $\mathcal{A}$ as a subroutine, when emulating $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands. In both cases, we show that

no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution.

Malicious prover. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ by receiving a field element r from $\mathcal{A}$, writing r to $(r_0, \ldots, r_{m-1})$ with $r = \sum_{i=0}^{m-1} r_i \cdot 2^i \mod p$, and recording their corresponding MAC tags sent to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ by $\mathcal{A}$.

(2) $\mathcal{S}$ executes the $\mathsf{BatchCheck}$ procedure with $\mathcal{A}$. If the values sent by $\mathcal{A}$ are not consistent with $z = x - r \mod p$ and the corresponding MAC tag, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{authZK}}$ and aborts. Note that x has been extracted by $\mathcal{S}$ in the procedure of generating authenticated value $[x]_p$.

(3) $\mathcal{S}$ decomposes z as $(z_0, \ldots, z_{m-1}) \in \{0,1\}^m$. Then $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ with circuit-based commands to execute the evaluation of circuit $\mathsf{AdderModp}$ with $\mathcal{A}$. In particular, $\mathcal{S}$ sends $x_0, \ldots, x_{m-1}$ to $\mathcal{A}$ and records their corresponding MAC tags computed with the MAC tags received from $\mathcal{A}$, where $\sum_{h=0}^{m-1} x_h \cdot 2^h \mod p = x$.

(4) $\mathcal{S}$ sends the MAC tags on bits $x_0, \ldots, x_{m-1}$ to $\mathcal{F}_{\mathsf{authZK}}$.

In the real protocol execution, if the value opened by $\mathcal{A}$ is not equal to $z = x - r \mod p$ in the $\mathsf{BatchCheck}$ procedure, then the honest verifier would abort except with probability at most $1/p^k + \mathsf{negl}(\lambda)$. If the opened value is identical to z, then using the MAC tag to check the correctness of the opened value, is equivalent to using the global and local keys to check, according to the IT-MAC relationship. Thus, the $\mathsf{BatchCheck}$ procedure simulated by $\mathcal{S}$ is indistinguishable from the real checking procedure. In the real-world execution, if $z = x - r \mod p$, then we easily see that adversary $\mathcal{A}$ obtains the bit decomposition

of x (i.e., $x_0, \ldots, x_{m-1}$) from the evaluation of circuit AdderModp. Overall, the view of adversary $\mathcal{A}$ that is simulated by $\mathcal{S}$ is indistinguishable from the view of $\mathcal{A}$ in the real protocol execution.

Malicious verifier. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) In the initialization phase, $\mathcal{S}$ emulates $\mathcal{F}_{\text{zk-edaBits}}$ and receives two global keys from $\mathcal{A}$, and then sends them to functionality $\mathcal{F}_{\text{authZK}}$.

(2) $\mathcal{S}$ emulates $\mathcal{F}_{\text{zk-edaBits}}$ by recording the local keys on $(r_0, \ldots, r_{m-1}, r)$, sent to $\mathcal{F}_{\text{zk-edaBits}}$ by $\mathcal{A}$.

(3) $\mathcal{S}$ runs the BatchCheck procedure with $\mathcal{A}$ by sampling $z \leftarrow \mathbb{F}_p$, and sending it and the corresponding MAC tag to $\mathcal{A}$, where the local keys on $[x]_p$ and $[r]_p$ have been extracted by $\mathcal{S}$, and thus the MAC tag on $[z]_p$ can be computed by $\mathcal{S}$ with the global and local keys.

(4) $\mathcal{S}$ decomposes z as $(z_0, \ldots, z_{m-1}) \in \{0,1\}^m$, and executes the evaluation of circuit AdderModp with $\mathcal{A}$ when emulating $\mathcal{F}_{\text{authZK}}$ with circuit-based commands. During the procedure, $\mathcal{S}$ records the local keys on the output bits of circuit AdderModp, which are computed with the global and local keys received from $\mathcal{A}$. Then, $\mathcal{S}$ sends these local keys to functionality $\mathcal{F}_{\text{authZK}}$.

In the $(\mathcal{F}_{\text{zk-edaBits}}, \mathcal{F}_{\text{authZK}})$-hybrid model, r is uniformly random from the view of adversary $\mathcal{A}$. Therefore, field element z in the real protocol execution is uniform, as it is masked by r. In conclusion, the view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfectly indistinguishable from its view in the real-world execution, which completes the proof. □

Theorem 12. *Protocol* $\Pi^{\mathsf{B2A}}_{\mathsf{Convert}}$ *UC-realizes the* convertB2A *command of functionality* $\mathcal{F}_{\mathsf{authZK}}$ *in the presence of a static, malicious adversary in the* $(\mathcal{F}_{\mathsf{zk\text{-}edaBits}}, \mathcal{F}_{\mathsf{authZK}})$*-hybrid model.*

Proof. As such, we consider two cases of a malicious prover or a malicious verifier. In each case, we construct a PPT simulator $\mathcal{S}$, which runs a PPT adversary $\mathcal{A}$ as a subroutine when emulating functionality $\mathcal{F}_{\mathsf{authZK}}$ with circuit-based commands. In both cases, we show that any PPT environment $\mathcal{Z}$ cannot distinguish the real-world execution from the ideal-world execution.

Malicious prover. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ by receiving a field element r from $\mathcal{A}$, decomposing r to $(r_0, \ldots, r_{m-1})$ with $r = \sum_{i=0}^{m-1} r_i \cdot 2^i \mod p$, and recording their corresponding MAC tags sent to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ by $\mathcal{A}$.

(2) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands, by sending the bits on the output wires of all gates to $\mathcal{A}$, and also receiving their corresponding MAC tags from $\mathcal{A}$. Then $\mathcal{S}$ defines $z_0, \ldots, z_{m-1}$ as the output bits of circuit AdderModp and records their corresponding MAC tags.

(3) $\mathcal{S}$ executes the BatchCheck procedure with $\mathcal{A}$. If the opened bits are not equal to $z_0, \ldots, z_{m-1}$, or the checking value sent by $\mathcal{A}$ does not match with that computed by $\mathcal{S}$ with the recorded MAC tags, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{authZK}}$ and aborts.

(4) $\mathcal{S}$ computes the MAC tag on $[x]_p = z - [r]_p$ locally where $z = \sum_{h=0}^{m-1} z_h \cdot 2^h \mod p$, and sends it to $\mathcal{F}_{\mathsf{authZK}}$.

Clearly, the view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect, except for the $\mathsf{BatchCheck}$ procedure. In the real protocol execution, the honest verifier checks the correctness of bits to be opened using its keys. In the ideal-world execution, $\mathcal{S}$ executes this procedure using the desired bits $z_0, \ldots, z_{m-1}$ and corresponding MAC tags. For $\mathsf{BatchCheck}$, we know that the opened bits are correct (and thus are equal to $z_0, \ldots, z_{m-1}$) if the honest verifier does not abort, except with probability $\mathsf{negl}(\lambda)$. In this case, the checking procedure using the keys is the same as that using the MAC tags, according to the IT-MAC relationship. Therefore, the view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is indistinguishable from the real view of $\mathcal{A}$. From the definitions of functionality $\mathcal{F}_{\mathsf{authZK}}$ and circuit $\mathsf{AdderModp}$, we have that $\sum_{h=0}^{m-1} z_h \cdot 2^h = \sum_{h=0}^{m-1} x_h \cdot 2^h + \sum_{h=0}^{m-1} r_h \cdot 2^h \mod p$ (and thus $z = x + r \mod p$). Then adversary $\mathcal{A}$ will obtain the value $x = \sum_{h=0}^{m-1} x_h \cdot 2^h$ in the real protocol execution. Thus, the output of the honest verifier in the real-world execution is indistinguishable from that in the ideal-world execution, where the verifier's output (i.e., the local key on $x \in \mathbb{F}_p$) is determined by its global key and the value and MAC tag known by $\mathcal{A}$.

Malicious verifier. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) In the initialization phase, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ and receives two global keys from $\mathcal{A}$, and then sends them to functionality $\mathcal{F}_{\mathsf{authZK}}$.

(2) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ by recording the local keys on $(r_0, \ldots, r_{m-1}, r)$, sent to $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ by $\mathcal{A}$.

(3) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands, and records the local keys on the output wires of all gates from adversary $\mathcal{A}$. Then, from these local keys, $\mathcal{S}$ defines and records the local keys on the output bits of circuit $\mathsf{AdderModp}$.

(4) $\mathcal{S}$ samples $z \leftarrow \mathbb{F}_p$ and decomposes it to $(z_0, \ldots, z_{m-1}) \in \{0,1\}^m$, where $\sum_{h=0}^{m-1} z_h \cdot 2^h = z \mod p$. Then, $\mathcal{S}$ executes the $\mathsf{BatchCheck}$ procedure with $\mathcal{A}$ using $z_0, \ldots, z_{m-1}$ and their corresponding MAC tags, where the MAC tags are computed with the global and local keys recorded by $\mathcal{S}$.

(5) $\mathcal{S}$ computes the local key on $[x]_p = z - [r]_p$ locally and sends it to functionality $\mathcal{F}_{\mathsf{authZK}}$.

In the $(\mathcal{F}_{\mathsf{zk\text{-}edaBits}}, \mathcal{F}_{\mathsf{authZK}})$-hybrid model, r is uniform in $\mathbb{F}_p$ and kept secret against adversary $\mathcal{A}$. In the real protocol execution, we have that $(z_0, \ldots, z_{m-1})$ is the bit-decomposition of a uniform element z, based on the uniformity of r. Therefore, the view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfectly indistinguishable from its view in the real-world execution, which completes the proof. □

Optimization using circuit-based zk-edaBits. In the conversion protocols described as above, a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ generate random $\mathsf{zk\text{-}edaBits}$ using functionality $\mathcal{F}_{\mathsf{zk\text{-}edaBits}}$ in the preprocessing phase, and then convert authenticated values between arithmetic and Boolean circuits using these random $\mathsf{zk\text{-}edaBits}$ in the online phase.

We can use an alternative approach to convert authenticated values between arithmetic and Boolean circuits, and obtain better whole efficiency but larger online cost. Specifically, for authenticated bits $[x_0]_2, \ldots, [x_{m-1}]_2$ on m output wires of a Boolean circuit, $\mathcal{P}$ can compute $x := \sum_{h=0}^{m-1} x_h \cdot 2^h \mod p$ locally. Then, $\mathcal{P}$ sends (input, x, p) to $\mathcal{F}_{\mathsf{authZK}}$ and $\mathcal{V}$ sends (input, p) to $\mathcal{F}_{\mathsf{authZK}}$, which returns $[x]_p$ to the parties. Similarly, the parties can also convert an authenticated value $[x]_p$ on an output wire of an arithmetic circuit into m authenticated bits $[x_0]_2, \ldots, [x_{m-1}]_2$, by calling the ($\mathsf{input}$) command of $\mathcal{F}_{\mathsf{authZK}}$. In

this way, two parties can create N circuit-based zk-edaBits for some integer N. However, in the circuit-based zk-edaBits, a malicious prover may cause the field elements over $\mathbb{F}_p$ are inconsistent with corresponding bits. Verifier $\mathcal{V}$ can check the consistency of these circuit-based zk-edaBits using the cut-and-bucketing technique. Specifically, in the online phase, two parties can execute the checking procedure shown in Figure 5.3 to check the consistency of these circuit-based zk-edaBits by sacrificing $(B-1)N+c$ random zk-edaBits generated in the preprocessing phase. Using this optimization, for computing N circuit-based zk-edaBits, we can save N random zk-edaBits and N evaluations of circuit AdderModp in terms of the whole efficiency, but increase the online cost by a factor of $B-1$.

5.2. Converting Publicly Committed Values to Privately Authenticated Values

The second type of conversions that we would like to study is the conversion from publicly committed data to privately authenticated data. Here, publicly committed data referred to those committed with a short digest, which can be published on something that can be modeled as a bulletin board (e.g., well-established websites or some blockchain). Privately authenticated data refers to the values only known by a prover that are authenticated by a designated verifier based on IT-MACs, and thus can be efficiently used to prove any mixed arithmetic-Boolean circuit using the recent ZK protocols [95, 45, 8, 101] and our arithmetic-Boolean conversion protocols. The conversion from publicly committed data to privately authenticated data will allow us to efficiently prove statements on consistent committed data to multiple different verifiers for multiple times.

Protocol $\Pi_{\mathsf{NICom}\to[\cdot]}$

Let $q \geq 2$ be a prime. Let $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{2\lambda}$ be a cryptographic hash function modeled as a random oracle, and $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{F}_q^m$ be a pseudorandom function.

Compute a public commitment: A prover $\mathcal{P}$ computes and publishes a *non-interactive* commitment on values:

(1) Sample $sk, r \leftarrow \{0,1\}^\lambda$; compute $\mathsf{Commit}_0 := \mathsf{H}(sk, r)$.
(2) On input $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell \in \mathbb{F}_q^m$ with $\ell, m \in \mathbb{N}$, compute $\boldsymbol{c}_i := \mathsf{PRF}(sk, i) + \boldsymbol{x}_i \in \mathbb{F}_q^m$ for $i \in [1, \ell]$.
(3) Compute $d_i := \mathsf{H}(\boldsymbol{c}_i)$ for all $i \in [1, \ell]$; build a Merkle tree on these values using H with Commit_1 as the root.
(4) Publish the commitment $(\mathsf{Commit}_0, \mathsf{Commit}_1)$.

Initialize: Prover $\mathcal{P}$ and a verifier $\mathcal{V}$ send (init) to $\mathcal{F}_{\mathsf{authZK}}$, which returns two uniform global keys to $\mathcal{V}$.

Convert committed values into authenticated values: This procedure can be executed multiple times. For some $i \in [1, \ell]$, $\mathcal{P}$ and $\mathcal{V}$ convert a committed value $\boldsymbol{x}_i \in \mathbb{F}_q^m$ only known by $\mathcal{P}$ to m authenticated values $[x_{i,1}], \ldots, [x_{i,m}]$:

(1) Let path_i be the set containing all siblings of the nodes in the path from the i-th leaf to the root Commit_1. Prover $\mathcal{P}$ sends $(\boldsymbol{c}_i, \mathsf{path}_i)$ to $\mathcal{V}$, who verifies that $\mathsf{H}(\boldsymbol{c}_i)$ is a leaf node rooted in Commit_1.
(2) By calling functionality $\mathcal{F}_{\mathsf{authZK}}$, the parties obtain authenticated bit-vectors $[\boldsymbol{sk}]_2, [\boldsymbol{r}]_2$ on key sk and randomness r, and then $\mathcal{P}$ proves in zero-knowledge that $\mathsf{Commit}_0 = \mathsf{H}([\boldsymbol{sk}]_2, [\boldsymbol{r}]_2)$.
(3) The parties call functionality $\mathcal{F}_{\mathsf{authZK}}$ to compute $([x_{i,1}], \ldots, [x_{i,m}]) \leftarrow \boldsymbol{c}_i - \mathsf{PRF}([\boldsymbol{sk}]_2, i) \in \mathbb{F}_q^m$, and then output $\{[x_{i,j}]\}_{j\in[1,m]}$.

Figure 5.6. **Protocol for converting committed values into authenticated values in the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model.**

Our commitment-authentication conversion protocol. We present our efficient conversion protocol in Figure 5.6. This protocol consists of two phases: 1) generating a non-interactive commitment and 2) converting publicly committed values to privately authenticated values in an interactive manner. To commit a large volume of data or different types of data, we divide them into pieces, where the i-th piece is denoted by $\boldsymbol{x}_i \in \mathbb{F}_q^m$ with a prime $q \geq 2$ and a parameter m. Then, we let the prover sample a random key sk and a uniform randomness r both in $\{0,1\}^\lambda$. Our commitment consists

of $\mathsf{Commit}_0 = \mathsf{H}(sk, r)$ and $\boldsymbol{c}_i = \mathsf{PRF}(sk, i) + \boldsymbol{x}_i \in \mathbb{F}_q^m$ for all $i \in [1, \ell]$ with some $\ell \in \mathbb{N}$, where H is a random oracle and PRF is a pseudorandom function. To perform conversion, the prover $\mathcal{P}$ proves knowledge of sk and $\boldsymbol{x}_i$, such that Commit_0 and $\boldsymbol{c}_i$ are computed with the key and data piece. Since $\boldsymbol{c}_i$ can be put in the public domain, one can further reduce the size of the overall commitment by computing a Merkle tree on top of all $\boldsymbol{c}_i$'s. In this way, the commitment only has a size of 4λ bits, including Commit_0 and the root of the Merkle tree (i.e., Commit_1).

Since key $sk \in \{0,1\}^\lambda$ has a high entropy, we can actually remove the randomness r. That is, the prover can just set $\mathsf{H}(sk)$ as Commit_0 in the commitment phase and prove $\mathsf{Commit}_0 = \mathsf{H}([\boldsymbol{sk}]_2)$ in the conversion phase. This will slightly improve the efficiency of this protocol.

Theorem 13. *Let* H *be a random oracle and* PRF *be a pseudorandom function. Then protocol* $\Pi_{\mathsf{NICom}\to[\cdot]}$ *shown in Figure 5.6 UC-realizes the* $\mathsf{convertC2A}$ *command of functionality* $\mathcal{F}_{\mathsf{authZK}}$ *in the presence of a static, malicious adversary in the* $\mathcal{F}_{\mathsf{authZK}}$*-hybrid model.*

Below, we discuss the intuition of the above theorem and provide the formal security proof. We commit to sk using a standard UC commitment in the random-oracle model, and so sk is computationally hiding, meaning that $\mathsf{PRF}(sk, i)$ for all $i \in [1, \ell]$ are indistinguishable from uniformly random values in $\mathbb{F}_q^m$. In the $\mathcal{F}_{\mathsf{authZK}}$-hybrid model, the ZK proof does not reveal any information of committed values. Overall, the committed data is hidden. In the proof of security, the simulator can extract the key sk from Commit_0 in the random-oracle model. Once sk was extracted, the simulator can easily recover $\boldsymbol{x}_i$ by decrypting $\boldsymbol{c}_i$ for $i \in [1, \ell]$. This also implies the binding property. Together with the

soundness of the ZK protocol realizing $\mathcal{F}_{\mathsf{authZK}}$, we can ensure the consistency between authenticated values and committed values.

Note that this commitment $(\mathsf{Commit}_0, \mathsf{Commit}_1)$ itself is not equivocal if we use the natural "open" algorithm that sends (sk, r): although it is possible to equivocate the key sk to any value by programming the random oracle, the function PRF is fixed. Equivocating from $\boldsymbol{x}_i$ to $\boldsymbol{x}'_i$ would require finding a key sk' such that $\mathsf{PRF}(sk', i) - \mathsf{PRF}(sk, i) = \boldsymbol{x}_i - \boldsymbol{x}'_i$ over $\mathbb{F}_q^m$. However, we can make it equivocal by an interactive opening: instead of directly sending (sk, r), we can send $\boldsymbol{c}_i$ and the corresponding path that can be verified with Commit_1, and prove knowledge of a key sk and a randomness r such that $\mathsf{Commit}_0 = \mathsf{H}(sk, r)$ and the other relationship on $\boldsymbol{c}_i$ hold. In this way, we can use the zero-knowledge property to equivocate the commitment.

Proof. We prove the security against a malicious prover and a malicious verifier separately. In each case, we construct a PPT simulator $\mathcal{S}$, who runs a PPT adversary $\mathcal{A}$ as a subroutine and emulates $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands. In both cases, we show that no PPT environment $\mathcal{Z}$ can distinguish between the real-world execution and ideal-world execution.

Malicious prover. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ simulates random oracle H by recording the queries made by $\mathcal{A}$, and sending the random responses to $\mathcal{A}$ when keeping the consistency of responses.

(2) In the commitment phase, after receiving $(\mathsf{Commit}_0, \mathsf{Commit}_1)$ from $\mathcal{A}$, simulator $\mathcal{S}$ extracts sk by retrieving the H query whose output is Commit_0. If no such

query or there exists two different queries whose outputs are Commit_0, $\mathcal{S}$ sends abort to functionality $\mathcal{F}_{\mathsf{authZK}}$ and aborts.

(3) $\mathcal{S}$ extracts $\boldsymbol{c}_i$ for all $i \in [1, \ell]$ by retrieving the random-oracle queries whose responses are identical to the values from the root Commit_1 of the Merkle tree to the leaves. If no such queries is found or there is a collision for H, $\mathcal{S}$ sends abort to functionality $\mathcal{F}_{\mathsf{authZK}}$ and aborts. Otherwise, for $i \in [1, \ell]$, $\mathcal{S}$ computes $\boldsymbol{x}_i := \boldsymbol{c}_i - \mathsf{PRF}(sk, i) \in \mathbb{F}_q^m$, and then sends $(\mathsf{commit}, x_{i,j}, q)$ to $\mathcal{F}_{\mathsf{authZK}}$ for $j \in [1, m]$.

(4) For a conversion execution, after receiving $(\boldsymbol{c}'_i, \mathsf{path}_i)$ from $\mathcal{A}$ for some $i \in [1, \ell]$, $\mathcal{S}$ checks that $\boldsymbol{c}'_i = \boldsymbol{c}_i$ where $\boldsymbol{c}_i$ is extracted as above. If the check fails, $\mathcal{S}$ aborts.

(5) In the procedure of proving $\mathsf{Commit}_0 = \mathsf{H}([\boldsymbol{sk}]_2, [\boldsymbol{r}]_2)$, $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands by recording sk' that consists of the bits sent to $\mathcal{F}_{\mathsf{authZK}}$ by $\mathcal{A}$. If $sk' \neq sk$ then $\mathcal{S}$ aborts, where sk is the secret key extracted by $\mathcal{S}$.

(6) During the process of evaluating authenticated values from $\boldsymbol{c}_i - \mathsf{PRF}([\boldsymbol{sk}]_2, i)$ for some $i \in [1, \ell]$, $\mathcal{S}$ continues to emulate $\mathcal{F}_{\mathsf{authZK}}$ by storing the output values $x_{i,1}, \ldots, x_{i,m} \in \mathbb{F}_q$, and recording their corresponding MAC tags sent to $\mathcal{F}_{\mathsf{authZK}}$ by $\mathcal{A}$. For $j \in [1, m]$, $\mathcal{S}$ sends the MAC tag on $x_{i,j}$ to functionality $\mathcal{F}_{\mathsf{authZK}}$ for the $(\mathsf{convertC2A})$ command.

In the random-oracle model, if no related query is made by adversary $\mathcal{A}$, then $\mathcal{A}$ successfully guesses the hash output (either Commit_0 or Commit_1) with probability at most $1/2^{2\lambda} = \mathsf{negl}(\lambda)$. As the output of random oracle H is uniformly random, the probability that there exists a collision is bounded by $q_{\mathsf{H}}^2/2^{2\lambda} = \mathsf{negl}(\lambda)$, where q_{H} is the number of queries made by adversary $\mathcal{A}$ to random oracle H. In the following, we assume that the

bad events do not happen. In this case, $\mathcal{S}$ can successfully extract key sk from Commit_0. Based on the security of Merkle trees, $\mathcal{S}$ can also extract the $\boldsymbol{c}_i$ for all $i \in [1, \ell]$ by observing the random-oracle queries and responses. Therefore, all the committed values can be extracted successfully by $\mathcal{S}$ via computing $\boldsymbol{x}_i := \boldsymbol{c}_i - \mathsf{PRF}(sk, i)$ for all $i \in [1, \ell]$.

In the ideal-world execution, for the emulation of $\mathcal{F}_{\mathsf{authZK}}$, $\mathcal{S}$ directly checks whether the secret key sk' input by $\mathcal{A}$ is consistent with sk extracted by it. In the real-world execution, the secret key sk' input by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{authZK}}$ is the same as that committed by it, unless a collision for H is found with probability $q_{\mathsf{H}}/2^{2\lambda}$. In the following, we assume that the secret key input by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{authZK}}$ is consistent with that committed by itself. According to the definition of functionality $\mathcal{F}_{\mathsf{authZK}}$, the output values $x_{i,1}, \ldots, x_{i,m}$ in the conversion phase are consistent with the values that have been committed by $\mathcal{A}$ in the commitment phase.

Overall, the PPT environment $\mathcal{Z}$ cannot distinguish the real-world execution from the ideal-world execution, except with probability $\mathsf{negl}(\lambda)$.

Malicious verifier. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

(1) $\mathcal{S}$ simulates H by recording the queries made by $\mathcal{A}$, and responds with random strings when keeping the consistency.

(2) In the commitment phase, $\mathcal{S}$ samples $\mathsf{Commit}_0 \leftarrow \{0,1\}^{2\lambda}$ and $\boldsymbol{c}_i \leftarrow \mathbb{F}_q^m$ for $i \in [1, \ell]$. Then, it computes Commit_1 as a root of Merkle tree with leaves $\boldsymbol{c}_i$ for all $i \in [1, \ell]$ following the protocol description, and then sends $(\mathsf{Commit}_0, \mathsf{Commit}_1)$ to $\mathcal{A}$.

(3) In the initialization procedure, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ with circuit-based commands by recording two global keys, sent to $\mathcal{F}_{\mathsf{authZK}}$ by $\mathcal{A}$. Then $\mathcal{S}$ sends the global keys to functionality $\mathcal{F}_{\mathsf{authZK}}$ in the ideal-world execution.

(4) In the conversion phase, for some $i \in [1, \ell]$, $\mathcal{S}$ sends $(\boldsymbol{c}_i, \mathsf{path}_i)$ to $\mathcal{A}$ following the protocol specification. Then, for the procedures that proving knowledge of (sk, r) such that $\mathsf{Commit}_0 = \mathsf{H}(sk, r)$ and computing authenticated values, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{authZK}}$ with only circuit-based commands by recording the local keys, sent to $\mathcal{F}_{\mathsf{authZK}}$ by $\mathcal{A}$.

(5) Finally, $\mathcal{S}$ computes and sends the local keys on $[x_{i,j}]$ for $j \in [1, m]$ to functionality $\mathcal{F}_{\mathsf{authZK}}$.

For the emulation of $\mathcal{F}_{\mathsf{authZK}}$, the simulation of $\mathcal{S}$ is perfect. Thus, we focus on proving the indistinguishability of commitment $(\mathsf{Commit}_0, \mathsf{Commit}_1)$. The probability that adversary $\mathcal{A}$ makes the query (sk, r) to random oracle H is at most $q_{\mathsf{H}}/2^{2\lambda}$. Therefore, Commit_0 simulated by $\mathcal{S}$ is computationally indistinguishable from the real value, except with probability $\mathsf{negl}(\lambda)$. By the pseudo-randomness of PRF, we have that the $\{\boldsymbol{c}_i\}_{i\in[1,\ell]}$ computed with PRF and key sk are computationally indistinguishable from uniformly random vectors in $\mathbb{F}_q^m$. Given the $\{\boldsymbol{c}_i\}_{i\in[1,\ell]}$, $\mathcal{S}$ computes Commit_1 in the same way as the real protocol execution. Thus, the simulation for Commit_1 is also computationally indistinguishable from the real value. Overall, no PPT environment $\mathcal{Z}$ can distinguish between the real-world execution and ideal-world execution, except with probability $\mathsf{negl}(\lambda)$. This completes the proof. □

Instantiation of PRF. We use LowMC [1] to instantiate PRF for reducing circuit complexity. One issue with LowMC is that it contains a lot of XOR gates. Although they are free cryptographically, the computation complexity can be fairly high. We adopt the following optimizations for competitive performance:

- Similar to the signature scheme Picnic [106], we need to run PRF on a single key for many times, and thus can precompute the matrix multiplication about the key *only once* and use it for all PRF evaluations.
- We pick the block size as 64 bits to further reduce the number of XOR operations. The resulting protocol is highly efficient, and can convert 18,000 publicly committed data blocks (totally 144KB) to authenticated values per second.
- To reduce the number of rounds in LowMC, we choose the data complexity to be 2^{30} blocks, which is sufficient to commit 8 GB data. If the data is larger than that, we can just pick a new PRF key and commit this key.

Comparing with other candidates. We briefly discuss the concrete efficiency of our protocol for one commitment-authentication conversion, and compares it with other alternatives shown in Table 5.1. Here, we ignore the efficiency comparison for the commitment-generation phase, as it needs to be executed only once.

Scheme	This work	SHA-256	LowMC-256
Time (μs)	55	395	$\geq$ 1000
Comm. (bits)	62	705	49

Table 5.1. **Efficiency comparison between our protocol and alternative protocol with natural commitments.** Running time in microsecond (μs) is based on two Amazon EC2 machines of type `m5.2xlarge`.

For SHA-256 and LowMC-256, they refer to building a hash function modeled as a random oracle, and further construct a commitment on message x via $\mathsf{H}(x, r)$ with a randomness r. For SHA-256, one invocation takes 22573 AND gates and can commit 256 bits of messages. For LowMC-256, we first pick a LowMC block cipher with 256-bit key and block sizes, and then use Davies–Meyer to build a hash function. The SHA-256 method requires a lot of communication due to a large circuit size. The LowMC-256 approach is significantly slower compared to ours because: 1) our 64-bit block cipher only computes 64-bit matrix multiplication, but LowMC-256 needs 256-bit matrix multiplication meaning 16 times more operations; 2) we only need 11 rounds but LowMC-256 needs 53 rounds; 3) we can use a fixed key for all messages but LowMC-256 needs to rekey for every block of the message.

Conversion from authenticated values to publicly committed values. In some applications, two parties $\mathcal{P}$ and $\mathcal{V}$ may want to convert authenticated values (say, output by some MPC protocol) into a public commitment on the same values. Based on the protocol $\Pi_{\mathsf{NICom}\to[\cdot]}$ shown in Figure 5.6, this is easy to be realized by the following execution:

(1) To convert authenticated values $\{[x_{i,j}]\}_{i\in[1,\ell],j\in[1,m]}$ into publicly committed values, $\mathcal{P}$ commits these vectors $(x_{i,1}, \ldots, x_{i,m})$ for $i \in [1, \ell]$ by executing the commitment-generation phase of protoocol $\Pi_{\mathsf{NICom}\to[\cdot]}$. Then $\mathcal{P}$ publishes the resulting commitment $(\mathsf{Commit}_0, \mathsf{Commit}_1)$.

(2) Then, $\mathcal{P}$ and $\mathcal{V}$ execute protocol $\Pi_{\mathsf{NICom}\to[\cdot]}$ to convert commitment $(\mathsf{Commit}_0, \mathsf{Commit}_1)$ into authenticated values $\{[x'_{i,j}]\}_{i\in[1,\ell],j\in[1,m]}$.

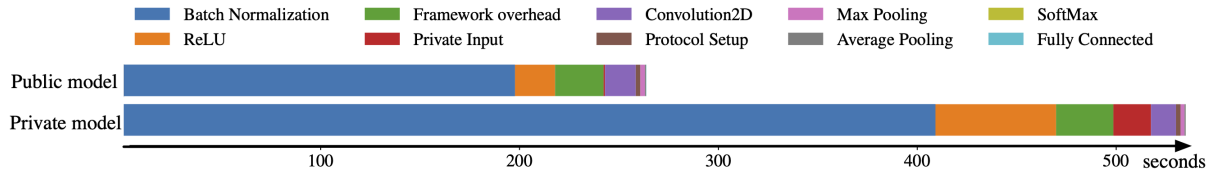

Figure 5.7. **Execution-time decomposition for ResNet-101 Inference.** The top bar is for public-model private-feature inference; the bottom bar is for private-model private-feature inference. The network bandwidth is throttled to 200 Mbps.

(3) The parties call the CheckZero procedure on $[x'_{i,j}] - [x_{i,j}]$ for all $i \in [1, \ell], j \in [1, m]$, and abort if the check fails.

5.3. Performance Evaluation

In this section, we benchmark the speed of Mystique and how it performs on large-scale ML-inspired applications. We used three neural network models: LeNet-5[1] (5 layers, 62000 model parameters), ResNet-50 (50 layers, 23.5 million model parameters), and ResNet-101 (101 layers, 42.5 million model parameters). All experimental results are obtained by running the protocol over two Amazon EC2 machines of type `m5.2xlarge`, each with 32 GB memory. We use all CPU resources but only a fraction of the memory. The largest example is for ResNet-101 that uses 12 GB of memory. Our implementations use the latest sVOLE-based protocol [101] as the underlying ZK proof. All our implementations achieve the computational security parameter $\lambda = 128$ and statistical security parameter $\rho \geq 40$.

	50 Mbps	200 Mbps	500 Mbps	1 Gbps
Conversions				
A2B	107 μs	45 μs	34 μs	29 μs
B2A	109 μs	49 μs	38 μs	33 μs
C2A	56 μs	55 μs	55 μs	55 μs
Fix2Float	50 μs	46 μs	46 μs	46 μs
Float2Fix	49 μs	46 μs	46 μs	46 μs
Machine Learning (ML) Functions				
Sigmoid	2.1 ms	1.6 ms	1.6 ms	1.6 ms
Max Pooling	1.6 ms	0.5 ms	0.4 ms	0.4 ms
ReLU	908 μs	262 μs	185 μs	188 μs
SoftMax-10	209 ms	157 ms	161 ms	171 ms
Batch Norm	415 ms	261 ms	257 ms	269 ms
Matrix Multiplications				
MatMult-512	361 ms	186 ms	185 ms	185 ms
MatMult-1024	2.42 s	1.48 s	1.39 s	1.37 s
MatMult-2048	15.19 s	11.30 s	10.63 s	10.39 s

Table 5.2. **Performance of the basic building blocks.** The dimension of Max Pooling is 2×2. The dimension of Batch Normalization is $[1, 16, 16, 4]$, which stands for the batch size, height, weight and channels. For ML functions, the inputs and outputs are authenticated values in $\mathbb{F}_p$ with $p = 2^{61} - 1$. The performance result assumes that the inputs and outputs are all private to the verifier.

5.3.1. Benchmarking Our Building Blocks

We test the performance of our key building blocks discussed in this paper and summarized the results in Table 5.2. From this table, we can see that our protocol is highly scalable and all basic operations are highly efficient. The arithmetic-Boolean conversion (i.e., A2B and B2A) consists of two phases. In the preprocessing phase, two parties generate random zk-edaBits, and the execution time per zk-edaBit decreases from 95 μs to 19 μs when the bandwidth increases from 50 Mbps to 1 Gbps. In the online phase, two parties can convert

[1]We use ReLU as activation function instead of tanh for better accuracy.

authenticated wire values between arithmetic and Boolean circuits cheaply. The efficiency of the conversion from a public commitment to privately authenticated values (i.e., C2A) is mainly dominated by the computation of PRF in a Boolean circuit. It only takes around 56 μs to apply the PRF to a 64-bit data block, when the network bandwidth is at least 50 Mbps, due to the high communication efficiency of our protocol. The terms Fix2Float and Float2Fix represent the conversions between fixed-point and floating-point numbers, where the execution time for both conversions is around 46 μs per conversion when the network bandwidth is larger than 50 Mbps.

For the ZK proof of matrix multiplication (i.e., MatMul), our protocol can obtain around 185 ms of execution time for dimension 512×512, when the network bandwidth is at least 200 Mbps. The execution time is increased to about 1.5 s and 11 s for dimensions 1024×1024 and 2048×2048, respectively. The main efficiency bottleneck is the local computation of matrix multiplication by the prover. Compared to the state-of-the-art ZK proof for matrix multiplication [**101**], which takes 10 seconds to prove a 1024×1024 matrix multiplication over a network bandwidth of 500 Mbps, our ZK protocol achieves a $7\times$ improvement.

5.3.2. Benchmarking Private Inference

With these building blocks, we connect them together to build a ZK system to prove the inference of large neural networks. We consider three canonical settings, where the model parameters and model feature input can either be private to the prover or public to both parties. We focus on three neural networks: LeNet-5, ResNet-50, and ResNet-101. While

Model	Image	LeNet-5	ResNet-50	ResNet-101
Communication				
Private	Private	16.5 MB	1.27 GB	1.98 GB
Private	Public	16.5 MB	1.27 GB	1.98 GB
Public	Private	16.4 MB	0.53 GB	0.99 GB
Execution time (seconds) in a 50 Mbps network				
Private	Private	7.3	465	736
Private	Public	7.5	463	735
Public	Private	6.5	210	369
Execution time (seconds) in a 200 Mbps network				
Private	Private	5.9	333	535
Private	Public	5.5	336	541
Public	Private	4.9	158	262

Table 5.3. **Performance of zero-knowledge neural-network inference.** All models are trained using the CIFAR-10 dataset.

the first example is relatively simple, the last two examples represent the state-of-the-art neural networks in terms of accuracy and complexity.

In Table 5.3, we summarize the performance for all neural networks, where the commitment on a model or data is not involved. After all optimizations, the slowest component in our protocol is Batch Normalization, which only exists in ResNet-50 and ResNet-101. For all models, we observe that when the model is private, the overall execution time is higher than the case in which the model parameters are public. This is because more operations have to be done in ZK proofs for private models. Regardless of this setting, LeNet-5 inference takes several seconds to finish. For all settings, ResNet-50 (resp., ResNet-101) takes about 2.6–5.6 (resp., 4.4–9) minutes to accomplish under a 200 Mbps network.

Microbenchmark. Figure 5.7 reports the microbenchmark of our ResNet-101 inference. We collect the time usage of different components including the protocol setup,

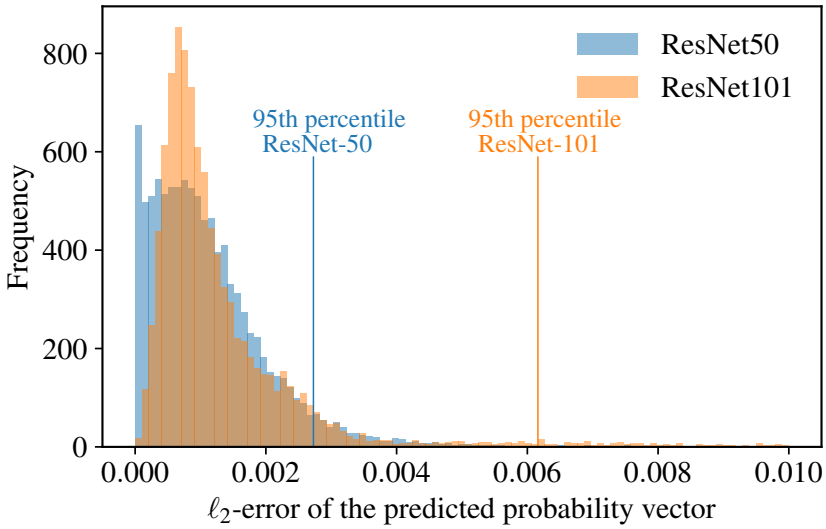

Figure 5.8. **ℓ_2-norm distance between the plaintext-inference probability vector and the ZK-inference probability vector.** The mean difference is 0.0011 for ResNet-50 and 0.0019 for ResNet-101.

private input (i.e., computing corresponding authenticated values), different operators and framework overhead. Significant amount of costs are used in Batch Normalization, ReLU, convolution2D and framework overhead. When the model is private, an additional proportion of time will also be used for private input. Note that the Batch Normalization takes around 70% of time in both cases because it involves complicated arithmetic operations and conversions between floating-point and fixed-point numbers, which are costly to maintain accuracy. It will be an interesting future work to further improve the efficiency of Batch Normalization and ReLU without losing accuracy.

Benchmarking the accuracy. Our approach is highly accurate, but could still cause some accuracy loss. This could particularly be a concern for deep neural networks with hundred of layers where the error could propagate and get amplified. To benchmark the accuracy of our protocol, we ran the whole CIFAR-10 testing dataset [80] containing 10000

ML applications	LeNet-5	ResNet-50	ResNet-101
ZK for evasion attacks	9.8 s	316 s	524 s
ZK for genuine inference	7.2 s	16.4 m	28 m
ZK for private benchmark	8.2 m	4.4 h	7.3 h

Table 5.4. **Efficiency of our ZK system in different applications.** All execution time is reported based on a 200 Mbps network and two `m5.2xlarge` machines.

imagines. CIFAR is one of the standard ML dataset to benchmark the performance of algorithms. Imagines in CIFAR-10 are all labeled within 10 different classes, each imagine is a 32×32 color picture. The accuracy difference between the plaintext model and our ZK model is only 0.02% for both ResNet-50 and ResNet-101. To further understand the accuracy difference, we also compare the underlying probability vector predicted for each testing imagine. The dataset CIFAR-10 has 10 classes, and thus each inference produces a probability vector of length 10, denoted as $\boldsymbol{p}_i$ for all $i \in [1, 10000]$. The final prediction of the i-th testing imagine is $\mathsf{ArgMax}_i(\boldsymbol{p}_i)$. We are interested in the distribution of $\|\boldsymbol{p}_i - \boldsymbol{p}'_i\|_2$, where $\boldsymbol{p}_i$ is from plaintext inference and $\boldsymbol{p}'_i$ is from ZK inference. In Figure 5.8, we show the ℓ_2-norm differences of all 10000 inferences, and we can see that even for ResNet-101, the ℓ_2-norm difference is smaller than 0.006 for 95% of the case. For LeNet-5, 99.9% of the ℓ_2-norm difference are below 0.006. Therefore, for top-k accuracy such as $k = 5$ (commonly used for ImageNet), our ZK inference will be highly accurate.

5.3.3. End-to-End Applications

By connecting the private models/features to publicly committed models/features, Mystique can be used to build the three end-to-end applications mentioned in the Introduction. Since we use CIFAR-10 dataset, each image is of size 32×32 pixels and each pixel uses

3 bytes to represent the color. This means that one image is of size 3072 bytes and takes about 2.6 milliseconds to convert from publicly committed values to privately authenticated values. The sizes of three models considered in this paper are 0.25 MB, 94 MB, and 170 MB. They take 1.7 seconds, 646 seconds, and 1169 seconds to convert from a public commitment to authenticated values that can be used in our protocols directly. The cost to "pull" a publicly committed model to be used in ZK proofs is high, but it could always be amortized over multiple private inferences.

- **ZK proofs for evasion attacks.** In this case, we need to prove knowledge of two almost identical inputs that get classified to different results under a public model. Therefore, the main cost is to prove the classification result in zero-knowledge under a public model twice.
- **ZK proofs for genuine inference.** In this application, the model parameters are private but publicly committed, while the input data is public. The main overhead is from: 1) proving the consistency between committed values and authenticated values for all model parameters; and 2) proving correct classification with private model and public input.
- **ZK proofs for private benchmark.** In this application, the testing data set is publicly committed and the model is public. Therefore, the main overhead comes from: 1) proving the consistency between committed testing data and authenticated data; and 2) proving correct classification with private input data and public model. In our example, we assume a testing data set of 100 images, and thus the second step is executed for 100 times, once for each image.

The execution time for every end-to-end application is reported in Table 5.4. Note that in the "ZK for private benchmark" application, 100 testing images were publicly committed, and then are converted to privately authenticated values using our conversion protocol shown in Section 5.2. Thus, the execution time for this application is significantly higher.

References

[1] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, Heidelberg, April 2015.

[2] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[3] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862. IEEE Computer Society Press, May 2017.

[4] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.

[5] Laasya Bangalore, Rishabh Bhadauria, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. On black-box constructions of time and space efficient sublinear arguments from symmetric-key primitives. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part I*, volume 13747 of *LNCS*, pages 417–446. Springer, Heidelberg, November 2022.

[6] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 192–211. ACM Press, November 2021.

[7] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 329–358. Springer, Heidelberg, August 2022.

[8] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions. Cryptology ePrint Archive, Report 2020/1410, 2020. https://eprint.iacr.org/2020/1410.

[9] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Heidelberg.

[10] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 495–526. Springer, Heidelberg, May 2020.

[11] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 390–420. Springer, Heidelberg, August 1993.

[12] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, October / November 2016.

[13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. Cryptology ePrint Archive, Report 2012/095, 2012. https://eprint.iacr.org/2012/095.

[14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 111–120. ACM Press, June 2013.

[15] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 255–272. Springer, Heidelberg, August 2012.

[16] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai,

editor, *TCC 2013*, volume 7785 of *LNCS*, pages 315–333. Springer, Heidelberg, March 2013.

[17] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 168–197. Springer, Heidelberg, November 2020.

[18] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 123–152, Virtual Event, August 2021. Springer, Heidelberg.

[19] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic SNARKs for diverse environments. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 427–457. Springer, Heidelberg, May / June 2022.

[20] Jonathan Bootle, Vadim Lyubashevsky, and Gregor Seiler. Algebraic techniques for short(er) exact lattice-based zero-knowledge proofs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 176–202. Springer, Heidelberg, August 2019.

[21] Cecilia Boschini, Jan Camenisch, Max Ovsiankin, and Nicholas Spooner. Efficient post-quantum SNARKs for RSIS and RLWE and their applications to privacy. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 247–267. Springer, Heidelberg, 2020.

[22] Florian Bourse, Rafaël del Pino, Michele Minelli, and Hoeteck Wee. FHE circuit privacy almost for free. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 62–89. Springer, Heidelberg, August 2016.

[23] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[24] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[25] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

[26] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. Homomorphic secret sharing: Optimizations and applications. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2105–2122. ACM Press, October / November 2017.

[27] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[28] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[29] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg, August 2012.

[30] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[31] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 681–710, Virtual Event, August 2021. Springer, Heidelberg.

[32] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.

[33] Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. Compiling low depth circuits for practical secure computation. In Ioannis G. Askoxylakis,

Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 80–98. Springer, Heidelberg, September 2016.

[34] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[35] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[36] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

[37] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[38] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 532–550. Springer, Heidelberg, December 2014.

[39] Sourav Das, Zhuolun Xiang, and Ling Ren. Powers of tau in asynchrony. Cryptology ePrint Archive, Report 2022/1683, 2022. https://eprint.iacr.org/2022/1683.

[40] Leo de Castro, Chiraag Juvekar, and Vinod Vaikuntanathan. Fast vector oblivious linear evaluation from ring learning with errors. Cryptology ePrint Archive, Report 2020/685, 2020. https://eprint.iacr.org/2020/685.

[41] Leo de Castro, Chiraag Juvekar, and Vinod Vaikuntanathan. Fast vector oblivious linear evaluation from ring learning with errors. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography – WAHC'21*, pages 29–41. ACM, 2021.

[42] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, November 2021.

[43] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 829–841. ACM Press, November 2022.

[44] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In *ACM CCS 2022*. ACM Press, November 2022.

[45] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Report 2020/1446, 2020. https://eprint.iacr.org/2020/1446.

[46] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography*, 2021.

[47] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. SPARKs: Succinct parallelizable arguments of knowledge. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 707–737. Springer, Heidelberg, May 2020.

[48] Muhammed F. Esgin, Ngoc Khanh Nguyen, and Gregor Seiler. Practical exact proofs from lattices: New techniques to exploit fully-splitting rings. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 259–288. Springer, Heidelberg, December 2020.

[49] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[50] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[51] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017,*

Part II, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

[52] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[53] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[54] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[55] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

[56] Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 116–129. Springer, Heidelberg, August 1999.

[57] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[58] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to prove all NP-statements in zero-knowledge, and a methodology of cryptographic protocol design. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 171–185. Springer, Heidelberg, August 1987.

[59] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[60] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 193–226. Springer, Heidelberg, August 2023.

[61] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris

Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 275–304, Virtual Event, August 2021. Springer, Heidelberg.

[62] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 321–340. Springer, Heidelberg, December 2010.

[63] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[64] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2014.

[65] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. TinyKeys: A new approach to efficient multi-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2018.

[66] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[67] Justin Holmgren and Ron Rothblum. Delegating computations with (almost) minimal time and space overhead. In Mikkel Thorup, editor, *59th FOCS*, pages 124–135. IEEE Computer Society Press, October 2018.

[68] Russell Impagliazzo, Leonid A. Levin, and Michael Luby. Pseudo-random generation from one-way functions (extended abstracts). In *21st ACM STOC*, pages 12–24. ACM Press, May 1989.

[69] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[70] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[71] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 575–594. Springer, Heidelberg, February 2007.

[72] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[73] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[74] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[75] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[76] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

[77] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758, 2022. https://eprint.iacr.org/2022/1758.

[78] Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023. https://eprint.iacr.org/2023/573.

[79] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388. Springer, Heidelberg, August 2022.

[80] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[81] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. Zksql: Verifiable and efiicient query evaluation with zero-knowledge proofs. *Proceedings of the VLDB Endowment*, 16(8):1804–1816, 2023.

[82] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.

[83] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[84] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. Cryptology ePrint Archive, Report 2022/1592, 2022. https://eprint.iacr.org/2022/1592.

[85] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.

[86] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[87] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1055–1072. ACM Press, November 2019.

[88] Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL, March 2022. Microsoft Research, Redmond, WA.

[89] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020.

[90] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023. https://eprint.iacr.org/2023/552.

[91] Alan Szepieniec. Polynomial IOPs for linear algebra relations. Cryptology ePrint Archive, Report 2020/1022, 2020. https://eprint.iacr.org/2020/1022.

[92] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 71–89. Springer, Heidelberg, August 2013.

[93] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and abhi shelat. Blind certificate authorities. In *2019 IEEE Symposium on Security and Privacy*, pages 1015–1032. IEEE Computer Society Press, May 2019.

[94] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[95] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Report 2020/925, 2020. `https://eprint.iacr.org/2020/925`.

[96] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

[97] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 501–518. USENIX Association, August 2021.

[98] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. AntMan: Interactive zero-knowledge proofs with sublinear communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2901–2914. ACM Press, November 2022.

[99] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 299–328. Springer, Heidelberg, August 2022.

[100] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

[101] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field.

Cryptology ePrint Archive, Report 2021/076, 2021. https://eprint.iacr.org/2021/076.

[102] Kang Yang and Xiao Wang. Non-interactive zero-knowledge proofs to multiple verifiers. Cryptology ePrint Archive, Report 2022/063, 2022. https://eprint.iacr.org/2022/063.

[103] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1627–1646. ACM Press, November 2020.

[104] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626. ACM Press, November 2020.

[105] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. Tota: Fully homomorphic encryption with smaller parameters and stronger security. Cryptology ePrint Archive, Paper 2021/1347, 2021. https://eprint.iacr.org/2021/1347.

[106] Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Jonathan Katz, Xiao Wang, and Vladmir Kolesnikov. Picnic. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions.

[107] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1919–1938. ACM Press, November 2020.

[108] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy*, pages 859–876. IEEE Computer Society Press, May 2020.