GKR for Boolean Circuits with Sub-linear RAM Operations

Yuncong Hu*, Chongrong Li*, Zhi Qiu+, Tiancheng Xie+,
Yue Ying†, Jiaheng Zhang†, Zhenfei Zhang+,

*Shanghai Jiao Tong University, +Polyhedra Networks, †National University of Singapore

huyuncong@sjtu.edu.cn, lichongrong9@gmail.com, chonps@polyhedra.network, tc@polyhedra.network,
yolanda.yueying@gmail.com, jhzhang@nus.edu.sg, zhenfei@polyhedra.network

***Abstract*—Succinct Non-Interactive Arguments of Knowledge (SNARKs) provide a powerful cryptographic framework enabling short, quickly verifiable proofs for computational statements. Existing SNARKs primarily target computations represented as arithmetic circuits. However, they become inefficient when handling binary operations, as each gates operates on a few bits while still incurring the cost of multiple field operations per gate. This inefficiency stems, in part, from their inability to capture the word-level operations that are typical in real-world programs. To better reflect this computational pattern, we shift our attention to data-parallel boolean circuits, which serve as a natural abstraction of word-level operations by allowing parallel munipulation of multiple bits. To precisely characterize the prover's overheads in our scheme, we adopt the word RAM model, which aligns with the nature of modern programming languages. Under this model, we propose a novel approach to achieve SNARKs with only sub-linear prover overhead for proving boolean circuits.**

Specifically, we present an optimized GKR protocol for boolean circuits that captures the word-level operations. To achieve this, we pack multiple bits as a single univariate polynomial, and exploiting the binary nature of circuit values to enable precomputation to accelerate the sumcheck process. This optimization leads to a highly efficient prover requiring only sub-linear RAM operations. Furthermore, we introduce a sub-linear polynomial commitment scheme designed specifically for binary polynomials, which ensures efficient commitments with minimal computational overhead.

Comprehensive evaluations reveal that our scheme achieves both theoretical efficiency and substantial practical performance gains. For instance, in proving randomly generated Boolean circuits with 2^{30} gates, proof generation with our optimized GKR protocol completes in just 5.38 seconds, yielding a $223\times$ speedup over LogUp (Haböck, ePrint 2022), the most efficient known scheme for lookup arguments.

1. Introduction

SNARKs (Succinct Non-Interactive Argument of Knowledge) are cryptographic tools that allow a prover to convince a verifier that a statement is true using a very short proof, which can also be verified in a short time. This makes SNARKs particularly useful in applications such as blockchain [1], [2], [3], and machine learning [4], [5], [6].

Existing SNARKs, including [7], [8], [9], [10], and subsequent advancements [11], [12], [13], primarily focused on generating proofs for computations represented as arithmetic circuits. This approach inherently favors representations based on low-degree polynomials over large finite fields, resulting in suboptimal efficiency for computation classes that lack an intrinsic algebraic structure.

A significant challenge arises in tasks heavily reliant on binary operations, such as many symmetric cryptographic primitives, including SHA-256 and Keccak-256, which are inefficient to express using arithmetic circuits. In contrast, boolean circuits, which directly manipulate binary values through logic gates, provide a more natural framework for these tasks. However, most SNARKs [14] simply treat boolean circuits as arithmetic circuits over binary fields. While recent advancements in polynomial commitment schemes (PCS) over binary field [14], [15], [16] have led to substantial efficiency improvements, the Interactive Oracle Proof (IOP) for boolean circuits remains inefficient. This inefficiency arises because each gate processes only a small number of bits, yet the prover incurs at least constant field operations per gate—an overhead that should ideally be much lower.

Recent theoretical advancements [17], [18] have mitigated these limitations by employing the code-switching technique from [19], enabling proof generation with only constant overhead relative to the boolean circuit size, albeit without guarantees of concrete efficiency. In the context of non-succinct proofs, combining the MPC-in-the-Head framework [20] with linear-time computable hash functions [21] also leads to constant prover overheads for boolean circuits. While these constructions enjoy high concrete efficiency, they suffer from large proof sizes and high verifier costs, limiting their practicality.

Although boolean circuits provide a plausible framework for capturing binary operations, a gap remains between the theoretical model and real-world applications. In practice,

binary operations are rarely performed on isolated bits; instead, they typically operate on words of varying sizes (e.g., 32- or 64-bit integers), where a single bitwise operation processes all bits within the word simultaneously.

Traditional boolean circuits, which operate at the bit level, struggle to efficiently model such word-level computations. In contrast, data-parallel boolean circuits present a promising alternative by mapping distinct bits within a word to different sub-circuits. With additional mechanism to handle shifts operations, they effectively emulate the word-level operations commonly found in practical applications.

An alternative approach is lookup arguments [22], [23], [24], [25], [26], which have been extensively studied and are now widely adopted in ZKP applications [6], [27]. These schemes allow the prover to demonstrate that each element in a committed vector belongs to a pre-determined table, allowing word-level operations to be proved without explicit circuit representations. However, lookup arguments are less flexible than boolean circuits, as they can only process a single operations per proof. Moreover, while they are typically efficient for structured computations, they lack the versatility of boolean circuits. Therefore, in this work, we primarily focus on boolean circuits.

When formulating word-level operations, evaluating efficiency by modeling prover as plain boolean circuits becomes inadequate. Instead, we want to model the operations on bounded-sized words as single-instruction executions. To achieve this, we adopt the word RAM model [28], as it closely aligns with the nature of modern programming languages. Under this model, operations are performed on registers called *words*, where each word is an integer in the range $[0, 2^w]$ and w is typically set to $\log N$ for N-sized computations. All basic operations, including binary operations, on words are assumed to execute in constant time. A detailed discussion of word RAM model can be found in Section 2.2.

Shifting to word-level operations dramatically alters the asymptotic complexity landscape. For instance, N bitwise XOR can be performed with only $\frac{N}{\log N}$ RAM operations by packing $\log N$ bits into words, amounting to only sublinear complexity relative to the total computation size.

Motivated by this observation, we raise the following question: Is it possible to construct a SNARK that efficiently supports boolean circuits with sub-linear time prover under word RAM model? In this work, we answer the question affirmatively by proposing a new SNARK for data-parallel and general boolean circuits with sub-linear RAM operations. At the core of our scheme is an optimization of GKR protocol for boolean circuits.

1.1. Our Contributions

More specifically, our contributions include:

Sub-linear Time GKR for Data-parallel Boolean Circuits. At the core of our scheme is a novel approach for performing sumchecks in GKR protocol tailored to data-parallel boolean circuits, significantly reducing the prover's workload.

Recall that the primary challenge in achieving a linear-time prover under the word RAM model lies in ensuring the prover performs only sub-linear field operations and linear constant-time operations. In our construction, our prover performs $O(\frac{N}{\log N})$ field operations and $O(N)$ binary value readings to prove a data-parallel boolean circuit of total size N. To achieve this, we assume the data-parallel circuits consist of $B = \frac{\log N}{3}$ identical sub-circuits.

In the traditional GKR protocol for data-parallel circuits, evaluations of all sub-circuits at each layer are combined using a $\log B$-variable multilinear polynomial. The sumcheck protocol is then applied recursively on each variable to compute a univariate polynomial in each layer. Although efficient implementations of the sumcheck protocol with linear field operations have been developed [10], [29], [30] for such representation, this strategy exceeds the prover's sub-linear field operations budget.

Our scheme packs all sub-circuits with univariate Lagrange polynomials, leading to a more compact representation. One key insight is leveraging the binary nature of the circuits to precompute possible values in the summation, thus alleviating the prover's workload. Performing sumcheck on this univariate variable requires to perform a summation of size $\frac{N}{B}$, which is sub-linear. Each summation term corresponds to a degree-$O(B)$ polynomial, which can be decomposed to a field elements multiplied by two binary polynomials and one constant polynomial. We demonstrate that the binary nature of the polynomials restricts possible outcomes of polynomials to $O(2^{2B})$. This allows the prover to precompute all possibilities via polynomial multiplication efficiently.

However, directly using the precomputed table is inefficient, as each summation step would leads to $O(B)$ field operations, leading to overall linear field operations. To circumvent this, we accumulate the field element for each polynomial in the table via a single field operation, and reconstruct the final polynomial by traversing the table entries and their accumulators. Since the table size is significantly smaller than summation size, the overall workload would be dominated by the accumulation phase, resulting in a total $O(\frac{N}{\log N})$ field operations, which can be performed by sub-linear RAM operations.

After the first round reduction, the remaining part of sumcheck is inherently sub-linear size, which can be performed efficiently using the implementation from [10].

Remark 1. *Our IOP construction relies on univariate polynomial multiplication; however, the polynomial's degree is $O(\log N)$, which is small enough for plain algorithm to achieve efficiency comparable to fast algorithm like FFT. As a result, the efficiency of our IOP system does not depend on specific properties of the underlying fields. This leads our IOP system field-agnostic and ensures compatibility across diverse application scenarios, free from constraints imposed by field selection.*

Sub-linear Time Polynomial Commitment Scheme for Binary Polynomials. For the arithmetization of layered circuits, polynomial commitment would typically not become

a bottleneck in either theoretic or concrete efficiency, as the commitment cost is limited to the secrete input. In rare cases where the input witness is as large as the whole computation size, polynomial commitment scheme (PCS) become necessary to maintain succinctness. For completeness, we present a sub-linear time PCS for binary polynomials over the binary field (i.e. $\mathbb{F}_2$). While the field choice is primarily driven by theoretical considerations, some optimizations—particularly in the evaluation proof generation—apply to arbitrary fields and may be of independent interest.

Our construction follows Orion [31], which utilizes general Spielman codes proposed in [32], [33]. To commit a polynomial, it is first represented as a matrix and then encode each row individually. Over a prime field, this encoding would lead to linear amount of field addition, which is undesirable. This issue is circumvented by performing the encoding in $\mathbb{F}_2$, which it reduces to linear amount of XOR operations, enabling efficient parallelization via bit-packing. To generate Merkle tree hash of the encoded matrix, we first transpose the encoded matrix with fast algorithm in [34] to gain the row-packed storage of the matrix. Then the merkle tree of multiple columns are computed simultaneously by applying the collision-resistant hash family computable in linear binary operations (e.g., [21]) using the packed rows.

For the opening proof, the verifier selects a random challenge vector from a sufficiently large extension field of $\mathbb{F}_2$ to ensure soundness. The primary computational costs lies in multiplying the challenge vector by the encoded matrix. Since the encoded matrix contains binary values, we show that this product can be done using only sub-linear RAM operations with precomputation.

Table 1 presents a comparison of the time complexity of popular SNARKs with our scheme under word RAM model.
Implementation and Evaluations. We have fully implemented our SNARKs based on the Rust ark-works ecosystem and conducted comprehensive benchmarks on randomly generated Boolean circuits of varying sizes. At 2^{30} gates, our system completes the proof in 5.38 seconds—$9.4\times$ faster than the state-of-the-art work Binius [14] and $223\times$ faster than LogUp [22], the most efficient known scheme for lookup arguments.

1.2. Related Work

GKR. In the seminal work of [35], Goldwasser et al. proposed an efficient interactive proof for layered arithmetic circuits. Xie et al. [10] introduced a variant of the GKR protocol with linear prover time for arbitrary layered arithmetic circuits. For data parallel circuits with copies of small subcircuits with size C', a $O(C \log C')$ protocol is presented by Thaler [30], later improved to $O(C+C' \log C)$ by Wahby et al. in [36]. Zhang et al. [37] generalized the GKR protocol to general circuits instead of layered circuit without any overhead on the prover time.

ZKP from MPC Techniques. ZKP with high concrete efficiency can be derived by utilizing techniques from secure multi-party computation (MPC). A line of work [38], [39], [40], [41], [42], [43] builds upon the MPC-in-the-Head techniques introduced by Ishai et al. [20], which compiles arbitrary MPC protocols into zero-knowledge proofs. When instantiating with constant overhead MPC protocol for boolean circuits [44] with linear time computable hash [21], these constructions can prove boolean circuits with constant overhead and constant soundness error.

However, these approaches face several limitations. Most notably, most of these schemes impose a linear workload on the verifier and require large proof size, which hinders their practicality in certain practical applications. Besides, they only consider bit-level binary operations. In contrast, our scheme have succinct verifier and support the word-level binary operations efficiently simultaneously.

Lookup Arguments. Lookup arguments enable a prover to show that all entries in a committed vector belong to a predetermined table, thereby allowing word-level binary operations to be proved directly without explicit arithmetization. A series of works [22], [23], [24], [25], [26], [45] have focused on improving the efficiency of such schemes. LogUp [22] represents the state-of-the-art in this domain, achieving an optimally constant number of field operations per lookup, as long as the table size does not exceed the lookup times. Lasso [26] overcomes this constraint by observing lookup arguments for large structured table can be break down into multiple proofs over smaller tables.

However, these constructions are less flexible than directly proving boolean circuits. Besides, they rely heavily on polynomial commitments. Although recent techniques [46] offer partial mitigation, the overall prover overhead remains substantially higher compared to our scheme.

SNARKs over Binary Tower Field. Recent works [14], [16], [47] have mainly focused on designing efficient PCS over binary fields. Leveraging a novel technique that lifts polynomials from small field to large field by packing coefficients, Binius [14] proposed a Brakedown-like PCS with Reed-Solomon codes over binary tower fields. Although Binius also proposed a new IOP construction by adapting Plonkish arithmetization to binary fields, its primary focus lies in arithmetic over $\mathbb{Z}_{2^k}$ and does not offer additional advantages for boolean circuits. Subsequently, Diamond et al. [47] generalized coefficient packing technique introduced in Binius [14] and proposed a general transformation from large-field PCS to small-field PCS. By combining the techniques from Basefold [48] and additive number-theoretic tranform (NTT) [49], they constructed an efficient PCS based on Fast Reed–Solomon IOP of Proximity (FRI) [50] over binary fields. Most recently, with a focus on enhancing prover efficiency, Blaze [16] introduced two PCS constructions based on BaseFold [48] and Brakedown [13] with Repeat-Accumulate-Accumulate code, achieving linear prover workloads and polylogarithmic verifier workloads.

These constructions are largely orthogonal to our main contribution, as our primary focus is on designing an IOP system that efficiently captures the nature of word-level binary operations prevalent in real-world applications.

Scheme	Circuit Type	$\mathcal{P}$ Time	$\mathcal{V}$ Time
Plonk	General Arithmetic	$O(N \log N) \cdot \omega(1)$	$O(\log N) \cdot \omega(1)$
Libra	Layered Arithmetic	$O(N) \cdot \omega(1)$	$O(\log N) \cdot \omega(1)$
Our's	Data-parallel Layered Boolean	$O(\frac{N}{\log N}) \cdot \omega(1)$	$O(\log N) \cdot \omega(1)$

TABLE 1: Comparison of RAM operations required by mainstream SNARKs for proving size-N circuits. Circuit Type denotes the category of circuits supported by each scheme, $\mathcal{P}$ Time denotes the prover time, and $\mathcal{V}$ Time denotes the verifier time.

2. Preliminaries

2.1. Notations

We use λ to denote the security parameter, and $\mathbb{F}$ to denote the field. We denote by $[a, b]$ the set $\{a, \ldots, b\}$ for $a, b \in \mathbb{Z}$, and we use $[n]$ to denote the set $\{1, \ldots, n\}$ for some $n \in \mathbb{N}$. We will use bold letters like $\boldsymbol{x}$ for vectors, and denote by $\boldsymbol{x}_i$ the i-th component of $\boldsymbol{x}$. We also use colon notation to denote slices of vectors. For example, $\boldsymbol{x}[i : j]$ denotes a vector constituting of $\boldsymbol{x}_i, \ldots, \boldsymbol{x}_j$.

2.2. Computation Model

In analyzing the computational complexity of algorithms, the Random Access Machine (RAM) model [51] has been widely adopted in prior works. This model operates on an infinite array of registers, each capable of storing numbers of arbitrary length. Furthermore, it assumes that basic operations on registers—such as arithmetic, binary operations and memory access—execute in constant time. Consequentially, field operations under this model are performed in constant time, as they can be simulated by constant number of arithmetic operations.

However, the RAM model includes a somewhat unrealistic assumption: it allows algorithms to access and manipulate arbitrarily large numbers in a single time step. In practice, the registers of modern CPUs store numbers with a bounded word length w. Thus field elements with different size contains in different number of register, leading to different time complexity when field size differs.

This problem arise exactly in the proof generation process. As we argued before, the bit length of field elements is chosen as $\lambda = \omega(\log N)$ for sized-N computation. This means that the field size actually grow with the computation size.

To capture this increase factor, we analyze the efficiency of algorithms with word RAM model [28], which captures the nature of programming languages and modern CPUs.

More precisely, the word RAM is an abstract machine similar to the RAM model but with finite memory and word-length. It operates on words of size up to w bits, meaning each memory can store integers up to $2^w - 1$. In this model, both arithmetic and binary operations on words are performed in constant time. For computations involving N operations, the word length w is typically set as $O(\log N)$.

Unlike the classical RAM model, field operations in the word RAM model no longer have constant-time costs. As we discussed before, the field elements have bit length $\lambda = \omega(\log N)$. Consequently, the field elements cannot fit within a constant amount of words; they requires $O(\frac{\lambda}{\log N})$ words. This leads to a cost of $O(\frac{\lambda}{\log N})$ for field addition and memory access for field elements, and $O((\frac{\lambda}{\log N})^2)$ for field multiplication.

Given the new time complexity metric, previous linear-time SNARKs for arithmetic circuits can no longer be classified as truly linear, as they involve a linear amount of field operations. In our scheme, the amount of field operations is $O(\frac{N}{\log N})$, which we justify as sub-linear time below.

One issue arise as we have only specified the asymptotic lower bound for λ, meaning it can grow arbitrarily large. For practical applications, however, it is typically suffice to select $\lambda = O(\log N \cdot \log \log N)$, leading to field operations (including memory access for field elements) costs no greater than $O((\log \log N)^2)$. Incorporating this cost, the time complexity for field operations part is $O(\frac{N}{\log N} \cdot (\log \log N)^2)$ which is sub-linear to N.

As detailed below, our proof generation process never requires more than $O(N)$ memory. In particular, it mainly needs to store $O(\frac{N}{\log N})$ field elements and additionally $O(\frac{N}{\log N})$ words comprising $O(N)$ bits in total, which allows us to fix the word length w as $O(\log N)$ and total memory size $O(2^w)$, ensuring no additional memory allocation throughout the whole process. This property allows us to always seen basic operations on words as fixed time. For simplicity, we would set the factor for the cost of field operations as a super-constant factor $\omega(1)$.

2.3. Linear-Time Encodable Linear Code

Definition 1 (Linear Code)**.** *A linear error-correcting code with message length k and codeword length n is a linear subspace $C \in \mathbb{F}^n$, such that there exists an injective mapping from message to codeword $\mathsf{Enc} : \mathbb{F}^k \to C$, which is called the encoder of the code. Any linear combination of codewords is also a codeword. The rate of the code is defined as $\frac{k}{n}$. The distance between two codewords u, v is the hamming distance denoted as $\Delta(u, v)$. The minimum distance is $d = \min_{u,v} \Delta(u, v)$. Such a code is denoted as $[n, k, d]$ linear code, and we also refer to $\frac{d}{n}$ as the relative distance of the code.*

2.4. Collision-Resistant Hash Function and Merkle Tree

Definition 2. *A commitment scheme is a tuple of algorithms* $\mathsf{Setup}(1^\lambda) \to \mathsf{ck}$, $\mathsf{Commit}(\mathsf{ck}, m, r) \to \mathsf{com}$, $\mathsf{Open}(\mathsf{ck}, \mathsf{com}, m, r) \to \{0, 1\}$ *such that:*

- ***Correctness***. *For any message* m,

$$\Pr\begin{bmatrix}\mathsf{Setup}(1^\lambda)\to \mathsf{ck}\\ \mathsf{Commit}(\mathsf{ck},m,r)\to \mathsf{com}\\ \mathsf{Open}(\mathsf{ck},\mathsf{com},m,r)\to 1\end{bmatrix}=1.$$

- ***Binding***. *For any PPT adversary* $\mathcal{A}$

$$\Pr\left[\begin{matrix}b_0=b_1\neq 0\\ \wedge\\ x_0\neq x_1\end{matrix}\middle|\begin{matrix}\mathsf{ck}\leftarrow\mathsf{Setup}(1^\lambda)\\ (\mathsf{com},m,m',r,r')\leftarrow\mathcal{A}(\mathsf{ck})\\ b_0\leftarrow\mathsf{Open}(\mathsf{ck},\mathsf{com},m,r)\\ b_1\leftarrow\mathsf{Open}(\mathsf{ck},\mathsf{com},m',r')\end{matrix}\right]\leq\mathrm{negl}(\lambda).$$

- ***Hiding***. *For any* $\mathsf{Setup}(1^\lambda)\to\mathsf{ck}$*, for all* m,m'*, the following two distributions are statistically close:*

$$\mathsf{Commit}(\mathsf{ck},m,r)\approx\mathsf{Commit}(\mathsf{ck},m',r').$$

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

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

1) $h\leftarrow\mathsf{Merkle.Commit}(m_1,\ldots,m_n)$;
2) $(m_i,\pi_i)\leftarrow\mathsf{Merkle.Open}(m,i)$;
3) $(\mathsf{acc},\mathsf{rej})\leftarrow\mathsf{Merkle.Verify}(\pi,m_i,h)$;

To achieve zero-knowledge, we requires the hash function to be hiding and we implicitly assumes for each hash function call on input x, we will append a randomness r.

2.5. Polynomial Interpolation

It is well-known that for any $f:\{0,1\}^n\to\mathbb{F}$, there is a unique multilinear polynomial $\tilde{f}:\mathbb{F}^n\to\mathbb{F}$ such that $\tilde{f}(\boldsymbol{x})=f(\boldsymbol{x})$ for all $\boldsymbol{x}\in\{0,1\}^n$. The polynomial $\tilde{f}$ is called the *multilinear extension* (MLE) of f, and can be expressed as $\tilde{f}(\boldsymbol{X})=\sum_{\boldsymbol{x}\in\{0,1\}^n}f(\boldsymbol{x})\cdot\widetilde{eq}(\boldsymbol{x},\boldsymbol{X})$, where $\widetilde{eq}(\boldsymbol{x},\boldsymbol{X}):=\prod_{i=1}^n(\boldsymbol{x}_i\boldsymbol{X}_i+(1-\boldsymbol{x}_i)(1-\boldsymbol{X}_i))$ and it can be constructed within linear time [52, Section 3.5].

2.6. Interactive Proof

Definition 3 (Interactive Argument of Knowledge)**.** *An interactive protocol* $\Pi=(\mathsf{Setup},\mathcal{I},\mathcal{P},\mathcal{V})$ *between a prover* $\mathcal{P}$ *and verifier* $\mathcal{V}$ *is an argument of knowledge for an indexed relation* $\mathcal{R}$ *with knowledge error* $\delta:\mathbb{N}\to[0,1]$ *if the following properties hold, where given a common input* $\mathbb{x}$ *and prover witness* $\mathbb{w}$*, the deterministic indexer outputs and the output of the verifier is denoted by the random variable* $\langle\mathcal{P}(\mathsf{pk},\mathbb{x},\mathbb{w}),\mathcal{V}(\mathsf{vk},\mathbb{x})\rangle$*:*

- ***Perfect Completeness:*** *for all* $(\mathbb{i},\mathbb{x},\mathbb{w})\in\mathcal{R}$*,*

$$\Pr\left[\langle\mathcal{P}(\mathsf{pk},\mathbb{x},\mathbb{w}),\mathcal{V}(\mathsf{vk},\mathbb{x})\rangle=1\,\middle|\,\begin{matrix}\mathsf{pp}\leftarrow\mathsf{Setup}(1^\lambda)\\ (\mathsf{vk},\mathsf{pk})\leftarrow\mathcal{I}(\mathsf{pp},\mathbb{i})\end{matrix}\right]=1.$$

- ***Knowledge Soundness:*** *There exists a polynomial* $\mathrm{poly}(\cdot)$ *and a PPT oracle machine* $\mathcal{E}$ *called the extractor such that given oracle access to any pair of PPT adversarial prover algorithm* $(\mathcal{A}_1,\mathcal{A}_2)$*, the following holds:*

$$\Pr\left[\begin{matrix}\langle\mathcal{A}_2(\mathbb{i},\mathbb{x},\mathsf{st}),\mathcal{V}(\mathsf{vk},\mathbb{x})\rangle=1\\ \wedge\\ (\mathbb{i},\mathbb{x},\mathbb{w})\notin\mathcal{R}\end{matrix}\middle|\begin{matrix}\mathsf{pp}\leftarrow\mathsf{Setup}(1^\lambda)\\ (\mathbb{i},\mathbb{x},\mathsf{st})\leftarrow\mathcal{A}_1(\mathsf{pp})\\ (\mathsf{vk},\mathsf{pk})\leftarrow\mathcal{I}(\mathsf{pp},\mathbb{i})\\ \mathbb{w}\leftarrow\mathcal{E}^{\mathcal{A}_1,\mathcal{A}_2}(\mathsf{pp},\mathbb{i},\mathbb{x})\end{matrix}\right]\leq\delta(|\mathbb{i}|+|\mathbb{x}|).$$

An interactive protocol is knowledge sound if the knowledge error δ *is negligible in* λ*.*

- ***Public coin:*** *An interactive protocol is public-coin if* $\mathcal{V}$*'s messages are chosen uniformly at random.*

If an interactive argument of knowledge protocol is public-coin, then it can be made non-interactive by the Fiat-Shamir transformation [53]. If the scheme further satisfies the following property:

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

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

2.7. Polynomial Commitment Scheme

A polynomial commitment scheme consists of three algorithms:

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

Definition 4 ((Multivariate) Polynomial Commitment)**.** *A polynomial commitment scheme has the following properties:*

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

$$\Pr\begin{bmatrix}\mathsf{PC.Commit}(\phi)\to\mathcal{R}\\ \mathsf{PC.Prove}(\phi,\boldsymbol{x},\mathcal{R})\to\boldsymbol{x},y,\pi\\ y=\phi(\boldsymbol{x})\\ \mathsf{PC.VerifyEval}(\pi,\boldsymbol{x},y,\mathcal{R})\to\mathsf{acc}\end{bmatrix}=1$$

- ***Knowledge Soundness***. *For any PPT adversary* $\mathcal{A}$ *with* $\mathsf{PC.Commit}^*$*,* $\mathsf{PC.Prove}^*$*, there exits a PPT extractor* $\mathcal{E}$ *such that*

$$\Pr\left[\begin{matrix}\phi^*\leftarrow\mathcal{E}(\mathcal{R}^*,\boldsymbol{x},\\ \pi^*,y^*)\\ \wedge\\ y^*\neq\phi^*(\boldsymbol{x})\end{matrix}\middle|\begin{matrix}\mathsf{PC.Commit}^*(\phi^*)\to\mathcal{R}^*\\ \mathsf{PC.Prove}^*(\phi^*,\boldsymbol{x},\mathcal{R}^*)\to\boldsymbol{x},y^*,\pi^*\\ \mathsf{PC.VerifyEval}(\pi^*,\boldsymbol{x},y^*,\mathcal{R}^*)\to\mathsf{acc}\end{matrix}\right]\leq\mathrm{negl}(\lambda).$$

3. Efficient GKR Protocol for Boolean Circuits

3.1. Review of SumCheck Protocol

The sumcheck protocol underpins many fast prover SNARKs [10], [11], [31], [35]. Proposed in [29],the protocol enables the prover $\mathcal{P}$ to convince the verifier $\mathcal{V}$ that the sum of a polynomial $f : \mathbb{F}^n \to \mathbb{F}$ on the boolean hypercube equals to the purported value v, i.e.

$$v = \sum_{\boldsymbol{x} \in \{0,1\}^n} f(x_1, \ldots, x_n).$$

Naïvely computing this sum for $\mathcal{V}$ would entail exponential time in n due to the 2^n combinations of $x_1, \ldots, x_n$. Instead, the protocol allows the verifier $\mathcal{V}$ to delegate the computation to a prover $\mathcal{P}$, who can convince $\mathcal{V}$ that v is the correct sum through multiple rounds of interaction. The detailed description of the sumcheck protocol is presented in Protocol 1. The proof size of the sumcheck protocol is $O(dn)$, where d is the variable-degree of f, as $\mathcal{P}$ sends a univariate polynomial of degree d in each round. The verifier time of the protocol is $O(dn)$. With efficient implementations [30], [54], the prover's runtime for multilinear polynomial consists of linear field operations. The sumcheck protocol is complete and sound with soundness error $\epsilon = \frac{dn}{|\mathbb{F}|}$.

3.2. Reveiw of GKR Protocol for Data-parallel Circuits

The GKR (Goldwasser-Kalai-Rothblum) protocol [35] is an interactive proof system designed to efficiently verify the correctness of computations over layered arithmetic circuits. It allows the prover to convince the verifier that the outputs of a circuit is correct, without requiring the verifier to re-execute the entire computation. The protocol is particularly well-suited for computations that can be represented as circuits with multiple layers of gates, such as addition and multiplication.

As our main focus is data-parallel circuit, here we first present the formulation of GKR protocol specifically designed for data-parallel circuits from previous works [30], [55].

Let C be a layered circuit with depth d over a finite field. We assume layer-i of the circuit have width $|C_i| = 2^{s_i}$. Let B denotes the number of identical sub-circuits contained in C, and $L_i := \frac{|C_i|}{B} = 2^{\ell_i}$ to represent the width of layer-i of each sub-circuits.

The GKR protocol operates iteratively from the output layer, denoted as layer 0, back to the input layer, denoted as layer d. For each layer of the circuit, the GKR protocol uses *multilinear extensions* to represent the gate values and the wiring of the circuit.

Let $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ denote the gate outputs in layer i. For data-parallel circuits, the index $\boldsymbol{g} \in \{0,1\}^{s_i}$ can be broken down into two pieces $(\boldsymbol{p}, \boldsymbol{b})$, where $\boldsymbol{p} \in \{0,1\}^{\ell_i}$ is the index of gates in each sub-circuits and $\boldsymbol{b} \in \{0,1\}^{\log B}$ is the index of the sub-circuits. The multilinear extension $\tilde{V}_i$ is computed to capture the relationships between two adjacent layers. More specifically, *wiring predicates* express how gates in layer $i-1$ depend on those in layer i. For a gate $\boldsymbol{p}$ in layer $i-1$ that takes inputs from gates $\boldsymbol{x}$ and $\boldsymbol{y}$ in layer i, the wiring predicates $\tilde{add}_{i-1}$ ($\tilde{mult}_{i-1}$) evaluates to 1 if this gate performs addition (multiplication) operation, and 0 otherwise. Not that these predicates ignore the index of sub-circuits as all sub-circuits are identical. Thus, the value of a gate $\tilde{\boldsymbol{g}} := (\tilde{\boldsymbol{p}}, \tilde{\boldsymbol{b}})$ in layer $i-1$ can be expressed as:

$$V_{i-1}(\tilde{\boldsymbol{p}}, \tilde{\boldsymbol{b}}) = \sum_{\boldsymbol{x},\boldsymbol{y},\boldsymbol{b}} \tilde{eq}(\boldsymbol{b}, \tilde{\boldsymbol{b}}) \Big[\tilde{mult}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) V_i(\boldsymbol{x}, \boldsymbol{b}) V_i(\boldsymbol{y}, \boldsymbol{b}) + \tilde{add}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y})(V_i(\boldsymbol{x}, \boldsymbol{b}) + V_i(\boldsymbol{y}, \boldsymbol{b})) \Big], \tag{1}$$

where the summation is taken over $\boldsymbol{b} \in \{0,1\}^{\log B}$ and $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^{\ell_i}$.

To verify the correctness of the circuit's evaluation, the sumcheck protocol is applied layer by layer. Using each layer's representation, the verifier checks that the values in layer $i-1$ are computed correctly based on the values in layer i.

At the end of each round's sumcheck, the claim from previous layer is distill into two claims of current layer $\tilde{V}_i(\boldsymbol{p}, \boldsymbol{b})$ and $\tilde{V}_i(\boldsymbol{p}', \boldsymbol{b}')$. Utilizing standard techniques, $\mathcal{V}$ could consolidate these two claims into one using additional random challenges. Then the remaining proof can proceeds to an layer above until reaching the input layer, where the final claim can be verified by evaluating $\tilde{V}_d$.

3.3. Arithmetization of Data-parallel Boolean Circuits on GKR

The original GKR protocol was designed for arithmetic circuits, focusing specifically on circuits composed solely of field operations. However, in our context, we aim to prove the satisfiability of boolean circuits. Thus, the representation requires revision to capture the binary operations and to enable us to take advantage of the binary nature of the circuits.

In boolean circuits, addition and multiplication gates are replaced by XOR and AND gates, respectively. For simplicity, we will abuse the notation $\tilde{add}_{i-1}$ and $\tilde{mult}_{i-1}$ to refer to the wiring predicates of XOR and AND gates. The representation of multiplication gates requires no modification, as $V_i(\boldsymbol{x})V_i(\boldsymbol{y})$ matches the AND operation's result for binary values. For addition gates, the representation should be modified as $V_i(\boldsymbol{x}) + V_i(\boldsymbol{y}) - 2V_i(\boldsymbol{x})V_i(\boldsymbol{y})$ catering for the result of XOR operations.[1]

For each layer's output polynomial, more delicate adaptations are needed to enable us to take advantages of the binary nature of circuits. The traditional representation with multilinear extension in Equation 1 would require $\ell + \log B$ variables to represent all outputs. Here we instead utilize a more generalized version of multilinear extension, allowing one variable, say b, to become degree of $B-1$, leading

1. Here we omitted the index for sub-circuits for simplicity, but the adaption follow similarly for data-parallel setting.

PROTOCOL 1. *SumCheck Protocol*

$\mathcal{P}$ claims $v = \sum_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x})$ to $\mathcal{V}$. Let $f_0 = v$.

- In the k-th round where $1 \le k \le n-1$:
  - $\mathcal{P}$ sends a univariate polynomial $f_k(X_k) = \sum_{\boldsymbol{x}\in\{0,1\}^{n-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x})$ to $\mathcal{V}$.
  - $\mathcal{V}$ checks $f_{k-1} = f_k(0) + f_k(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}$ and sets $f_k = f_k(r_k)$.
- In the n-th round:
  - $\mathcal{P}$ sends a univariate polynomial $f_n(X_n) = f(r_1, \ldots, r_{n-1}, X_n)$ to $\mathcal{V}$.
  - $\mathcal{V}$ checks $f_{n-1} = f_n(0) + f_n(1)$. If the check passes, $\mathcal{V}$ generates a random challenge $r_n \in \mathbb{F}$, and accepts iff $f_n(r_n) = f(r_1, \ldots, r_n)$ using one oracle call to f.

Figure 1: The Vanilla SumCheck Protocol

to a more compact representation. Intuitively speaking, this representation allows us to deal multiple bits in one shot, instead of just a single bit with traditional representation, which leaves more room for precomputation. Formally we extend $f : \{0,1\}^\ell \times [B] \to \mathbb{F}$, and the extension is defined as:

$$\tilde{f}(\boldsymbol{x}, b) = \sum_{\boldsymbol{y}\in\{0,1\}^\ell} \sum_{t\in[B]} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{y}) \cdot L_t(b) \cdot f(\boldsymbol{y}, t), \quad (2)$$

where the $L_t(b) = \prod_{i\in[B], i\neq t} \frac{(i-b)}{(i-t)}$ denotes the t-th Lagrange polynomial over the domain $[B]$.[2] In the original GKR protocol, we use $\boldsymbol{b} \in \{0,1\}^{\log B}$ to denote the $\boldsymbol{b}$-th sub-circuits. For our modification, we would instead use $b \in [B]$ to denote the index of sub-circuits. Furthermore, to facilitate later optimizations, we assume the evaluations of the binary function $f(b, \boldsymbol{x})$ is provided as a whole B-bit integer for each $\boldsymbol{x} \in \{0,1\}^\ell$, rather than as B separated bits. This assumption is made without loss of generality, as batched computation for data-parallel Boolean circuits also adopts this formulation.

Then incorporating all revision above, we present the modified representation between different layers as follows:

$$\tilde{V}_{i-1}(\tilde{\boldsymbol{p}}, \tilde{b}) = \sum_{b,\boldsymbol{x},\boldsymbol{y}} L_b(\tilde{b}) \Big[\tilde{mult}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \tilde{V}_i(\boldsymbol{x}, b) \tilde{V}_i(\boldsymbol{y}, b) + \tilde{add}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \Big(\tilde{V}_i(\boldsymbol{x}, b) + \tilde{V}_i(\boldsymbol{y}, b) - 2\tilde{V}_i(\boldsymbol{x}, b) \tilde{V}_i(\boldsymbol{y}, b) \Big) \Big], \quad (3)$$

where the summation is taken over $b \in [B]$ and $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^{\ell_i}$.

3.4. Optimized Sumcheck Protocol for Boolean Circuits

We have already explained how to arithmetize data-parallel boolean circuits for the GKR protocol with modified multilinear extensions and gate evaluations across each layer. In the following, we will show how this reformulation allows us to prove the correctness of computations in boolean circuits with better efficiency.

For simplicity, in the following, we consider the circuits with AND gate only, i.e.:

$$\tilde{V}_{i-1}(\tilde{\boldsymbol{p}}, \tilde{b}) = \sum_{b,\boldsymbol{x},\boldsymbol{y}} L_b(\tilde{b}) \tilde{mult}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \tilde{V}_i(\boldsymbol{x}, b) \tilde{V}_i(\boldsymbol{y}, b), \quad (4)$$

where $\tilde{b}$ and $\tilde{\boldsymbol{p}}$ are random queries from the previous sumcheck protocol, and the summation is taken over $b \in [B]$ and $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^{\ell_i}$.

Recall that in the word RAM model, the word size is typically set to $O(\log N)$ for computations of size N. Accordingly, we assume the boolean circuits to be proven operate on words of size $B = \frac{\log N}{3}$. This assumption is made without loss of generality, as any word-level binary operation in the word RAM model can be simulated using only a constant number of words within the circuits, which would increases the prover's overheads by a constant factor.

Now, we delve into optimizing the sumcheck protocol, which is a critical component in reducing the prover's workload.

Let the prover performs the sumcheck on variable b first, where the prover computes a univariate polynomial $f(b)$, such that for any $t \in [B]$:

$$f(t) = \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^{\ell_i}} L_t(\tilde{b}) \tilde{mult}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \tilde{V}_i(\boldsymbol{x}, t) \tilde{V}_i(\boldsymbol{y}, t). \quad (5)$$

Since $L_t(\tilde{b})$ is fully determined by the random challenge $\tilde{b}$ and $\tilde{mult}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y})$ is specified by the random challenge $\tilde{\boldsymbol{p}}$ and the predicate $\tilde{mult}_{i-1}(\boldsymbol{p}, \boldsymbol{x}, \boldsymbol{y})$ as

$$\tilde{mult}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) = \sum_{\boldsymbol{p}\in\{0,1\}^{\ell_{i-1}}} \widetilde{eq}(\boldsymbol{p}, \tilde{\boldsymbol{p}}) \tilde{mult}_{i-1}(\boldsymbol{p}, \boldsymbol{x}, \boldsymbol{y}),$$

the remaining job for computing $f(b)$ is to decide the factor $\tilde{V}_i(\boldsymbol{x}, b)\tilde{V}_i(\boldsymbol{y}, b)$ for each $\boldsymbol{x}$ and $\boldsymbol{y}$.

Recall that $\tilde{V}_i(\boldsymbol{x}, b)$ (or equivalently $\tilde{V}_i(\boldsymbol{y}, b)$) refers to the binary output of gate $\boldsymbol{x}$ at layer i within the b-th sub-circuit. Additionally, for each $\boldsymbol{x}$, B evaluations of all sub-circuits are represented as a B-bit integer. Since there are 2^B distinct B-bit integers, a key observation is that the total

2. The Lagrange polynomial over binary field can be similarly derived by selecting B distinct values to replace the set $[B]$.

number of unique combinations of $\tilde{V}_i(\boldsymbol{x}, b)\tilde{V}_i(\boldsymbol{y}, b)$ is 2^{2B}. However, the grand summation in Equation 5 has a size of $2^{2\ell_i}$, which is significantly larger than 2^{2B}.

Let $g(b) = \sum_{t\in[B]} L_t(\tilde{b}) \cdot L_t(b)$, and $g_{\boldsymbol{x},\boldsymbol{y}}(b) := g(b) \cdot \tilde{V}_i(\boldsymbol{x}, b) \cdot \tilde{V}_i(\boldsymbol{y}, b)$. Then each additive factor in Equation 5 can be represented as $\tilde{mult}_{i-1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y})g_{\boldsymbol{x},\boldsymbol{y}}(b)$. Once all possible combinations of $g_{\boldsymbol{x},\boldsymbol{y}}(b)$ are computed and stored in a bookkeeping table, it can be efficiently retrieved by a single table looking up. The pre-computation procedure is specified in Algorithm 1.

Algorithm 1 $\mathbf{A}_g \leftarrow \mathsf{TablePrecomputation}(\tilde{b}, B)$

Input: : $\tilde{b}$: random query point for b, B: number of parallel bits

Output: Precomputed table of polynomials for all combinations of $\tilde{V}_i(\boldsymbol{x}, b)$ and $\tilde{V}_i(\boldsymbol{y}, b)$

1: Set $g(b) = [L_t(\tilde{b})]_{t\in[B]}$
2: $\forall x, y \in [2^B]$, set $\mathbf{A}_g[x \cdot 2^B + y] = 0$.
3: **for** $x = 0$ to $2^B - 1$ **do**
4: **for** $y = 0$ to $2^B - 1$ **do**
5: $vx \leftarrow \mathsf{bin}(x)$
6: $vy \leftarrow \mathsf{bin}(y)$
7: $g_{\boldsymbol{x},\boldsymbol{y}}(b) \leftarrow g(b) \cdot vx \cdot vy$
8: $\mathbf{A}_g[x \cdot 2^B + y] = g_{\boldsymbol{x},\boldsymbol{y}}(b)$
9: **end for**
10: **end for**
11: **return** $\mathbf{A}_g$

We present the precomputation pseudocode in section B.

Theorem 1. *The bookkeeping table $\mathbf{A}_g$ can be evaluated with $O(B^2 \cdot 2^{2B}) \cdot \omega(1)$ RAM operations using Algorithm 1.*

Proof. The computation complexity for computing $g(b)$ with the naïve algorithm is dominated with $O(B^2)$ field operations, which leads to $O(B^2) \cdot \omega(1)$ RAM operations.

For the iteration step, the most computationally expensive phase occurs at step 7, where three univariate polynomials of degree $B-1$ are multiplied together. This multiplication requires $O(B^2)$ field operations with plain algorithm, which is equivalent to $O(B^2)\cdot\omega(1)$ RAM operations.[3] Thus, the total RAM complexity for computing the table $\mathbf{A}_g$ is $O(B^2 \cdot 2^{2B}) \cdot \omega(1)$. □

Consider a practical scenario where $B = 8$ and the field size satisfies $\log|\mathbb{F}| = 128$: the preprocessed table contains $2^{2B} = 2^{16}$ elements, each requiring 2 polynomial multiplications. Consequently, a total of 2^{17} polynomial multiplications is performed. The memory required for the table is calculated as $2^{16} \times 128 \times 3 \times B$ bits. This is because there are 2^{16} polynomials, each with a degree of $3B$, and each coefficient occupies 128 bits. The total memory usage is approximately 24 MB, which fits comfortably within the CPU's L3 cache, enabling efficient and fast data access during computation.

With the precomputed table, the prover can now compute the polynomial as:

$$\begin{aligned} f(b) &= \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^{\ell_i}} \tilde{mult}_i(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y})g_{\boldsymbol{x},\boldsymbol{y}}(b) \\ &= \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^{\ell_i}} \sum_{\boldsymbol{p}\in\{0,1\}^{\ell_{i-1}}} \tilde{mult}_i(\boldsymbol{p}, \boldsymbol{x}, \boldsymbol{y})\widetilde{eq}(\boldsymbol{p}, \tilde{\boldsymbol{p}})g_{\boldsymbol{x},\boldsymbol{y}}(b) \end{aligned}$$

To present the algorithm for computing $f(b)$ with sublinear RAM operations, we first provide a straw-man in Algorithm 2 that helps introduce the intuition behind our solution. Without ambiguity, as binary vector $\boldsymbol{p}$ uniquely defines an integer, we abuse the notation and see $\boldsymbol{p}$ as the corresponding integer representation in the algorithm description. Recall that the outputs of the B sub-circuits outputs for the same gate $\boldsymbol{p}$ are packed into a single B-bit integer. Thus, we denote the output word of the $\boldsymbol{p}$-th gate across all sub-circuits in layer i as $V_i.output[\boldsymbol{p}]$. Additionally, $V_i.gates[\boldsymbol{p}]$ is a 3-tuple contains the type, the index of left input and the index of right input of $\boldsymbol{p}$-th gate in layer i.

Algorithm 2 $f(b) \leftarrow \mathsf{NaïveUniEval}(\mathbf{A}_g, V_i, V_{i-1}, \tilde{\boldsymbol{p}})$

Input: Precomputed table $\mathbf{A}_g$ containing $g_{\boldsymbol{x},\boldsymbol{y}}(b)$ polynomials and accumulators, Current circuit layer V_i, Previous circuit layer V_{i-1}, Random query point $\tilde{\boldsymbol{p}}$

Output: The polynomial $f(b)$ of degree $3(B-1)$

1: Initialize an empty polynomial $f(b)$
2: Initialize the evaluation table $\mathbf{A}_{eq}$ for $\widetilde{eq}(\boldsymbol{p}, \tilde{\boldsymbol{p}})$.
3: **for** each gate $\boldsymbol{p}$ in $V_{i-1}.gates$ **do**
4: $x \leftarrow V_i.output[V_{i-1}.gates[\boldsymbol{p}][1]]$
5: $y \leftarrow V_i.output[V_{i-1}.gates[\boldsymbol{p}][2]]$
6: $f(b) \leftarrow f(b) + \mathbf{A}_{eq}[\boldsymbol{p}] \times \mathbf{A}_g[x \cdot 2^B + y]$
7: **end for**
8: **return** res

Theorem 2. *The univariate polynomial $f(b)$ can be evaluated with $O(|C_i|) \cdot \omega(1)$ RAM operations using Algorithm 2.*

Proof. Computing the table $\mathbf{A}_{eq}$ primarily involves $O(\frac{|C_i|}{B})$ memory reads and writes for field elements and same amount of field operations, resulting in $O(\frac{|C_i|}{B}) \cdot \omega(1)$ RAM operations. In each iteration step, retrieving x and y requires $O(1)$ read into memory for words and thus requires $O(1)$ RAM operations. Additionally, in step 6, multiplying one field element with a polynomial of degree $3(B-1)$ requires $O(B) \cdot \omega(1)$ RAM operations. As the iteration repeated $\frac{|C_i|}{B}$ times, the total cost is $O(B) \cdot \omega(1) \cdot \frac{|C_i|}{B} = O(|C_i|) \cdot \omega(1)$ RAM operations. □

The primary issue with Algorithm 2 is that the actual number of possible polynomials $g_{\boldsymbol{x},\boldsymbol{y}}$ is 2^{2B}, which can be much fewer than the round of iterations with proper choice of B. Given that polynomial operations are much

3. Algorithm for univariate polynomial interpolation with better asymptotical performance exists (e.g. ECFFT [56]). For our usage, however, the plain algorithm suffices as B would typically be small enough that plain algorithm could outpace the optimized ones. Additionally, the overhead of this part of computation is in itself minor compared with other computations which we introduce below.

more expensive than field operations, redundant computation occurs when the same polynomial is revisited by different multipliers. With this observation, we could accumulate the multiplier of each polynomial using only one field operation, thereby saving repeated scalar-polynomial multiplications. The full process is presented in Algorithm 3.

Algorithm 3 $f(b) \leftarrow \mathsf{UniEval}(\mathbf{A}_g, V_i, V_{i-1}, \tilde{\boldsymbol{p}})$

Input: Precomputed table $\mathbf{A}_g$ containing $g_{\boldsymbol{x},\boldsymbol{y}}(b)$ polynomials and accumulators, Current circuit layer V_i, Previous circuit layer V_{i-1}, Random query point $\tilde{\boldsymbol{p}}$

Output: The polynomial $f(b)$ of degree $3(B-1)$

1: Initialize an empty polynomial $f(b)$
2: Initialize the evaluation table $\mathbf{A}_{eq}$ for $\widetilde{eq}(\boldsymbol{p}, \tilde{\boldsymbol{p}})$
3: Initialize the accumulator $\mathbf{accu}[i] = 0, \forall i \in [2^{2B}]$.
4: **for** each gate $\boldsymbol{p}$ in $V_{i-1}.gates$ **do**
5: $\quad x \leftarrow V_i.output[V_{i+1}.gates[\boldsymbol{p}][1]]$
6: $\quad y \leftarrow V_i.output[V_{i+1}.gates[\boldsymbol{p}][2]]$
7: $\quad \mathbf{accu}[x \cdot 2^B + y] \leftarrow \mathbf{accu}[x \cdot 2^B + y] + \mathbf{A}_{eq}[\boldsymbol{p}]$ ▷ Accumulate multiplier for the polynomial $g_{\boldsymbol{x},\boldsymbol{y}}$
8: **end for**
9: **for** $i = 0$ to $2^{2B} - 1$ **do**
10: $\quad f(b) \leftarrow f(b) + \mathbf{A}_g[i] \times \mathbf{accu}[i]$
11: **end for**
12: **return** $f(b)$

Theorem 3. *The univariate polynomial $f(b)$ can be evaluated with $O(\frac{|C_i|}{B} + B \cdot 2^{2B}) \cdot \omega(1)$ RAM operations using Algorithm 3.*

Proof. Computing the table $\mathbf{A}_{eq}$ primarily involves $O(\frac{|C_i|}{B})$ memory reads and writes for field elements and same amount of field operations, resulting in $O(\frac{|C_i|}{B}) \cdot \omega(1)$ RAM operations. In first iteration, retrieving each x and y requires $O(1)$ RAM operations. Additionally, step 7 performs a single field operations, resulting in $\omega(1)$ RAM operations. The iteration repeat $\frac{|C_i|}{B}$ times, thus the total cost for first iteration is $O(\frac{|C_i|}{B}) \cdot \omega(1)$. In second iteration, step 10 multiplies a field element to a degree $3(B-1)$ polynomial and then add it to the accumulated polynomial, which necessitates $O(B)$ field operations and memory reads for field elements. Since this step is repeated 2^{2B} times, the total RAM complexity for second iteration is $O(B \cdot 2^{2B}) \cdot \omega(1)$. Integrating all cost leads to the result. □

We present the pseudocode in section C. The optimized process now takes $\frac{|C_i|}{B}$ field additions and 2^{2B} polynomial additions, which overall amounts to $\frac{|C_i|}{B}$ field additions and $2^{2B} * (3(B-1))$ field multiplications/additions. If $B = 8$, then the second cost is 1376256, and $\log_2(\text{cost}) \approx 20.39$.

3.5. Putting Everything Together

After using the optimized algorithm to compute the univariate polynomial $f(b)$, the prover $\mathcal{P}$ sends $f(b)$ the verifier. $\mathcal{V}$ could check

$$\sum_{b \in [B]} f(b) = \tilde{V}_{i+1}(\tilde{\boldsymbol{p}}, \tilde{b}),$$

and sends a random challenge $\tilde{b}'$ to the prover. Then both parties proceed with the following:

$$f(\tilde{b}') = \sum_{\boldsymbol{x},\boldsymbol{y} \in \{0,1\}^{\ell_i}} g(\tilde{b}') \tilde{mult}_i(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \tilde{V}_i(\boldsymbol{x}, \tilde{b}') \tilde{V}_i(\boldsymbol{y}, \tilde{b}') \quad (6)$$

The remaining steps follow the standard process of the GKR protocol [10], [35], which takes $O(\frac{|C_i|}{B})$ field operations and memory reading for field elements, which leads to a total $O(\frac{|C_i|}{B}) \cdot \omega(1)$ RAM operations. We present the full description of our optimized sumcheck in Protocol 2. Combining with Theorem 3, we formalize the cost for each layer's sumcheck in the following theorem.

Theorem 4. *The prover's work in optimized sumcheck described in Protocol 2 is $O(\frac{|C_i|}{B} + B \cdot 2^{2B}) \cdot \omega(1)$ RAM operations.*

Remark 2. *The above construction is not zero-knowledge, but this can be achieved by using the standard techniques. Therefore, we omit the details here for simplicity.*

Theorem 5. *Let $C : \{0,1\}^n \to \{0,1\}^k$ be a depth-d data-parallel boolean circuit. Protocol 3 is an interactive proof for the function computed by C with soundness error $O\big(d \cdot (B + \log(|C|/B))/|\mathbb{F}|\big)$. It uses $O(d \log |C|)$ rounds of interaction and running time of the prover $\mathcal{P}$ $O\left(d \cdot \left(\frac{|C|}{B} + B^2 \cdot 2^{2B}\right)\right) \cdot \omega(1)$ RAM operations.*

Proof. Completeness of the protocol is self-evident. Thus we prove the theorem as follows:

For Security. If the prover begins the protocol with a false claim regarding the output values in out, then the verifier can only be convinced to accept if, in at least one round j of the interactive proof the prover either sends a univariate polynomial $\hat{f} \neq f$ during the sumcheck for variate b, or cheats in the subsequent multivariate sumcheck. In the first case, the probability of cheating equals to the probability that $\hat{f}(\tilde{b}') = f(\tilde{b}')$, which is at most $3(B-1)/|\mathbb{F}|$ given that $\deg(f) \leq 3(B-1)$. In the second case, the probability of cheating is bounded by $\log(|C_i|/B)/|\mathbb{F}| \leq \log(|C|/B)/|\mathbb{F}|$. Applying the union bound over these two scenarios, the probability of of prover successfully cheating in any round j of optimized GKR protocol is at most $O\big((B + \log(|C|/B))/|\mathbb{F}|\big)$. Thus, the total soundness error across all rounds is $O\big(d \cdot (B + \log(|C|/B))/|\mathbb{F}|\big)$.

We further claim that as long as the prover pass all checks in Protocol 3 and the output polynomial consists entirely of binary values, the verifier can be assured that the input polynomial is also binary with overwhelming probability. If the prover performs the calculation dishonestly, the probability of passing the checks is negligible, so we focus on the situation where the calculation is done honestly. It is easy to see that the neither AND gate nor XOR gate can produce a binary output if one of the inputs is non-binary. Therefore, if there is at least one non-binary input

PROTOCOL 2. *Optimized SumCheck*

The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ collectively check that $\tilde{V}_{i+1}(\tilde{\boldsymbol{p}}, \tilde{b}) = \sum_{\boldsymbol{x},\boldsymbol{y},b} \tilde{mult}_i(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) L_b(\tilde{b}) \tilde{V}_i(b, \boldsymbol{x}) \tilde{V}_i(b, \boldsymbol{y})$.

1) $\mathcal{P}$ computes the univariate polynomial $f(b)$ as $f(b) = \sum_{\boldsymbol{x},\boldsymbol{y}} \tilde{mult}_i(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) g(b) \tilde{V}_i(\boldsymbol{x}, b) \tilde{V}_i(\boldsymbol{y}, b)$, where $g(b) = \sum_{t\in[B]} L_t(b) \cdot L_t(\tilde{b})$, using Algorithm 3, and sends it to $\mathcal{V}$.
2) $\mathcal{V}$ checks that $\tilde{V}_{i+1}(\tilde{\boldsymbol{p}}, \tilde{b}) = \sum_b f(b)$. Then $\mathcal{V}$ uniformly selects $\tilde{b}'$ and sends it to $\mathcal{P}$.
3) $\mathcal{P}$ and $\mathcal{V}$ check $f(\tilde{b}') = \sum_{\boldsymbol{x},\boldsymbol{y}} g(\tilde{b}') \tilde{mult}_i(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \tilde{V}_i(\boldsymbol{x}, \tilde{b}') \tilde{V}_i(\boldsymbol{y}, \tilde{b}')$, using Algorithm 9.

Figure 2: The Optimized Sumcheck Protocol

PROTOCOL 3. *Optimized GKR*

1) On a layered binary circuit C with d layers and input in, the prover $\mathcal{P}$ sends the output of the circuit out to the verifier $\mathcal{V}$.
2) $\mathcal{V}$ checks out has all binary outputs, and defines $\tilde{V}_0(p, b)$ as the extension of out. $\mathcal{V}$ evaluates it at a random point $\tilde{V}_0(\tilde{\boldsymbol{p}}^{(0)}, \tilde{b}^{(0)})$ and sends $\tilde{\boldsymbol{p}}^{(0)}, \tilde{b}^{(0)}$ to $\mathcal{P}$.
3) $\mathcal{P}$ and $\mathcal{V}$ execute the optimized sumcheck protocol presented in Protocol 2 on

$$\tilde{V}_0(\tilde{\boldsymbol{p}}^{(0)}, \tilde{b}^{(0)}) = \sum_{b,\boldsymbol{x},\boldsymbol{y}} L_b(\tilde{b}^{(0)}) \left[\tilde{mult}_1(\tilde{\boldsymbol{p}}^{(0)}, \boldsymbol{x}, \boldsymbol{y}) \tilde{V}_1(\boldsymbol{x}, b) \tilde{V}_1(\boldsymbol{y}, b) + \tilde{add}_1(\tilde{\boldsymbol{p}}^{(0)}, \boldsymbol{x}, \boldsymbol{y}) \left(\tilde{V}_1(\boldsymbol{x}, b) + \tilde{V}_1(\boldsymbol{y}, b) - 2\tilde{V}_1(\boldsymbol{x}, b) \tilde{V}_1(\boldsymbol{y}, b) \right) \right]$$

At the end of the protocol, $\mathcal{V}$ receives $\tilde{V}_1(\tilde{\boldsymbol{p}}_0^{(1)}, \tilde{b}^{(1)})$ and $\tilde{V}_1(\tilde{\boldsymbol{p}}_1^{(1)}, \tilde{b}^{(1)})$. $\mathcal{V}$ computes $\tilde{mult}_1(\tilde{\boldsymbol{p}}^{(0)}, \tilde{\boldsymbol{p}}_0^{(1)}, \tilde{\boldsymbol{p}}_1^{(1)}), \tilde{add}_1(\tilde{\boldsymbol{p}}^{(0)}, \tilde{\boldsymbol{p}}_0^{(1)}, \tilde{\boldsymbol{p}}_1^{(1)})$ and checks that

$$\tilde{mult}_1(\tilde{\boldsymbol{p}}^{(0)}, \tilde{\boldsymbol{p}}_0^{(1)}, \tilde{\boldsymbol{p}}_1^{(1)}) \tilde{V}_1(\tilde{\boldsymbol{p}}_0^{(1)}, \tilde{b}^{(1)}) \tilde{V}_1(\tilde{\boldsymbol{p}}_1^{(1)}, \tilde{b}^{(1)}) + \tilde{add}_1(\tilde{\boldsymbol{p}}^{(0)}, \tilde{\boldsymbol{p}}_0^{(1)}, \tilde{\boldsymbol{p}}_1^{(1)}) \left(\tilde{V}_1(\tilde{\boldsymbol{p}}_0^{(1)}, \tilde{b}^{(1)}) + \tilde{V}_1(\tilde{\boldsymbol{p}}_1^{(1)}, \tilde{b}^{(1)}) - 2\tilde{V}_1(\tilde{\boldsymbol{p}}_0^{(1)}, \tilde{b}^{(1)}) \tilde{V}_1(\tilde{\boldsymbol{p}}_1^{(1)}, \tilde{b}^{(1)}) \right)$$

equals to the last message of the sumcheck.
4) For layer $i = 1, \ldots, d-1$:
   a) $\mathcal{V}$ randomly selects $\alpha^{(i)}, \beta^{(i)} \in \mathbb{F}$ and sends them to $\mathcal{P}$.
   b) Let $Mult_{i+1}(\boldsymbol{x}, \boldsymbol{y}) = \alpha^{(i)} \tilde{mult}_{i+1}(\boldsymbol{p}_0^{(i)}, \boldsymbol{x}, \boldsymbol{y}) + \beta^{(i)} \tilde{mult}_{i+1}(\boldsymbol{p}_1^{(i)}, \boldsymbol{x}, \boldsymbol{y})$, and $Add_{i+1}(\boldsymbol{x}, \boldsymbol{y}) = \alpha^{(i)} \tilde{add}_{i+1}(\boldsymbol{p}_0^{(i)}, \boldsymbol{x}, \boldsymbol{y}) + \beta^{(i)} \tilde{add}_{i+1}(\boldsymbol{p}_1^{(i)}, \boldsymbol{x}, \boldsymbol{y})$. $\mathcal{P}$ and $\mathcal{V}$ run the optimized sumcheck on

$$\alpha^{(i)} \tilde{V}_i(\tilde{\boldsymbol{p}}_0^{(i)}, \tilde{b}^{(i)}) + \beta^{(i)} \tilde{V}_i(\tilde{\boldsymbol{p}}_1^{(i)}, \tilde{b}^{(i)}) = \sum_{b,\boldsymbol{x},\boldsymbol{y}} L_b(\tilde{b}^{(i)}) \Big[Mult_{i+1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \tilde{V}_{i+1}(\boldsymbol{x}, b) \tilde{V}_{i+1}(\boldsymbol{y}, b) + Add_{i+1}(\tilde{\boldsymbol{p}}, \boldsymbol{x}, \boldsymbol{y}) \left(\tilde{V}_{i+1}(\boldsymbol{x}, b) + \tilde{V}_{i+1}(\boldsymbol{y}, b) - 2\tilde{V}_{i+1}(\boldsymbol{x}, b) \tilde{V}_{i+1}(\boldsymbol{y}, b) \right) \Big].$$

   c) At the end of the protocol, $\mathcal{V}$ receives $\tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{b}^{(i+1)})$ and $\tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_1^{(i+1)}, \tilde{b}^{(i+1)})$.
   d) $\mathcal{V}$ computes

$$a_{i+1} = \alpha^{(i)} \tilde{mult}_{i+1}(\tilde{\boldsymbol{p}}_0^{(i)}, \tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{\boldsymbol{p}}_1^{(i+1)}) + \beta^{(i)} \tilde{mult}_{i+1}(\tilde{\boldsymbol{p}}_1^{(i)}, \tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{\boldsymbol{p}}_1^{(i+1)})$$
$$b_{i+1} = \alpha^{(i)} \tilde{add}_{i+1}(\tilde{\boldsymbol{p}}_0^{(i)}, \tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{\boldsymbol{p}}_1^{(i+1)}) + \beta^{(i)} \tilde{add}_{i+1}(\tilde{\boldsymbol{p}}_1^{(i)}, \tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{\boldsymbol{p}}_1^{(i+1)})$$

   e) $\mathcal{V}$ checks that

$$a_{i+1} \tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{b}^{(i+1)}) \tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_1^{(i+1)}, \tilde{b}^{(i+1)}) + b_{i+1} \left(\tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{b}^{(i+1)}) + \tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_1^{(i+1)}, \tilde{b}^{(i+1)}) - 2\tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{b}^{(i+1)}) \tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_1^{(i+1)}, \tilde{b}^{(i+1)}) \right)$$

   equals to the last message of the sumcheck. If checks pass, $\mathcal{V}$ uses $\tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_0^{(i+1)}, \tilde{b}^{(i+1)}), \tilde{V}_{i+1}(\tilde{\boldsymbol{p}}_1^{(i+1)}, \tilde{b}^{(i+1)})$ to proceed to the $(i+1)$-th layer. Otherwise, $\mathcal{V}$ outputs rej and aborts.
5) At the input layer d, $\mathcal{V}$ has two claims $\tilde{V}_d(\tilde{\boldsymbol{p}}_0^{(d)}, \tilde{b}^{(d)})$ and $\tilde{V}_d(\tilde{\boldsymbol{p}}_1^{(d)}, \tilde{b}^{(d)})$. $\mathcal{V}$ checks that these value are correct using the oracle call to $\tilde{V}_d(\boldsymbol{p}, b)$. If the check passes, output acc; otherwise, output rej.

Figure 3: The Optimized GKR Protocol

to the circuit, there must have at least one non-binary output leading to layer 1. This non-binary value would propagate through the circuit, ultimately resulting in a non-binary value at the output layer, contradicting the initial assumption.

For Efficiency. The prover's complexity follows directly from Theorem 1 and Theorem 4. For the verifier, the workload in each round involves computing $3(B-1)$ evaluations of a degree $3(B-1)$ polynomial for the sumcheck of variate b, which incurs $O(B^2)$ field operations. In addition, the verifier must perform $\log(|C_i|/B)$ field operations for the remaining sumcheck. In the final round, the verifier also needs to check the final sumcheck message using one oracle call to in. Therefore, the total verifier workload amounts to $O\big(d \cdot (B^2 + \log(|C|/B))\big)$ field operations plus one oracle call. □

4. Sub-linear Time Polynomial Commitment Scheme for Binary Polynomials

In companion with our optimized GKR, we proposed an adapted version of Orion [31] for binary polynomials, which enjoys sub-linear prover workload.

To perform the polynomial commitment, we first express the polynomial as a matrix $\boldsymbol{w}$ derived from its coefficient w , and specify two vector $\boldsymbol{r_0}$ and $\boldsymbol{r_1}$ such that the evaluation proof is transformed into proving $\langle w, \boldsymbol{r_0} \otimes \boldsymbol{r_1} \rangle$. Since our polynomial extension does not follow the traditional multilinear form, we detail how to properly construct its matrix representation. Recall that our polynomial extension is defined as:

$$\tilde{f}(\tilde{\boldsymbol{x}}, \tilde{b}) = \sum_{\boldsymbol{x} \in \{0,1\}^\ell} \sum_{b \in [B]} \widetilde{eq}(\boldsymbol{x}, \tilde{\boldsymbol{x}}) \cdot L_b(\tilde{b}) \cdot f(\boldsymbol{x}, b).$$

We define $x = \sum_{j=1}^{\ell} 2^{j-1} \cdot \boldsymbol{x}_j$, $N = B \cdot 2^\ell$,and let $w[x \cdot B + b] = f(\boldsymbol{x}, b), \forall x \in \{0,1\}^\ell, b \in [B]$.

The polynomial coefficients are then arranged into a $B \cdot k \times k$ matrix, where $k = 2^{\ell/2}$. Specifically, the matrix is defined as $\boldsymbol{w}[x_1 \cdot B + b, x_2] = w[(x_2 \cdot k + x_1)B + b], \forall x_1, x_2 \in [k], b \in [B]$. Since the binary polynomial coefficients are provided as B-bit integera for all $\boldsymbol{x} \in \{0,1\}^\ell$, this matrix representation preserves the given structure, which is useful for later usages.

To ensure that $\tilde{f}(\tilde{\boldsymbol{x}}, \tilde{b}) = \langle w, \boldsymbol{r_0} \otimes \boldsymbol{r_1} \rangle = \boldsymbol{r_0} \boldsymbol{w} \boldsymbol{r_1}^T$, we define $\boldsymbol{r_0}$ and $\boldsymbol{r_1}$ as follows:

$$\boldsymbol{r_0} = \otimes_{i=1}^{\ell/2} (1 - \tilde{\boldsymbol{x}}_i, \tilde{\boldsymbol{x}}_i) \otimes (L_0(\tilde{b}), \ldots, L_{B-1}(\tilde{b})),$$
$$\boldsymbol{r_1} = \otimes_{i=\ell/2+1}^{\ell} (1 - \tilde{\boldsymbol{x}}_i, \tilde{\boldsymbol{x}}_i).$$

It can be showed that this specification satisfies the required form.

To commit the polynomial, we leverage the observation in [57] that only the rows of the matrix need to be encoded using a constant-distance linear code, rather than applying a tensor code to the entire matrix.

Achieving sub-linear prover overheads raising the challenge of encoding the matrix using sub-linear operations. For general Sipelman codes over a prime field, this seems infeasible, as the encoding process would entail a linear amount of field additions, exceeding our computational budget. A workaround is to use the original Sipelman code which operates over $\mathbb{F}_2$. Its encoding algorithm involves only bitwise XOR operations, which have constant cost and are well-suited for bit-packing, enabling efficient parallel encoding for multiple rows. Concretely, it allows $O(\log N)$ rows to be encoded with a single invocation of the encoding algorithm. Consequently, the encoding process requires $O(k \cdot \frac{B \cdot k}{\log N}) = O(\frac{N}{\log N})$ RAM operations.

Another question is computing the merkle tree of the encoded matrix's columns using sub-linear operations. With Applebaum's hash [21], the commitment for each columns can be computed in $O(\sqrt{N})$ binary operations. A natural thought is again to take the advantage of word RAM model by packing $O(\log N)$ columns together and to compute the commitment for these rows in one shot.

However, achieving this requires transposing the encoded matrix, as the prover initially stores it in a column-packed format, whereas row-packed storage is needed for hashing columns efficiently. This can be done using fast algorithms for transposing binary matrices. Specifically, we partition the matrix into multiple $O(\log N) \times O(\log N)$ sub-matrices, where each sub-matrix's columns are packed in one word. The transposition is performed in two states: first, by repositioning the sub-matrices, and then by transposing each individually using a fast algorithm. This process is illustrated as follows:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \rightarrow \begin{bmatrix} A & C \\ B & D \end{bmatrix} \rightarrow \begin{bmatrix} A^T & C^T \\ B^T & D^T \end{bmatrix} \tag{7}$$

Using the algorithm from [34], each sub-matrix can be transposed with $O(\log N \times \log \log N)$ RAM operations. Consequently, the entire matrix can be transposed in $O(\frac{N}{\log^2 N} \times \log N \times \log \log N) = O(\frac{N \log \log N}{\log N})$ RAM operations, which remains sub-linear.

For opening phase, to achieve polylogarithmic proof size and verifier time, Orion compiles the code-switching technique from [19] using commit-and-prove SNARK (CP-SNARK) [58], [59], [60]. However, Hollander et al. [61] figured out that the CP-SNARK Orion designed is not sound and proposed a remedy to restore soundness, which is presented in Protocol 5.

However, as the encoding algorithm works on binary field, to achieve desired level of security, we require the random challenge from the verifier selected from the field large enough for later proximity test. Thus, the proximity test should be performed over the extension field of the original field to meet this requirement. This process would require the encoding algorithm be lifted to the extension field, leading to a new linear code over $\mathbb{K}$, which we call it extension code.

Definition 5. *We fix an $[n, k, d]$-code $\subset \mathbb{F}^n$, with generator matrix $M \in \mathbb{F}^{n \times k}$. For the extension field of $\mathbb{F}$, denoted as $\mathbb{K}$, the extension code $\hat{C} \subset \mathbb{K}^n$ of the original code C is defined as the image of the map $\mathbb{K}^k \rightarrow \mathbb{K}^n$ which sends $t \mapsto M \cdot t$.*

Intuitively speaking, the encoding algorithm for extension code is identical to the original code, with the only difference that the input vector is in the extension field, and the operation is performed in the extension field. A natural question is how the distance of the extension code would change as it would be crucial for the soundness analysis and further decide our parameter choices. The good news is that the distance of extension code is preserved. We formalize this as a theorem.

Theorem 6. *The extension code $\hat{C} \subset \mathbb{K}^n$ has distance d.*

Proof. Suppose there exists some vector $v \in \mathbb{K}^k$ such that $\Delta(M \cdot v) < d$. Here, $M \in \mathbb{F}^{k \times n}$, and the extension field $\mathbb{K}$ can be seen as $\mathbb{F}$-vector space, with the basis $\{b_t\}$. All encoding operations can be seen being done over this vector space. We express v in terms of the basis $\{b_t\}$ of $\mathbb{K}$ as: $v = (\sum_t c_{it} b_t)_{i=1}^k$, where $c_{it} \in \mathbb{F}$ are the coefficients of v with respect to the basis elements b_t. Since by assumption $\Delta(M \cdot v) < d$, there exists an index i such that $\sum_j M_{ij} \sum_t c_{it} b_t = 0$. Because the elements in the basis $\{b_t\}$ are linearly independent, this equation implies that $\sum_j M_{ij} c_{it} = 0$ must hold for each t.

Next, for some fixed k, us form a vector $c = (c_{it})_{i=1}^k \in \mathbb{F}^k$ by collecting the coefficients c_{it} for each index i. Given that $\Delta(M \cdot v) < d$ and for entries equal to 0 in $M \cdot v$, the corresponding entries of $M \cdot c$ must also be 0 by construction. This implies that $\Delta(M \cdot c) < d$, which leads to a contradiction. Thus, we conclude that the extension code must have distance at least d. The reverse inequality is straightforward: $\min_{t \in \mathbb{K}^k \setminus \{\mathbf{0}\}} \Delta(M \cdot t) \leq \min_{t \in \mathbb{F}^k \setminus \{\mathbf{0}\}} \Delta(M \cdot t)$.

Thus, the the extension code have distance d. □

With this property, we can decide the repetition times required for proximity test to achieve desired level of soundness error using the original code's distance, and the rest construction directly follows. We present the polynomial commitment scheme in Protocol 4.

In the evaluation phase, the primary computational costs come from generating $\boldsymbol{c_1}, \boldsymbol{y_1}, \boldsymbol{c_\gamma}$, and $\boldsymbol{y_\gamma}$, as the remaining computations inherently have sub-linear size. We demonstrate that $\boldsymbol{y_1}$ can be computed with sub-linear RAM operations after which $\boldsymbol{c_1}$ is obtained by encoding $\boldsymbol{y_1}$, which incurs minimal additional cost. The computation for $\boldsymbol{y_\gamma}$ and $\boldsymbol{c_\gamma}$ follows similarly.[4]

Recall that $\boldsymbol{r_0}$ can be generated using $O(B \cdot k)$ field operations, amounting to $O(B \cdot k) \cdot \omega(1)$ RAM operations. Since $\boldsymbol{y_1}$ is defined as the product $\boldsymbol{r_0} \cdot w$, each entry of $\boldsymbol{y_1}$ corresponds to the inner-product of $\boldsymbol{r_0}$ with the columns of matrix w. Notably, since the entries of w are binary, $\boldsymbol{y_1}$ can be evaluated efficiently via precomputation. To achieve this, the columns of matrix w are partitioned into $\frac{4Bk}{\log N}$ groups, each comprising a word of $\frac{\log k}{4}$ bits, which transforms into $O(2^{\log N/4}) = O(N^{1/4})$ distinct combinations. Each combination corresponds to a boolean sum of $\frac{\log N}{4}$ field elements from $\boldsymbol{r_0}$. Consequently, constructing the complete evaluation table for each group requires $O(N^{1/4} \log N)$ field operations. Across all groups, this results in a total of $O(N^{1/4} \log N \cdot \frac{4Bk}{\log N}) = O(B^{1/2} \cdot N^{3/4})$ field operations, translating to $O(B^{1/2} \cdot N^{3/4}) \cdot \omega(1)$ RAM operations.

Finally, to reconstruct each entry of $\boldsymbol{y_1}$, we extract $\frac{\log N}{4}$ binary values of each columns of matrix w (which is packed into a single word), perform table lookups and sum the result. This process is repeated $O(\frac{4Bk}{\log N})$ times across all group, where each involves $O(1)$ memory reading for the packed integer and $O(1)$ field operations for each group. Thus, reconstruct each entry of $\boldsymbol{y_1}$ requires $(\omega(1) \cdot O(\frac{4Bk}{\log N})$ RAM operations. Since $\boldsymbol{y_1}$ has length $O(k)$, the total cost is $O(\frac{4Bk^2}{\log N}) \cdot \omega(1) = O(\frac{N}{\log N}) \cdot \omega(1)$ RAM operations.

Combining all techniques stated above, we have the following theorem.

Theorem 7. *Protocol 4 is a polynomial commitment scheme that is complete and sound with sub-linear prover overhead.*

5. Evaluation

5.1. Implementation

We implemented our work in Rust using the arkworks [62] ecosystem. Our implementation also take advantage of Advanced Vector Extensions (AVX), a SIMD extension to x86 instruction sets for larger vectors, to speed up proof generation. We conducted all benchmarks on an Amazon Web Services (AWS) c7i.16xlarge compute-optimized cloud instance, equipped with a 4th-generation Intel Xeon Scalable processor ("Sapphire Rapids 8481C"), 44 virtual cores, and 128 GiB of RAM. Our implementation is currently single-threaded, and all experiments were run using a single CPU core. We report the average running time and proof size over 10 independent executions.

5.2. Comparison with Related Works on Boolean Circuits

We compare our system against the following state-of-the-art SNARK frameworks:

- **Binius** [14], which introduces a Brakedown-style PCS and adapts the Plonkish arithmetization to binary fields as its PIOP, largely based on Hyperplonk. We use the official implementation provided by the authors [63].
- **Basefold** [48], which utilizes a custom PCS paired with Hyperplonk as the PIOP. We use the official implementation over a 64-bit prime field released by the authors.
- **Lookup Arguments**. We use LogUp [22] with Hyrax-PCS as its PCS. We leverage it to directly prove word-level binary operations without explicit arithmetization. For example, to prove 2^{20} independent 8-bit AND operations—equivalent to 2^{28} boolean gates—LogUp uses a precomputed lookup table of size 2^{16} that enumerates all possible combinations of two 8-bit inputs and their corresponding outputs. Each of the 2^{20} AND operations is then verified through this lookup table.

4. This optimization also extends to the prime field setting.

PROTOCOL 4. *Sub-linear Polynomial Commitment Scheme for Binary Polynomials*

- $\mathsf{pp} \leftarrow \Pi.\mathsf{Setup}(1^\lambda, (B, 2^\ell), \mathbb{F})$: On input 1^λ, $(B, 2^\ell)$ and $\mathbb{F}$, set integer $k_1 := B \cdot 2^{\ell/2}$ and $k_2 := 2^{\ell/2}$. Return an extension field $\mathbb{K}$ for which $|\mathbb{K}| \geq 2^{\omega(\log \lambda)}$, and an $[n, k_2, d]$-code $C \subset \mathbb{F}^n$ for which $n = O(k_2)$ and $d = \Omega(n)$ and a repetition parameter $\gamma = \Theta(\lambda)$.
- $(c, u) \leftarrow \Pi.\mathsf{Commit}(\mathsf{pp}, f)$:
  - On input $f(b, \boldsymbol{x})$, parse its coefficient list as a $k_1 \times k_2$ matrix w. The prover computes encoding C_1, where C_1 is a $k_1 \times n$ matrix.
  - Compute the Merkle tree root: $\forall j \in [n] : \mathcal{R}_j \leftarrow \mathsf{Commit}(\mathsf{C}_1[:, j])$.
  - Compute the Merkle tree root $\mathcal{R} \leftarrow \mathsf{Commit}(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ and output $\mathcal{R}$ as the commitment.
- $\mathsf{Eval}(\mathsf{pp}, \mathcal{R}, \boldsymbol{x}, y, \mu, f)$: Here we assume the evaluation point $\boldsymbol{x}$ is selected from $\mathbb{K}^{1+\ell}$, and parsed as a tensor product $\boldsymbol{r} = \boldsymbol{r_0} \otimes \boldsymbol{r_1}$.
  - $\mathcal{V}$ sends a uniformly random vector $\boldsymbol{\eta} \in \mathbb{K}^{\log k_1}$ to $\mathcal{P}$.
  - The prover computes $\boldsymbol{\gamma} = \otimes_{i=1}^{\log k_1}(1 - \boldsymbol{\eta}_i, \boldsymbol{\eta}_i)$ and

$$\boldsymbol{c_1} = \sum_{i=1}^{k_1} \boldsymbol{r_0}[i]\mathsf{C}_1[i], \quad \boldsymbol{y_1} = \sum_{i=1}^{k_1} \boldsymbol{r_0}[i]w[i], \quad \mathcal{R}_{\boldsymbol{c_1}} = \mathsf{Commit}(\boldsymbol{c_1}), \quad \boldsymbol{c_\gamma} = \sum_{i=1}^{k_1} \boldsymbol{\gamma}[i]\mathsf{C}_1[i], \quad \boldsymbol{y_\gamma} = \sum_{i=1}^{k_1} \boldsymbol{\gamma}[i]w[i], \quad \mathcal{R}_{\boldsymbol{c_\gamma}} = \mathsf{Commit}(\boldsymbol{c_\gamma}).$$

  - $\mathcal{P}$ computes the answer $y = \langle \boldsymbol{y_1}, \boldsymbol{r_1} \rangle$ and sends $\mathcal{R}_{\boldsymbol{c_1}}$, $\mathcal{R}_{\boldsymbol{c_\gamma}}$ and y to the verifier.
  - $\mathcal{V}$ randomly samples and sends column indices $\hat{J} \subset [n]$.
  - The prover commits to the witness of the CP-SNARK consisting of $\boldsymbol{y_1}$, $\boldsymbol{y_\gamma}$, and $\forall j \in \hat{J}, w[:, j]$.
  - The verifier samples and sends row index set $I \subset [n], |I| = t$, as well as a new column index set $J \subset [n], |J| = t$, uniformly at random.
  - The prover computes the CP-SNARK argument π for the statement defined in Protocol 5 and sends the output $c_1[j], c_{\boldsymbol{\gamma}}[j] \forall j \in J$, $\mathsf{C}_1[i, j], \forall i \in I, j \in \hat{J}$ to the verifier.
  - The prover sends the Merkle tree proofs of $\mathsf{C}_1[i, j], i \in I, j \in \hat{J}$ under $\mathcal{R}_j$, $\mathcal{R}_j$ for all $j \in \hat{J}$ under $\mathcal{R}$, $\boldsymbol{c_1}[j]$ for all $j \in J$ under $\mathcal{R}_{\boldsymbol{c_1}}$, and $\boldsymbol{c_\gamma}[j]$ for all $j \in J$ under $\mathcal{R}_{\boldsymbol{c_\gamma}}$.
- $\mathsf{VerifyEval}(\pi, \boldsymbol{x}, y = \phi(\boldsymbol{x}), \mathcal{R})$
  - $\mathcal{V}$ checks the proof π using the CP-SNARK verification procedure.
  - $\mathcal{V}$ checks the Merkle tree proofs of $\mathsf{C}_1[i, j]$ in $\mathcal{R}_j$ for all $(i, j) \in I \times \hat{J}$, $\mathcal{R}_j$ in $\mathcal{R}$ for all $j \in \hat{J}$, $\boldsymbol{c_1}[j]$ in $\mathcal{R}_{\boldsymbol{c_1}}$ for all $j \in J$, and $\boldsymbol{c_\gamma}[j]$ in $\mathcal{R}_{\boldsymbol{c_\gamma}}$ for all $j \in J$.

Figure 4: Sub-linear Polynomial Commitment Scheme for Binary Polynomials

PROTOCOL 5 ([61]). *Code Switching Statement*

- **Committed witness:** $\boldsymbol{y_1}, \boldsymbol{y_\gamma}, w[:, j], \forall j \in \hat{J}$
- **Public input:** $\boldsymbol{\gamma}, \boldsymbol{r_0}, \boldsymbol{r_1}, y$.
- **Proof circuit:** The CP-SNARK circuit performs the following checks and computations:
  - Compute $\boldsymbol{\gamma} = \otimes_{i=1}^{\log k_1}(1 - \boldsymbol{\eta}_i, \boldsymbol{\eta}_i)$, $c_1 = \mathsf{Enc}(\boldsymbol{y_1})$, $c_{\boldsymbol{\gamma}} = \mathsf{Enc}(\boldsymbol{y_\gamma})$, and $\mathsf{C}_1[:, j] = \mathsf{Enc}(w[:, j]), \forall j \in \hat{J}$.
  - Check if $\boldsymbol{c_1}[j] = \langle \boldsymbol{r_0}, w[:, j] \rangle$ and $\boldsymbol{c_\gamma}[j] = \langle \boldsymbol{\gamma}_i, w[:, j] \rangle$ for all $j \in \hat{J}$, and $\langle \boldsymbol{r_1}, \boldsymbol{y_1} \rangle = y$.
  - $\forall j \in J$, output $\boldsymbol{c_1}[j], \boldsymbol{c_\gamma}[j]$, $\forall (i, j) \in I \times \hat{J}$, output $\mathsf{C}_1[i, j]$.

Figure 5: Code-Switching Statement in CP-SNARK

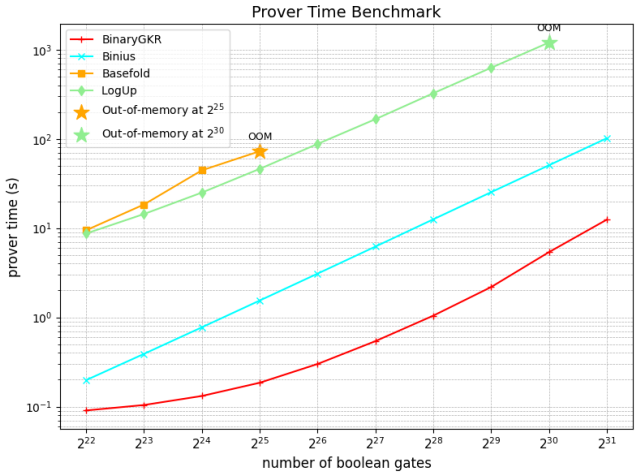

Figure 6: Prover Time for Different Works of Varying Sizes

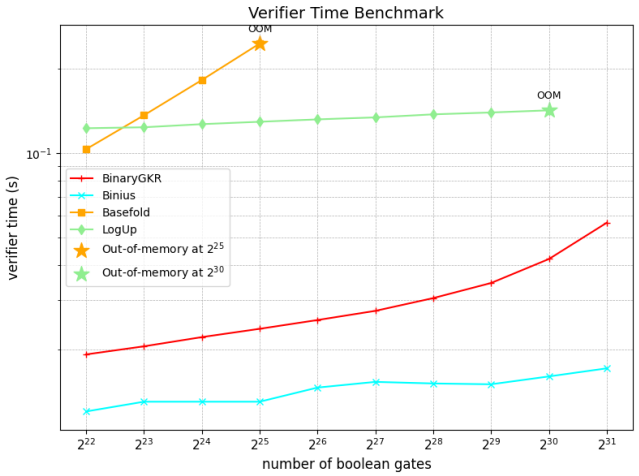

Figure 7: Verifier Time for Different Works of Varying Sizes

We note that **Blaze** [16] focuses on PCS constructions and does not provide a complete SNARK system, and is therefore not included in our experimental comparison.

All systems are evaluated on randomly generated boolean circuits. Each circuit is constructed by randomly sampling gate types, input values, and wiring patterns.

We report the prover time, proof size, and verification time of different schemes over different circuit sizes respectively in Figure 6, 7, 8.

Prover Time. As shown in Figure 6, our system, BinaryGKR, achieves the best performance throughout all tested scales. At small sizes (e.g., 2^{22} gates), only BinaryGKR and Binius complete proof generation within one second. As the circuit size increases, the differences become more pronounced. At 2^{30} gates, BinaryGKR completes in 5.38s, while Binius and LogUp require 50.61s and 1202.03s respectively. Basefold shows the slowest performance up to 2^{25} gates and fails to scale beyond 2^{26} due to memory

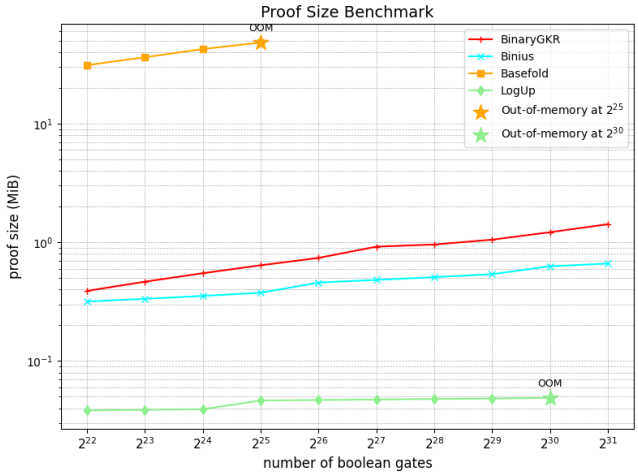

Figure 8: Proof Size for Different Works of Varying Sizes

exhaustion. Notably, Basefold operates over a 64-bit prime field rather than a binary field, which results in higher computational and memory costs when applied to Boolean circuits. Furthermore, we observe that LogUp fails to complete the proof beyond 2^{30} gates due to excessive memory consumption when generating its lookup tables. These results highlight that BinaryGKR offers strong scalability and practical efficiency, particularly when applied to large-scale Boolean circuits.

Verifier Time. In Figure 7, BinaryGKR begins with fast verification (0.019s at 2^{22} gates) and scales moderately with circuit size, reaching 0.056s at 2^{31} gates. This growth stems from the GKR-style interactive verification, which scales with circuit depth. Binius achieves the lowest verifier time overall, remaining under 0.02s across all sizes, thanks to its use of succinct verification over binary towers. LogUp exhibits a flat verifier time around 0.15s but fails beyond 2^{30} gates due to memory constraints. Basefold, on the other hand, shows rapidly increasing verifier time and does not scale beyond 2^{26} gates, reflecting the inefficiency of operating over a 64-bit prime field. While BinaryGKR does not achieve the same level of verifier speed as Binius, its efficiency remains sufficient for practical use, especially in settings where prover time is the dominant cost.

Proof Size. Figure 8 shows that BinaryGKR achieves compact proof sizes, ranging from 0.39 MiB at 2^{22} gates to 1.42 MiB at 2^{31} gates. Binius produces slightly smaller proofs (0.32–0.66 MiB), benefiting from its binary-field succinct design. In contrast, Basefold incurs the largest overhead, with proof sizes growing from 31 MiB to 48 MiB. LogUp maintains very small proof sizes (around 0.04 MiB), but is specialized for lookup-based operations and lacks general-purpose support. Overall, our scheme achieves 75–80× smaller proofs than Basefold, remains competitive with Binius, and offers broader generality than LogUp.

Overall, our system offers the most balanced trade-off among prover time, verifier time, and proof size. While Binius achieves slightly smaller proofs and low verifier

time, it falls short in proving performance. Basefold suffers from both high prover time and large proof sizes, limiting its practicality. LogUp, while optimized for lookup-based operations, lacks general-purpose support and fails to scale due to excessive prover time and memory usage. In contrast, BinaryGKR delivers fast proving, compact proofs, and moderate verifier time, making it a practical and scalable solution for large-scale Boolean circuits.

6. Conclusion

In this paper, we introduced a novel approach to optimizing the GKR protocol specifically for data-parallel binary circuits. By leveraging the binary nature of the inputs and employing advanced techniques such as bit packing, precomputation, and segmented evaluation, we significantly reduced the computational burden on the prover. These optimizations allowed us to accelerate the sumcheck process, which is the most computationally intensive part of the GKR protocol, thereby making it more efficient for circuits that are common in cryptographic applications, such as hash functions in blockchain systems.

Our field-agnostic approach ensures that these optimizations can be integrated with various SNARK technologies and adapted to different application scenarios without being constrained by the choice of the field. We also demonstrated that, in certain fields, such as binary fields or Mersenne prime fields, our techniques can achieve even greater performance improvements and compatibility.

Overall, the proposed method provides a robust, efficient, and scalable solution for proving the correctness of computations in binary circuits, making it highly suitable for applications like zero-knowledge proofs and decentralized systems. Future work could explore further optimizations and extensions of this approach to other types of circuits and computational models.

References

[1] "Zcash," 2020, https://z.cash/.

[2] A. Pruden, "zkcloud: Decentralized private computing," 2021, https://aleo.org/post/zkcloud.

[3] Z. J. Williamson, "The aztec protocol," 2018. [Online]. Available: https://github.com/AztecProtocol/AZTEC

[4] T. Liu, X. Xie, and Y. Zhang, "zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2968–2985. [Online]. Available: https://doi.org/10.1145/3460120.3485379

[5] B. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang, "ZKML: an optimizing system for ML inference in zero-knowledge proofs," in *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 2024, pp. 560–574. [Online]. Available: https://doi.org/10.1145/3627703.3650088

[6] H. Sun, J. Li, and H. Zhang, "zkllm: Zero knowledge proofs for large language models," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 4405–4419. [Online]. Available: https://doi.org/10.1145/3658644.3670334

[7] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proceedings of the 35th Annual International Conference on Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT '16, 2016, pp. 305–326.

[8] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," ePrint 2019/953, 2019.

[9] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zkSNARKs with universal and updatable SRS," in *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT '20, 2020.

[10] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 733–764. [Online]. Available: https://doi.org/10.1007/978-3-030-26954-8_24

[11] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Proceedings of the 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT '23. Cham: Springer Nature Switzerland, 2023, pp. 499–530.

[12] S. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup," in *Proceedings of the 40th Annual International Cryptology Conference*, ser. CRYPTO '20, 2020, pp. 704–737.

[13] A. Golovnev, J. Lee, S. T. V. Setty, J. Thaler, and R. S. Wahby, "Brakedown: Linear-time and field-agnostic snarks for R1CS," in *Proceedings of the 43rd Annual International Cryptology Conference*, ser. CRYPTO '23. Cham: Springer Nature Switzerland, 2023, pp. 193–226.

[14] B. E. Diamond and J. Posen, "Succinct arguments over towers of binary fields," *Cryptology ePrint Archive*, 2023.

[15] ——, "Polylogarithmic proofs for multilinears over binary towers," *IACR Cryptol. ePrint Arch.*, p. 504, 2024. [Online]. Available: https://eprint.iacr.org/2024/504

[16] M. Brehm, B. Chen, B. Fisch, N. Resch, R. D. Rothblum, and H. Zeilberger, "Blaze: Fast snarks from interleaved raa codes," *Cryptology ePrint Archive*, p. 1609, 2024.

[17] N. Ron-Zewi and R. D. Rothblum, "Proving as fast as computing: succinct arguments with constant prover overhead," in *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1353–1363. [Online]. Available: https://doi.org/10.1145/3519935.3519956

[18] J. Holmgren and R. D. Rothblum, "Faster sounder succinct arguments and IOPs," in *Advances in Cryptology – CRYPTO 2022*, Y. Dodis and T. Shrimpton, Eds. Cham: Springer Nature Switzerland, 2022, pp. 474–503.

[19] N. Ron-Zewi and R. D. Rothblum, "Local proofs approaching the witness length [extended abstract]," in *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, S. Irani, Ed. IEEE, 2020, pp. 846–857. [Online]. Available: https://doi.org/10.1109/FOCS46700.2020.00083

[20] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Zero-knowledge from secure multiparty computation," in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 21–30.

[21] B. Applebaum, N. Haramaty, Y. Ishai, E. Kushilevitz, and V. Vaikuntanathan, "Low-complexity cryptographic hash functions," in *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, ser. LIPIcs, C. H. Papadimitriou, Ed., vol. 67. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 7:1–7:31. [Online]. Available: https://doi.org/10.4230/LIPIcs.ITCS.2017.7

[22] U. Haböck, "Multivariate lookups based on logarithmic derivatives," *IACR Cryptol. ePrint Arch.*, 2022.

[23] L. Eagen, D. Fiore, and A. Gabizon, "cq: Cached quotients for fast lookups," *IACR Cryptol. ePrint Arch.*, 2022.

[24] A. Gabizon and D. Khovratovich, "flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size," *IACR Cryptol. ePrint Arch.*, 2022.

[25] A. Gabizon and Z. J. Williamson, "plookup: A simplified polynomial protocol for lookup tables," *IACR Cryptol. ePrint Arch.*, 2020.

[26] S. T. V. Setty, J. Thaler, and R. S. Wahby, "Unlocking the lookup singularity with lasso," ser. EUROCRYPT '24, 2024.

[27] A. Arun, S. T. V. Setty, and J. Thaler, "Jolt: Snarks for virtual machines via lookups," in *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part VI*, ser. Lecture Notes in Computer Science, M. Joye and G. Leander, Eds., vol. 14656. Springer, 2024, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-031-58751-1_1

[28] M. L. Fredman and D. E. Willard, "Surpassing the information theoretic bound with fusion trees," *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 424–436, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022000093900404

[29] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," vol. 39, no. 4, pp. 859–868, 1992.

[30] J. Thaler, "Time-optimal interactive proofs for circuit evaluation," in *Annual Cryptology Conference*, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 71–89.

[31] T. Xie, Y. Zhang, and D. Song, "Orion: Zero knowledge proof with linear prover time," in *CRYPTO 2022*. Cham: Springer Nature Switzerland, 2022, pp. 299–328.

[32] D. A. Spielman, "Linear-time encodable and decodable error-correcting codes," vol. 42, no. 6, pp. 1723–1731, 1996, preliminary version appeared in STOC '95.

[33] E. Druk and Y. Ishai, "Linear-time encodable codes meeting the gilbert-varshamov bound and their cryptographic applications," in *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science*, ser. ITCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 169–182. [Online]. Available: https://doi.org/10.1145/2554797.2554815

[34] H. S. Warren, *Hacker's delight*. Pearson Education, 2013.

[35] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: Interactive proofs for Muggles," in *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, ser. STOC '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 113–122.

[36] R. S. Wahby, Y. Ji, A. J. Blumberg, A. Shelat, J. Thaler, M. Walfish, and T. Wies, "Full accounting for verifiable outsourcing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 2071–2086.

[37] J. Zhang, T. Liu, W. Wang, Y. Zhang, D. Song, X. Xie, and Y. Zhang, "Doubly efficient interactive proofs for general arithmetic circuits with linear prover time," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 159–177.

[38] I. Giacomelli, J. Madsen, and C. Orlandi, "{ZKBoo}: Faster {Zero-Knowledge} for boolean circuits," in *25th usenix security symposium (usenix security 16)*. Austin, TX: USENIX Association, 2016, pp. 1069–1083.

[39] M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha, "Post-quantum zero-knowledge and signatures from symmetric-key primitives," in *Proceedings of the 2017 acm sigsac conference on computer and communications security*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1825–1842.

[40] J. Katz, V. Kolesnikov, and X. Wang, "Improved non-interactive zero knowledge with applications to post-quantum signatures," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 525–537.

[41] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 2087–2104.

[42] R. Bhadauria, Z. Fang, C. Hazay, M. Venkitasubramaniam, T. Xie, and Y. Zhang, "Ligero++: A new optimized sublinear iop," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 2025–2038.

[43] C. Delpech de Saint Guilhem, E. Orsini, and T. Tanguy, "Limbo: efficient zero-knowledge mpcith-based arguments," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3022–3036.

[44] I. Damgård and S. Zakarias, "Constant-overhead secure computation of boolean circuits using preprocessing," in *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, ser. Lecture Notes in Computer Science, A. Sahai, Ed., vol. 7785. Springer, 2013, pp. 621–641. [Online]. Available: https://doi.org/10.1007/978-3-642-36594-2_35

[45] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin, "Caulk: Lookup arguments in sublinear time," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3121–3134.

[46] S. Papini and U. Haböck, "Improving logarithmic derivative lookups using GKR," *IACR Cryptol. ePrint Arch.*, p. 1284, 2023. [Online]. Available: https://eprint.iacr.org/2023/1284

[47] B. E. Diamond and J. Posen, "Polylogarithmic proofs for multilinears over binary towers," *Cryptology ePrint Archive*, 2024.

[48] H. Zeilberger, B. Chen, and B. Fisch, "Basefold: efficient field-agnostic polynomial commitment schemes from foldable codes," in *Annual International Cryptology Conference*. Springer, 2024, pp. 138–169.

[49] S. Lin, W.-H. Chung, and Y. S. Han, "Novel polynomial basis and its application to Reed–Solomon erasure codes," in *Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science*, ser. FOCS '14, 2014, pp. 316–325.

[50] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast Reed–Solomon interactive oracle proofs of proximity," in *Proceedings of the 45th International Colloquium on Automata, Languages and Programming*, ser. ICALP '18, 2018, pp. 14:1–14:17.

[51] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1974.

[52] J. Thaler, "Proofs, arguments, and zero-knowledge," *Found. Trends Priv. Secur.*, vol. 4, no. 2-4, pp. 117–660, 2022.

[53] A. Fiat and A. Shamir, "How to prove yourself: practical solutions to identification and signature problems," in *Proceedings of the 6th Annual International Cryptology Conference*, ser. CRYPTO '86. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 186–194.

[54] G. Cormode, J. Thaler, and K. Yi, "Verifying computations with streaming interactive proofs," *Proceedings of the VLDB Endowment*, vol. 5, no. 1, pp. 25–36, 2011.

[55] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zksnarks without trusted setup," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 926–943.

[56] E. Ben-Sasson, D. Carmon, S. Kopparty, and D. Levit, "Elliptic curve fast fourier transform (ECFFT) part I: low-degree extension in time $O(n \log n)$ over all finite fields," in *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, N. Bansal and V. Nagarajan, Eds. SIAM, 2023, pp. 700–737. [Online]. Available: https://doi.org/10.1137/1.9781611977554.ch30

[57] B. E. Diamond and J. Posen, "Proximity testing with logarithmic randomness," *Cryptology ePrint Archive*, 2023.

[58] M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodriguez, "Lunar: a toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions," in *Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT '21, 2021.

[59] M. Campanelli, D. Fiore, and A. Querol, "Legosnark: Modular design and composition of succinct zero-knowledge proofs," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2075–2092. [Online]. Available: https://doi.org/10.1145/3319535.3339820

[60] D. F. Aranha, E. M. Bennedsen, M. Campanelli, C. Ganesh, C. Orlandi, and A. Takahashi, "Eclipse: Enhanced compiling method for pedersen-committed zksnark engines," in *Public-Key Cryptography – PKC 2022: 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8–11, 2022, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 584–614. [Online]. Available: https://doi.org/10.1007/978-3-030-97121-2_21

[61] T. den Hollander and D. Slamanig, "A crack in the firmament: Restoring soundness of the orion proof system and more," *Cryptology ePrint Archive*, 2024.

[62] arkworks contributors, "`arkworks` zksnark ecosystem," 2022. [Online]. Available: https://arkworks.rs

[63] B. Project, "Binius Benchmarks," 2025, https://www.binius.xyz/benchmarks/.

Appendix A. Arithmetization of Boolean Circuits

We explicitly define our arithmetic as follows. The wiring predicates (Mul and Add) are implemented as methods of the Layer class, corresponding to the functions in the equation above. Note that the eq_p_p function ($\widetilde{eq}$), mod, and $segment\_size$ variables are assumed to be defined elsewhere in the implementation.

```
# Constants for gate types
mul_type = 0  # Multiplication gate
add_type = 1  # Addition gate

class Layer:
    """Represents a single layer in the arithmetic circuit."""
    def __init__(self):
        self.gates = []   # List to store gates in this layer
        self.input = []   # Input values for this layer
        self.output = []  # Output values after evaluation

    def insert_gate(self, gate_type, input0, input1):
        """Adds a new gate to the layer."""
        self.gates.append([gate_type, input0, input1])

    def evaluate(self, input):
        """Evaluates all gates in the layer given an input."""
        result = input
        for ty, a, b in self.gates:
            for bit_index in range(segment_size):
                if ty == mul_type:
                    # Multiplication gate: multiply corresponding bits
                    result.append(result[a * segment_size + bit_index] * result[b * segment_size + bit_index] % mod)
                elif ty == add_type:
                    # Addition gate: add corresponding bits
                    result.append(result[a * segment_size + bit_index] + result[b * segment_size + bit_index] % mod)
        self.output = result
        return result

    def wiring_predicate_mul(self, tilde_p, tilde_x, tilde_y):
        """Computes the wiring predicate for multiplication gates.
        This corresponds to the Mul function in the paper."""
        result = 0
        for i, (ty, a, b) in enumerate(self.gates):
            if ty == mul_type:
                # Sum up the contributions of each multiplication gate
                result = result + eq_p_p(i, tilde_p) * eq_p_p(a, tilde_x) * eq_p_p(b, tilde_y)
                result = result % mod
        return result

    def wiring_predicate_add(self, tilde_p, tilde_x, tilde_y):
        """Computes the wiring predicate for addition gates.
        This corresponds to the Add function (omitted in the paper)."""
        result = 0
        for i, (ty, a, b) in enumerate(self.gates):
            if ty == add_type:
                # Sum up the contributions of each addition gate
                result = result + eq_p_p(i, tilde_p) * eq_p_p(a, tilde_x) * eq_p_p(b, tilde_y)
                result = result % mod
        return result

class LayeredArithmeticCircuit:
    """Represents the entire layered arithmetic circuit."""
    def __init__(self):
        self.layers = []       # List to store all layers in the circuit
        self.layer_values = [] # Stores intermediate values after each layer evaluation

    def insert_layer(self, layer):
        """Adds a new layer to the circuit."""
        self.layers.append(layer)
```

```
    def evaluate(self, input):
        """Evaluates the entire circuit given an input."""
        for layer in self.layers:
            input = layer.evaluate(input)
        return input

# Note: The `eq_p_p` function, `mod`, and `segment_size` variables are not defined in this snippet.
# They should be defined elsewhere in the codebase.
```

Appendix B. Precomputation Procedure

```
class BPolynomial:
    """
    Represents a polynomial defined by its evaluations.
    This is used to efficiently handle polynomials in the pre-computation process.
    """
    def __init__(self):
        self.evaluations = []  # The polynomial is defined by its evaluations at points 0 to B-1

    def insert_evaluation(self, evaluation):
        """Add an evaluation point to the polynomial."""
        self.evaluations.append(evaluation)

    def __mul__(self, other):
        """
        Multiply this polynomial with another.
        Note: Implementation is omitted for simplicity.
        In practice, this would perform polynomial multiplication.
        """
        pass  # Implement polynomial multiplication here

def pre_computation(tilde_b, B):
    """
    Perform pre-computation for the sumcheck optimization.

    Args:
    tilde_b: The random query point for b.
    B: The number of parallel bits (usually log(N)/3).

    Returns:
    A pre-computed table of polynomials for all possible combinations of V_i(b,x) and V_i(b,y).
    """
    eq_polynomial = BPolynomial()
    preprocessed_table = []

    # Compute the eq_b_b polynomial
    for b in range(segment_size):
        eq_polynomial.insert_evaluation(eq_b_b(b, tilde_b))

    # Iterate over all possible combinations of V_i(b,x) and V_i(b,y)
    for vx in range(2**segment_size):  # 2**8 combinations for x
        vx_polynomial = BPolynomial()
        for b in range(segment_size):
            # Convert integer to binary representation
            vx_polynomial.insert_evaluation((vx >> b) % 2)

        for vy in range(2**segment_size):  # 2**8 combinations for y
            vy_polynomial = BPolynomial()
            for b in range(segment_size):
                # Convert integer to binary representation
                vy_polynomial.insert_evaluation((vy >> b) % 2)

            # Compute g_{x,y}(b) = eq_b_b(b, tilde_b) * V_i(b,x) * V_i(b,y)
            overall_polynomial = eq_polynomial * vx_polynomial * vy_polynomial

            # Store the computed polynomial and a placeholder for future use
            preprocessed_table.append([overall_polynomial, 0])

    return preprocessed_table
```

Appendix C. Sumcheck Computation Procedure

```
def compute_f_b_optimized(preprocessed_table, V_i: Layer, tilde_p):
    """
    Compute the polynomial f(b) for the sumcheck protocol with optimization.

    This function calculates f(b) = \sum_{x,y} tilde_Mul(tilde_p, x, y) * g_{x,y}(b)
    using an optimized method that reduces the number of polynomial operations.

    Args:
    preprocessed_table: The pre-computed table containing g_{x,y}(b) polynomials and accumulators.
    V_i: The current layer in the arithmetic circuit.
    tilde_p: Random query point for p.

    Returns:
    BPolynomial: The resulting f(b) polynomial.
    """
    result = BPolynomial()

    # First pass: Accumulate coefficients in the preprocessed table
    for i in range(len(V_i.gates)):
        if V_i.gates[i][0] == mul_type:
            vx = 0
            vy = 0
            mul_coefficient = eq_p_p(i, tilde_p)

            # Convert binary representations of inputs to integers
            for b in range(segment_size):
```

```
                value_x = V_i.output[V_i.gates[i][1] * segment_size + b]
                value_y = V_i.output[V_i.gates[i][2] * segment_size + b]
                vx = vx + value_x * (1 << b)
                vy = vy + value_y * (1 << b)

            # Accumulate the coefficient in the preprocessed table
            # This is the key optimization: we're accumulating coefficients
            # instead of performing polynomial multiplications for each gate
            preprocessed_table[vx * 2** segment_size + vy][1] += mul_coefficient
            preprocessed_table[vx * 2** segment_size + vy][1] %= mod

        if V_i.gates[i][0] == add_type:
            # TODO: Implement the computation for addition gates
            # This part is omitted for simplicity in the current implementation
            pass

    # Second pass: Compute the final result
    for i in range(len(preprocessed_table)):
        # Multiply each pre-computed polynomial by its accumulated coefficient
        # and add to the result
        result = result + preprocessed_table[i][0] * preprocessed_table[i][1]

    return result

# Note: This function assumes the existence of:
# - BPolynomial class for polynomial operations
# - eq_p_p function for computing equality predicates
# - mul_type and add_type constants for gate types
# - segment_size constant for the number of bits per segment
# - mod constant for modular arithmetic
#
# Optimization details:
# - The function uses the second element of each preprocessed table entry as an accumulator
# - This reduces the number of polynomial operations from O(|C_i|) to O(2^(2*segment_size))
# - For large circuits, this can significantly reduce computation time
```

Appendix D.
Linear Prover in Libra [10]

Algorithm 4 $\mathcal{F} \leftarrow \mathsf{FunctionEvaluations}(f, \mathbf{A}, r_1, \ldots, r_\ell)$

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

Output: All function evaluations $f(r_1, \ldots, r_{i-1}, t, b_{i+1}, \ldots, r_\ell)$;

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

Algorithm 5 $\{a_1, \ldots, a_\ell\} \leftarrow \mathsf{SumCheck}(f, \mathbf{A}, r_1, \ldots, r_\ell)$

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

Output: ℓ sumcheck messages for $\sum_{x \in \{0,1\}^\ell} f(x)$. Each message a_i consists of 3 elements (a_{i0}, a_{i1}, a_{i2});

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

Algorithm 6 $\{a_1, \ldots, a_\ell\} \leftarrow \mathsf{SumCheckProduct}(f, \mathbf{A}_f, g, \mathbf{A}_g, r_1, \ldots, r_\ell)$

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

Output: ℓ sumcheck messages for $\sum_{x \in \{0,1\}^\ell} f(x)g(x)$. Each message a_i consists of 3 elements (a_{i0}, a_{i1}, a_{i2});

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

Algorithm 7 $\mathbf{A}_{h_g} \leftarrow$ Initialize_PhaseOne$(f_1, f_3, \mathbf{A}_{f_3}, g)$

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

function TABLEPRECOMPUTATION(g)
  Set $\mathbf{G}[0] = 1$
  for $i = 1, \ldots, \ell - 1$ **do**
    for $b \in \{0,1\}^i$ **do**
      $\mathbf{G}[b, 0] = \mathbf{G}[b] \cdot (1 - g_{i+1})$
      $\mathbf{G}[b, 1] = \mathbf{G}[b] \cdot g_{i+1}$
    end for
  end for
  return $\mathbf{G}$
end function
$\forall x \in \{0,1\}^\ell$, set $\mathbf{A}_{h_g} = 0$
for every (z, x, y) such that $f_1(z, x, y)$ is non-zero **do**
  $\mathbf{A}_{h_g}[x] = \mathbf{A}_{h_g}[x] + \mathbf{G}[z] \cdot f_1(z, x, y) \cdot \mathbf{A}_{f_3}[y]$
end for
return $\mathbf{A}_{h_g}$

Algorithm 8 $\mathbf{A}_{f_1} \leftarrow$ Initialize_PhaseTwo(f_1, g, u)

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

$\mathbf{G} \leftarrow$ TablePrecomputation(g)
$\mathbf{U} \leftarrow$ TablePrecomputation(u)
$\forall y \in \{0,1\}^\ell$, set $\mathbf{A}_{f_1}[y] = 0$
for every (z, x, y) such that $f_1(z, x, y)$ is non-zero **do**
  $\mathbf{A}_{f_1}[y] = \mathbf{A}_{f_1}[y] + \mathbf{G}[z] \cdot \mathbf{U}[x] \cdot f_1(z, x, y)$
end for
return $\mathbf{A}_{f_1}$

Algorithm 9 $\{a_1, \ldots, a_{2\ell}\} \leftarrow$ MultiSumCheck$(f_1, f_2, f_3, u_1, \ldots, u_\ell, v_1, \ldots, v_\ell, g)$

Input: Multilinear extensions $f_1(z, x, y)$ (with $O(2^\ell)$ non-zero entries), $f_2(x), f_3(y)$ and their bookkeeping tables $\mathbf{A}_{f_2}, \mathbf{A}_{f_3}$, randomness $u = u_1, \ldots, u_\ell$ and $v = v_1, \ldots, v_\ell$ and point g;
Output: 2ℓ sumcheck messages for $\sum_{x,y \in \{0,1\}^\ell} f_1(g, x, y) f_2(x)\ f_3(y)$;

$\mathbf{A}_{h_g} \leftarrow$ Initialize_PhaseOne$(f_1, f_3, \mathbf{A}_{f_3}, g)$
$\{a_1, \ldots, a_\ell\} \leftarrow$ SumCheckProduct$(\sum_{y \in \{0,1\}^\ell} f_1(g, x, y) f_3(y), \mathbf{A}_{h_g}, f_2, \mathbf{A}_{f_2}, u)$
$\mathbf{A}_{f_1} \leftarrow$ Initialize_PhaseTwo(f_1, g, u)
$\{a_{\ell+1}, \ldots, a_{2\ell}\} \leftarrow$ SumCheckProduct$(f_1(g, u, y), \mathbf{A}_{f_1}, f_3(y) \cdot f_2(u), \mathbf{A}_{f_3} \cdot f_2(u), v)$
return $\{a_1, \ldots, a_{2\ell}\}$

Appendix E. Fast Algorithm for Binary Matrix Transposition

Roughly speaking, the algorithm performs the similar procedure as presented in Equation 7. We present a description of the algorithm for transposing 32×32 binary matrix transposition from [34], which can be extended to arbitrary word size naturally.

```
void transpose32(unsigned A[32]) {
  int j, k;
  unsigned m, t;
  m = 0x0000FFFF

  for (j = 16; j != 0; j = j >> 1, m = m ^ (m << j
    )) {
    for (k = 0; k < 32; k = ((k | j) + 1) & ~j) {
      t = (A[k] ^ (A[k | j] >> j)) & m;
      A[k] ^= t;
      A[k | j] ^= (t << j);
    }
  }
}
```