Ceno⋆: Non-uniform, Segment and Parallel Zero-knowledge Virtual Machine

Tianyi Liu[1], Zhenfei Zhang[4], Yuncong Zhang[2], Wenqing Hu[3], and Ye Zhang[4]

[1] University of Illinois Urbana-Champaign
tianyi28@illinois.edu
[2] Shanghai Jiao Tong University
shjdzhangyuncong@sjtu.edu.cn
[3] Missouri University of Science and Technology
huwen@mst.edu
[4] Scroll Foundation
{ye,zhenfei}@scroll.io

Abstract. In this paper, we explore a novel Zero-knowledge Virtual Machine (zkVM) framework leveraging succinct, non-interactive zero-knowledge proofs for verifiable computation over any code. Our approach divides program execution proof into two stages. In the first stage, the process breaks down program execution into segments, identifying and grouping identical sections. These segments are then proved through data-parallel circuits that allow for varying amounts of duplication. In the subsequent stage, the verifier examines these segment proofs, reconstructing the program's control and data flow based on the segments' duplication number and the original program. The second stage can be further attested by a uniform recursive proof.

We propose two specific designs of this concept, where segmentation and parallelization happen at two levels: opcode and basic block. Both designs try to minimize control flow that affects the circuit size and support dynamic copy numbers, ensuring that computational costs directly correlate with the actual code executed (i.e., you only pay as much as you use). In our second design, in particular, by proposing an innovative data-flow reconstruction technique in the second stage, we can drastically cut down on the stack operations even compared to the original program execution. Note that the two designs are complementary rather than mutually exclusive. Integrating both approaches in the same zkVM could unlock more significant potential for accommodating diverse program patterns.

We present an asymmetric GKR scheme to implement our designs, pairing a non-uniform prover and a uniform verifier to generate proofs for dynamic-length data-parallel circuits. The use of a GKR prover also significantly reduces the size of the commitment: GKR allows us to commit only the circuit's input and output, whereas in Plonkish-based solutions, the prover needs to commit to all the witnesses.

⋆ *Ceno* means on-Chain Efficient Non-uniform zk-rOllup

1 Introduction

Zero-knowledge proof (ZKP) protocols [26] are a cryptographic primitive that allows a prover to convince a verifier of the correctness of computations without leaking the actual computation. Zero-Knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) system is a ZKP system that ensures that the proof size is significantly smaller than the computation size and enables faster validation. The past decade has witnessed fruitful results of zk-SNARKs, in terms of both theoretical breakthroughs [13, 15, 22, 25, 29, 50], and practical and concrete instantiations [1, 3, 41, 45, 49, 54] enabling various applications, such as anonymous payment protocols [21, 28, 40], distributed private computations [11, 17, 23, 32, 65], zero-knowledge machine learning (zkML) [30, 33, 66] and more.

In this paper, we focus on zero-knowledge Virtual Machines (zkVMs), a specific application of zk-SNARKs. zkVM allows for private and verifiable computation over generic programs. To prove that a program, represented by a list of opcodes running on a specific virtual machine, produces a desired output when given an input, a prover must generate a proof that confirms the correct execution of the opcodes using the committed inputs and results. zkVM can be seen as a superset of all previous applications, wherein a zkVM protocol, the prover must support all opcodes from the virtual machine, dynamically and of arbitrary order. In contrast, prior applications, such as anonymous payment protocols and zero-knowledge machine learning protocols, prove a static and prior-known program, i.e., a subset of the opcodes of fixed order.

Verifiable computation. zkVMs have seen many use cases. A typical one is general-purpose verifiable computation. This is achieved via applying zkVM over a Turing complete language, such as RISC-V or WASM. Since any program can be compiled to, for instance, RISC-V opcodes, in theory, one can generate proof for code written in any language while developers do not require any prior exposure to cryptography. There exists a set of toolchains that build proofs for existing languages such as RISC-V [35, 48] and WASM. On the other hand, active research is conducted to invent zero-knowledge-friendly programming languages and their intermediate representations [24, 39]. The main challenge of this route remains efficiency, with multiple research directions in the form of better and dedicated VM designs such as [24, 43, 59] and high-performance proof systems [42, 45].

zkEVM. Another typical use case of zkVM is its application to the Ethereum virtual machine, also known as zero-knowledge Ethereum Virtual machines (zkEVMs) [47, 55]. The EVM is the execution environment that runs on the Ethereum blockchain. In the Ethereum blockchain, each program is a smart contract committed publicly and executed automatically upon receiving transactions. It has been widely applied to finance, supply chains, voting systems, legal industry, etc. The EVM is a computational engine that functions as a decentralized computer, hosting and executing smart contracts on the Ethereum blockchain. It allows

developers to create applications, ensuring consistency and security across the network. One of Ethereum's biggest challenges is its scalability, in that transactions are congested due to block data and network throughput limitations.

zkEVM is one of the two major candidates for Ethereum scalability and the only one backed by cryptography, with the other being optimistic rollups and relying on game theory with financial incentives. At a high level, with zkEVM, one can aggregate (also known as *roll up*) multiple transactions into a single one, consisting of a succinct proof validating the executions of smart contracts invoked by those transactions. Abstractly speaking, those transactions happen one layer above the blockchain (and hence *layer two*) instead of the mainnet. This effectively reduces the congestion of Ethereum, resulting in orders of magnitude cheaper transactions.

Interestingly, a zkEVM can be built directly from EVM's opcode or indirectly from another zkVM. As illustrated in Figure 1, one can compile the Go or Rust implementation of EVMs [36, 38] into RISC-V opcodes and then use zkVM to prove the RISC-V opcodes. This method may look more complex than directly building a zkEVM, as it involves more components in its workflow. However, in practice, most toolchains exist already, and one only needs to build a uniform prover for a stable version of RISC-V. This alleviates the burden of code maintenance and auditing because the EVM itself is a fast-moving target and is under active development. Recent progress in [31] shows that this route delivers compelling performance compared to the first method.

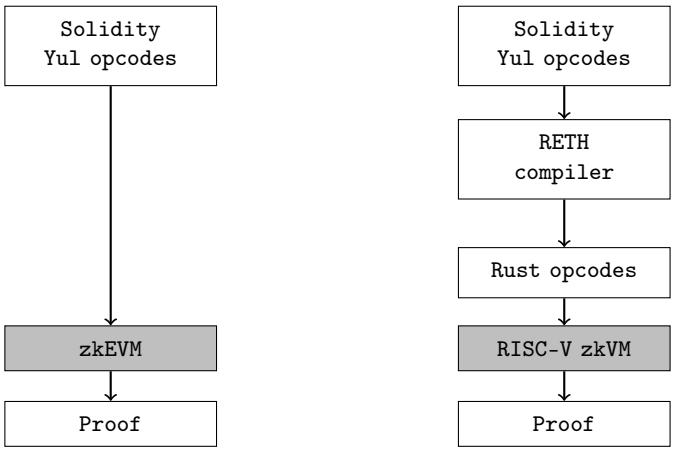


Figure 1: Two typical ways to build zero-knowledge Ethereum virtual machines

It is worth noting that the *zk* term in zk(E)VM stems from conventional reasons. For the aforementioned scalability use case, zero-knowledge is not essential and is sometimes even undesirable for regulation reasons.

1.1 Related work

Zero knowledge proof systems Zero-knowledge proofs were invented in the seminar work of [26]. Modern zk-SNARKs are constructed by compiling an information-theoretic object called an Interactive Oracle Proof (IOP) [7] to a SNARK via a polynomial commitment scheme. As briefly mentioned in previous subsections, there exists a long list of proof systems, tailored for different setups. Instead of reviewing all those candidates, we focus on a special category of proof systems, the GKR protocol.

GKR protocol. The GKR protocol was an interactive proof system first put forth by Goldwasser, Kalai, and Rothblum [25]. Converting GKR into a non-interactive protocol is a direct application of the classic Fiat-Shamir transformation [20]. For the rest of the paper, we will focus on the interactive version for ease of presentation. In such a protocol, the circuit is layered and linked via a chain of reductions, with the first and last layers dedicated to the circuit outputs and inputs. The reduction is done by repetitively invoking a sumcheck protocol [34], asserting that any given layer and its consecutive layer satisfy certain constraints derived from the actual statement. By iterating through all the layers, we enforce that the first and last layers, i.e., the outputs and inputs of the program, are valid for the circuit. We defer to Section 2.3 for a more detailed illustration of the GKR protocol.

The original GKR protocol runs in at least cubic time. Then, in [19], the complexity was reduced to quasilinear time. The start-of-the-art for data-parallel circuits is due to [56] and [61], which proves that GKR is linear for data-parallel circuits, i.e., circuits consisting of a repetitive pattern. Cormode et al. [19] considered high-degree gates to be presented in Appendix A, a useful tool for writing expressive circuits. Looking ahead, we will use an improved high-degree custom gate. For a degree d gate, our solution requires more communication ($2d$ vs. d), and in return, our proving time is linear in d, while [19] is at least quasilinear in terms of d. Although we only use the GKR prover for data-parallel circuits in our scheme, we mention that Xie et al. [63] create a linear GKR prover for general circuits, and Zhang et al. [67] improves concrete performance by allowing non-consecutive layer constraints.

GKR has also been optimized for many dedicated applications. The works [30, 33] and [2] show that GKR has excellent potential for machine learning circuits, specifically, convolutional neural networks, decision trees, and training. The works [37, 52] build concretely efficient lookup tables from sumcheck and GKR protocols. ZK-Bridge [64] reports great performance numbers for repetitive ECDSA circuits, attributing to GKR's efficiency potential of GKR.

Verifiable RAM computation Verifiable RAM computation generates proof for a program executed on a random-access memory machine. There is a series of work [5,6,8,9,12,60,69] reducing the verification of a RAM program to the verification of a circuit. Particularly, the approach by Zhang et al. [69], often referred to as the vRAM method, similarly employs GKR for data-parallel circuits in its

system. However, their design has notable limitations. First, since they leverage the highest bits of variables in MLE to select opcodes, they have to standardize the size of individual opcode circuits to a uniform size, set a maximum limit for the program length, and pad all program circuits to this cap. Second, vRAM must reorder RAM operations chronologically and then verify the time stamps and address relations between consecutive entries. This necessitates committing to the entire copy of the memory access sequence and generating range-check witnesses for the comparison of adjacent pairs. In contrast, our method eliminates the need for selecting and padding by utilizing separate circuits for each opcode and a non-uniform prover that accommodates varying lengths. Moreover, we simplify the permutation process through offline memory checking and minimize comparison witnesses by incorporating only essential comparisons directly into the opcode circuit when generating read records. This significantly reduces the number of witnesses required for stack operations, offering an advantage over the direct application of RAM techniques seen in vRAM. It's important to note that while traditional offline memory-checking protocols depend on the total memory size—requiring initial and final clearing of all entries—our adaptive prover approach needs to initialize and clear only the specific cells used during execution. This makes our protocol irrespective of the RAM address space size.

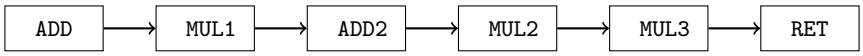
zkVMs. zkVM is an extension to verifiable RAM that proves the behavior of a virtual machine on any input program with the help of structures such as process units, stacks, memory, and chips. In the literature, there exists a list of zkVM solutions, such as Scroll [55] and Polygon [43,44]. They all follow the same paradigm:

1. Describe each sub-module in the virtual machine, including but not limited to execution units, stack, memory, and chips, in a constraint system such as Plonkish [46], rank one constraint system (R1CS) or customizable constraint system (CCS) [51];
2. Apply a SNARK protocol (not necessarily zero-knowledge) to the constraint system to generate a proof. Candidates are Plonky [41], Starky [53] or Halo2 [45] for Plonkish arithmetizations, and Groth16 [27], Marlin [16] or Spartan [50] for R1CS.
3. Leverage one or several layers of proof recursion to reduce the proof size and the verification cost. Zero-knowledgeness can be achieved during the final recursion if desired [11].

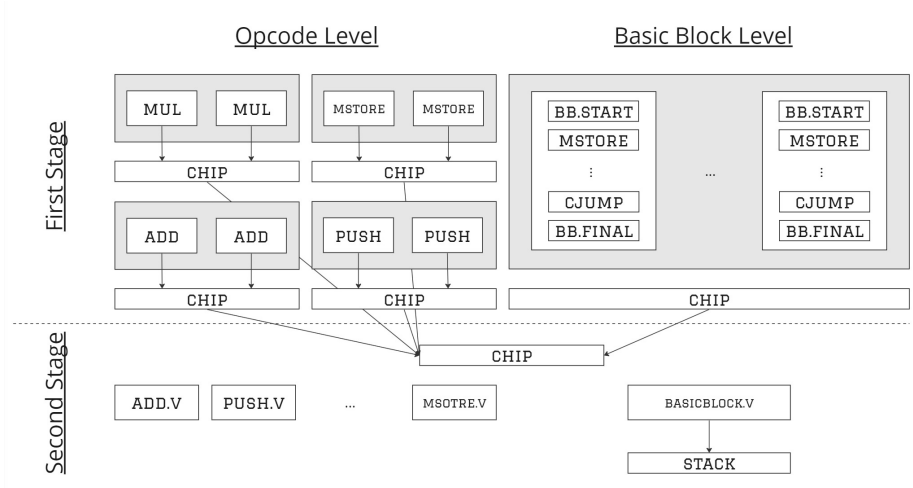
When dealing with large programs, the time it takes to prove correctness becomes a significant challenge. A practical solution is *execution continuation*, where a lengthy sequence of opcodes is split into several shorter segments. Each segment is then proven individually, with a macro proof connecting these segments. Techniques like folding schemes, exemplified by references such as Nova [29] and Protostar [13], are particularly well-suited for these scenarios. These methods allow instances from multiple small lists to be combined into a single, more manageable entity.

The above framework has a crucial drawback: The prover algorithm describes the execution logic of the virtual machine, which is uniform through all programs. Therefore, it fails to exploit the structure of the program. Considering that in the programming language domain, programs are always optimized by a compiler before being translated into machine code and executed on processors, we ask, can a prover also take advantage of the code structure before generating a uniform proof?

1.2 Our Techniques



(a) Classic zkVM design



(b) Our design

Figure 2: Classical zkVM vs Our design

Before explaining the details of our zkVM design, we choose the GKR protocol as the backend prover. Compared with other leading candidates in this domain, such as Plonk [22, 41] based solutions, GKR is better in many ways. First, the nature of most programs is that they usually have compact inputs and outputs, with a lot of intermediate data during computation. With Plonkish arithmetics, one is required to commit to all those intermediate data, whereas, in GKR, these costs are avoided. Second, GKR's IOP uses the sumcheck protocol, a linear time, parallelization-friendly protocol, whereas classic Plonk uses a univariate polynomial identity check protocol which requires the FFT algorithm for polynomial multiplications. This requires $O(n \log(n))$ time to compute.

In this paper, we introduce a new paradigm for zero-knowledge virtual machines. Existing models generally involve multiple stages. Initially, the prover creates a proof that encompasses the entire logic of the virtual machine. Later stages involve a recursive prover that encloses the proof verification within a circuit and produces a new, often smaller, proof. Typically, the first stage is the most time-consuming part of the process. Thus, by reducing the proving time in this initial stage—even if it increases costs in subsequent stages—the overall time required for all stages can be reduced.

We proposed two schemes, *Ceno Basic* and *Ceno Pro*, aimed at reducing the first-stage proving time. Both of them consist of two stages. In the first stage, the process breaks down program execution into segments, identifying and grouping identical sections. These segments are then proved through non-uniform data-parallel circuits that allow for varying amounts of duplication. For the two schemes, segmentation and parallelization happen at the opcode and basic block levels, respectively. In the second stage, the verifier examines these segment proofs, reconstructing the program's control and data flow based on the segments' duplication number and the original program. The second stage can be further attested by a uniform recursive proof.

Ceno Basic. The first scheme is named *Ceno Basic*. Unlike traditional methods that sequentially prove program opcodes in the order of their execution (as depicted in Figure 2a), our strategy involves clustering identical opcodes, as shown in the "Opcode Level" part of Figure 2b, and proving them in batches. Specifically, after executing the entire program to identify all the opcodes along with their pertinent data needed for witness generation, the two-stage methodology unfolds as follows:

1. The prover proves the correct execution and collection of the chip records for each opcode in a separate circuit.
2. The verifier verifies each separate circuit and the consistency of the chip record collections through all opcodes.

For the first stage, we model the execution of opcodes as a data-parallel circuit where the inputs are witnesses provided by the prover, and the outputs are various chip records, such as memory and stack read/write operations, bytecode lookups, or range checks, among others. The consistency of these chip records is maintained through either set equality or lookup arguments, which involves calculating the product or reciprocal sums (as detailed in the scheme introduced by [37]) of randomized records. Specifically, the collection of chip records forms a tree-structured circuit, culminating in a product or summation outcome at the tree's root. To generate a proof for this stage, we leverage the GKR backend for data-parallel circuits. The prover is non-uniform, as the number of duplicates is determined by the program's execution.

In the second stage, beyond just checking the proofs from circuits created in the previous stage, the verifier also calculates—either a product or a fractional summation—the record collection for each chip across all opcodes. This step

ensures consistency; for instance, the total product of stack pop records must match that of the stack push records. This verification stage can be transformed into a uniform proof by a recursive prover with individual proofs of each opcode's circuit, their duplication counts, and the outcomes of chip record collections as the input.

Our framework enhances existing sequential methods in several key ways. First, it bypasses the overhead associated with a universal opcode circuit—the type designed to accommodate every possible opcode for a static zero-knowledge proof system. Second, our approach allows the dynamic circuit to prove exactly as many opcodes as are executed in the original program, avoiding the need for classical solutions to prepare their static circuits for the maximum number of opcodes a program might compile into.

Ceno Pro. The current arithmetic system primarily handles fixed computations, such as circuits. Unlike circuits, A program executes branches dynamically based on the intermediate computation values. However, we can still identify fixed patterns within a program. In the field of compiler design, people leverage "basic blocks" to recognize fixed patterns of a program, which are defined as sequences of straight-line opcodes without internal branches. A program can be represented as a directed graph of these basic blocks, with execution being a walk starting from the entry block. Our improved method concentrates on these basic blocks, converting each into a data-parallel circuit. In this framework, opcodes other than stack operations become circuit gates and stack operations are modeled as wires linking these gates.

For the first stage, as indicated in the "Basic block Level" part of Figure 2b, the process is similar to *Ceno Basic*, with the prover creating a GKR proof for each individual circuit, though these circuits now reflect the layout of basic blocks. For the second phase, the verifier validates proofs for each basic block by reversely verifying GKR proofs of each opcode in the block. However, instead of the GKR circuit representation of the basic block, it only has the opcode sequence. Hence, the verifier doesn't know how to link the value from one opcode to another, or how to transmit MLE evaluations between opcodes. To reconstruct the circuit structure, it redoes the reversed stack operations that occurred in the original sequence, which are applied only to the MLE evaluations during the verification process. Consequently, we no longer implement internal stack operations of each basic block in the first-stage circuit, which produces proving cost every time an opcode is executed. Instead, we let the verifier perform stack operations just once per MLE evaluation, covering the stack values of all executions throughout the program's runtime. This approach enables us to minimize the number of stack records and associated constraints (like range checks for timestamps and stack tops) generated in the circuit, making them even fewer than the actual stack operations during program execution.

Table 1 shows the number of witness cells our protocol commits per opcode. The small number comes from the joint benefits of GKR protocol and our zkVM circuit design.

Combining *Ceno Basic* and *Ceno Pro* in a single program. We suggest that *Ceno Basic* is suited for a broader range of applications, whereas *Ceno Pro* is particularly advantageous when a basic block is executed multiple times but may underperform in other situations. However, given that each part within both *Ceno Basic* and *Ceno Pro* operates independently, it's possible to selectively apply one of these approaches based on the situation of each segment in a single program. This strategy could significantly enhance adaptability to various program patterns.

Asymmetric GKR prover. To accommodate the dynamic circuits in *Ceno Basic* and *Ceno Pro*, we present an asymmetric GKR protocol where the prover dynamically adjusts to the circuit while the verifier remains static. Therefore, during verification, the verifier will additionally receive auxiliary information for the circuit description. A similar asymmetric approach was proposed for the Plonkish prove systems in [18].

opcode	*Ceno Basic*	*Ceno Pro*
JUMP	16	0
POP	32	-
SWAP2	64	-
DUP1	32	-
ADD	128	32
GT	128	32
JUMPI	64	16
PUSH1	16	-
MSTORE	$64 + 32 \times 32$	$64 + 32 \times 32$

Table 1: Witness sizes for some EVM opcodes with 256-bit word size in *Ceno Basic* and *Ceno Pro*, where sizes are padded to the power of 2s for simplicity. We use "-" to indicate that the opcode is not needed in *Ceno Pro*. The numbers are somewhat outdated, as we have made several minor design changes in this paper that have not yet been incorporated into the implementation.

1.3 Organization of This Paper

In Section 2, we introduce several key definitions. Following this, in Section 3, we introduce the IOP protocol based on GKR. In Section 4, we define the asymmetric protocol, which forms the fundamental structure of our zkVM. Prior to delving into our primary designs, we discuss the chip arguments within our zkVM in Section 5. Subsequently, in Section 6 and Section 7, we present the fundamental architecture of the *Ceno Basic* and *Ceno Pro* protocols. Finally, in Section 8, we explore additional topics such as the application of our protocols to register-based zkVM, the integration of *Ceno Basic* and *Ceno Pro*, parallel witness generation, and alternative prover options.

2 Preliminaries

2.1 Notations

We use $\mathbf{b}$ to stand for binary input vectors, e.g., $\mathbf{b} = (b_0, \ldots, b_{n-1}) \in \{0,1\}^n$ and $\mathbf{x} = (x_0, \ldots, x_{n-1}) \in \mathbb{F}^n$, where $\mathbb{F}$ is a finite field.

We define a vector of field elements as $a(\mathbf{b}) : \{0,1\}^n \to \mathbb{F}$, which is indexed by binary string. Its Multilinear Extensions (MLE, [57, Section 3.5]) polynomial is defined by $\tilde{a}(\mathbf{X}) : \mathbb{F}^n \to \mathbb{F}$ via Definition 2, where $\mathbf{X} = (X_0, \ldots, X_{n-1})$ is a list of variables. We usually follow the conventions using $a(\mathbf{b})$ to index a value in the vector, $a(\mathbf{x})$ to indicate evaluate $a(\mathbf{X})$ on some random point $\mathbf{x}$ (usually generated by a verifier in the succinct proving protocol), and a_{eval} to denote the evaluation. We use $(\mathbf{b}_x \| \mathbf{b}_s)$, $(\mathbf{x} \| \mathbf{s})$ and $(\mathbf{X} \| \mathbf{S})$ to denote the concatenation of bit strings, random points, and variables.

We still frequently use the following functions: for $\mathbf{X}$ and $\mathbf{Y}$, let

$$\tilde{eq}(\mathbf{X}, \mathbf{Y}) = \prod_{i=0}^{n-1} ((1 - X_i)(1 - Y_i) + X_i Y_i) \ .$$

The above notion of $\tilde{eq}$ can be extended to the multi-variable case, in which we define

$$\tilde{eq}(\mathbf{X}, \mathbf{Y}^{(0)}, \ldots, \mathbf{Y}^{(d-1)}) = \prod_{i=0}^{n-1} \left((1 - X_i)(1 - Y_0^{(i)}) \cdots (1 - Y_{d-1}^{(i)}) + X_i Y_0^{(i)} \cdots Y_{d-1}^{(i)} \right) \ .$$

We use $[\![N]\!]$ to denote the set of $\{0, \ldots, N-1\}$.

2.2 Interactive Argument

Definition 1 (Interactive Argument). *We say that* $\mathsf{ARG} = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is an interactive argument of knowledge for a relation* $\mathcal{R}$ *if it satisfies the following completeness and knowledge properties.*

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

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

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

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

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

If the interactive argument of knowledge protocol ARG is public-coin, it has been shown that by the Fiat-Shamir transformation [20], we can derive a non-interactive argument of knowledge from ARG. If the scheme further satisfies the following property:

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

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

2.3 GKR Protocol

Sumcheck Protocol *Sumcheck protocol* is one of the most important interactive proofs in the literature. The sumcheck problem is to prove that the sum of a multivariate polynomial $f : \mathbb{F}^n \to \mathbb{F}$ on all binary inputs is a certain value c, i.e., $c = \sum_{b_0,\ldots,b_{n-1}\in\{0,1\}} f(b_0, \ldots, b_{n-1})$. Calculating the sum directly requires exponential time in n, as there are 2^n combinations of $b_0, \ldots, b_{n-1}$. Lund et al. [34] proposed a *sumcheck* protocol that allows a verifier $\mathcal{V}$ to delegate the computation to a computationally unbounded prover $\mathcal{P}$, who can convince $\mathcal{V}$ that σ is the correct sum. In Protocol 1, we present the non-interactive version of the sumcheck protocol after applying the Fiat-Shamir transform.

Multilinear Extension *Multilinear extension* is a type of multivariable polynomials, often represented with an array or a bookkeeping table. The definition is as follows:

Definition 2 (Multilinear Extension [19]). *Let $a : \{0,1\}^n \to \mathbb{F}$ be a function. The multilinear extension of a is the unique polynomial $\tilde{a} : \mathbb{F}^n \to \mathbb{F}$ such that $\tilde{a}(X_0, \ldots, X_{n-1}) = a(X_0, \ldots, X_{n-1})$ for all $X_0, \ldots, X_{n-1} \in \{0,1\}$. $\tilde{a}$ can be expressed as:*

$$\begin{aligned}\tilde{a}(\mathbf{X}) &= \sum\nolimits_{\mathbf{b}\in\{0,1\}^n} \tilde{eq}(\mathbf{X}, b) \cdot a(\mathbf{b}) \\ &= \sum\nolimits_{\mathbf{b}\in\{0,1\}^n} \prod\nolimits_{i=0}^{n} ((1 - X_i)(1 - b_i) + X_i b_i)) \cdot a(\mathbf{b}),\end{aligned}$$

Inspired by the closed-form equation of the multilinear extension given above, we can view an array $\mathbf{a} = (a_0, \ldots, a_{N-1})$ as a function $a : \{0,1\}^{\log N} \to \mathbb{F}$ such that $\forall i \in [0, N), a(i_0, \ldots, i_{\log N-1}) = a_i$ where i_j is the j-th bit of i. Here, we assume N is a power of two. Therefore, in this paper, we abuse the notation of multilinear extension on an array as the multilinear extension $\tilde{a}$ of a.

In this paper, we mostly utilize the sumcheck protocol for products of MLEs. Xie et al. [63] proposed the state-of-the-art algorithm, whose performance is summarized in Lemma 1.

Lemma 1. *Sumcheck protocol for a product of d MLEs with n variables runs in $O(d2^n)$ time.*

Protocol 1 (Sumcheck Protocol) *The sumcheck protocol is an interactive proof protocol between $\mathcal{P}$ and $\mathcal{V}$, described as follows:*

- $\mathsf{SC.Prove}_{n,d}(\sigma, F(\mathbf{X}))$*: With the input $\sigma \in \mathbb{F}$, $F : \mathbb{F}^n \to \mathbb{F}$ with degree at most d for each variable, $\mathcal{P}$ goes through the following steps:*
  1. *For $i = 0, \ldots, n-1$, run the following steps:*
     (a) *Set*
     $$f^{(i)}(X) = \sum_{\mathbf{b} \in \{0,1\}^{n-i-1}} F(\mathbf{b}, X, x_{n-i}, \ldots, x_{n-1}).$$
     (b) *Compute $f^{(i)}(1), \ldots, f^{(i)}(d)$ and send to the verifier.*
     (c) *Receive a challenge x_{n-i-1} from the verifier.*
- $\mathsf{SC.Verify}^{f(\cdot)}_{n,d}(\sigma)$*: With the input $\sigma \in \mathbb{F}$, $\mathcal{V}$ goes through the following steps:*
  1. *Set $\sigma_0 = \sigma$.*
  2. *For $i = 0, \ldots, n-1$, run the following steps:*
     (a) *Receive $f^{(i)}(1), \ldots, f^{(i)}(d)$ from the verifier and compute $f^{(i)}(0) = \sigma_i - f^{(i)}(1)$.*
     (b) *Randomly generate $x_{n-i-1} \leftarrow \mathbb{F}$ and send to the prover.*
     (c) *Recover $f^{(i)}(X)$ from $\left(f^{(i)}(0), \ldots, f^{(i)}(d)\right)$ and compute $\sigma_{i+1} = f^{(i)}(x_{n-i-1})$.*
  3. *Query the oracle $F_{\mathsf{eval}} = F(x_0, \ldots, x_{n-1})$. If $F_{\mathsf{eval}} = \sigma_n$, output $\mathtt{1}$. Otherwise, output $\mathtt{0}$.*

In Section 3 we will present a variant of Sumcheck protocol. The highlights here are the changes we will make in our version in Figure 4.

Figure 3: Sumcheck Protocol

GKR Protocol *GKR* [25] is an interactive protocol for general arithmetic circuits with the prover running in *linear time* in the circuit size. It uses the sumcheck protocol as a building block. Let C be a layered arithmetic circuit with depth d over a finite field $\mathbb{F}$. Here, layer 0 is the output layer, and layer d is the input layer. Each gate in the i-th layer takes inputs from two wires in the $(i+1)$-th layer. Following the conventions in prior work [19, 56, 63, 68, 70], for any i, the computation between two adjacent layers $\widetilde{V}_{i+1} : \{0,1\}^{s_{i+1}} \to \mathbb{F}$ and $\widetilde{V}_i : \{0,1\}^{s_i} \to \mathbb{F}$ is defined as follows:

$$\begin{aligned}\widetilde{V}_i(\mathbf{Z}) &= \sum\nolimits_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} f_i(\mathbf{Z}, \mathbf{x}, \mathbf{y}) \\ &= \sum\nolimits_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} \tilde{\mathsf{mul}}_{i+1}(\mathbf{Z}, \mathbf{x}, \mathbf{y}) \widetilde{V}_{i+1}(\mathbf{x}) \widetilde{V}_{i+1}(\mathbf{y}) \\ &+ \sum\nolimits_{\mathbf{x},\mathbf{y} \in \{0,1\}^{s_{i+1}}} \tilde{\mathsf{add}}_{i+1}(\mathbf{Z}, \mathbf{x}, \mathbf{y})(\widetilde{V}_{i+1}(\mathbf{x}) + \widetilde{V}_{i+1}(\mathbf{y})), \end{aligned} \tag{1}$$

where $\widetilde{V}_i$ is the MLE for the i-th layer while s_i is the number of variables of V_i. This can be verified by the sumcheck protocol, at the end of which the statement is reduced to verification on two evaluations of $\widetilde{V}_{i+1}$. Then, these two evaluation

arguments can be merged by a random linear combination as in the following equation:

$$\begin{aligned}&\alpha \cdot \widetilde{V}_{i+1}(\mathbf{X}) + \beta \cdot \widetilde{V}_{i+1}(\mathbf{Y}) \\ &= \sum\nolimits_{\mathbf{z} \in \{0,1\}^{s_{i+1}}} (\alpha \cdot \tilde{eq}(\mathbf{X}, \mathbf{z}) + \beta \cdot \tilde{eq}(\mathbf{Y}, \mathbf{z})) \cdot \widetilde{V}_{i+1}(\mathbf{z}).\end{aligned}$$

where α and β are sampled from the transcript. Both equations above can be proved by sumcheck protocols in linear time.

For layered circuits, the state-of-art instantiation of GKR protocol is from Xie et al. [63]. We recap their results as follows:

Theorem 1. *For an input size n and a finite field $\mathbb{F}$, there is a zero-knowledge argument protocol for the relation:*

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

under q-Strong Bilinear Diffie-Hellman and (d, ℓ)-Extended Power Knowledge of Exponent assumptions. Moreover, for every $(C, \mathbb{x}; \mathbb{w}) \in \mathcal{R}$, the running time of $\mathcal{P}$ is $O(|C|)$ field operations and $O(n)$ multiplications in the base group of the bilinear map. The running time of $\mathcal{V}$ is $O(|\mathbb{x}|+d\cdot\log|C|)$ if C is log-space uniform with d layers. $\mathcal{P}$ and $\mathcal{V}$ interact $O(d \log |C|)$ rounds and the total communication (proof size) is $O(d \log |C|)$. In case d is $\mathsf{polylog}(|C|)$*, the protocol is a succinct argument.*

Zhang et al. [67] generalize the GKR protocol to non-consecutive layered circuits. In their scheme, the circuit remains layered, but the input of a given layer may come from multiple previous layers. They prove the following result over this type of circuit:

Theorem 2. *Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d general arithmetic circuit. An interactive proof exists for the function computed by C with soundness $O(d \log |C|/|\mathbb{F}|)$. The running time of $\mathcal{P}$ is $O(|C|)$. The proof size is $\min\left\{O(d \log C + d^2), O(|C|)\right\}$. Let the time to evaluate all gate evaluations at random points be T. Then, the running time of $\mathcal{V}$ is $\min\left\{O(n + d \log |C| + d^2 + T), O(|C|)\right\}$.*

2.4 Offline Memory Checking

Offline memory checking, introduced by Blum et al. [10], focuses on verifying the consistency of memory access sequences, which include load (read) and store (write) operations. Each memory operation is represented as a triplet (a, v, t), detailing the address, value, and timestamp, respectively. The methodology is to maintain two separate sets, R for reads and W for writes. Assuming the memory address space is denoted by A, the verification process unfolds in the following steps:

1. Initially, the write operation set W is set up with $\{(a, 0, 0)\}_{a \in A}$, signifying the memory's initial state.

2. Upon a read operation (a, v, t), the sets are updated as $R = R \cup \{(a, v, t')\}$ and $W = W \cup \{(a, v, t)\}$, ensuring that the timestamp t' of when v was last written to address a is less than t.
3. For a write operation (a, v, t), the sets are modified to $R = R \cup \{(a, v', t')\}$ and $W = W \cup \{(a, v, t)\}$, with a check that t', the timestamp when the previous value v' was written to address a, is less than t.
4. The process concludes by incorporating into R the entries $\{(a, v_a, t_a)\}_{a \in A}$, where t_a is the most recent timestamp at which a value v_a was written to address a.

Additionally, they propose simplified protocols for both stack and queue data structures, effectively halving the total size of $|R| + |W|$. The integrity of the operation sequences is assured by demonstrating that $R = W$. This constraint is instantiated through the following argument:

Set equality argument For two sets $S_1 = \left\{v_1^{(0)}, \ldots, v_1^{(|S_1|-1)}\right\}$ and $S_2 = \left\{v_2^{(0)}, \ldots, v_2^{(|S_2|-1)}\right\}$, with an extra challenge τ, the constraints to check $S_1 = S_2$ is

$$\prod_{i=0}^{|S_1|-1} \left(v_1^{(i)} + \tau\right) = \prod_{i=0}^{|S_2|-1} \left(v_2^{(i)} + \tau\right). \tag{2}$$

where $v_b^{(i)}$ with $b \in \{1, 2\}$ can either be a single element or a random linear combination of a tuple with some randomness β.

We further generalize the usage of offline memory checking in Section 5.

Lookup argument Lookup argument is defined as proving $A \subseteq T$ for two multisets, $A = a_1, a_2, \ldots, a_{|A|}$, referred to as the input, and $T = t_1, t_2, \ldots, t_{|T|}$, referred to as the table. Setty et al. [52] observe that the lookup operations are similar to the read operations on read-only memory (ROM). They propose an efficient method that reduces the lookup argument to a simplified simplified offline memory checking.

3 GKR IOP

Justin Thaler's textbook [58] notes that adding randomness can significantly boost the efficiency of specific proving tasks with only a minimal impact on soundness. An illustrative case is the set equality problem. The conventional, deterministic methods—either direct comparison or sorting—require either quadratic or quasi-linear complexity, respectively, making them impractical for arithmetic circuits due to their computational expense. By introducing randomness, one can utilize a technique known as the grand-product argument to efficiently verify set equality using a straightforward circuit design with a linear gate count, similar to the approach used in the PLONK cryptographic system [22].

The original GKR protocol is to generate proofs for computational circuits. We propose the following enhancements:

- **Expansion of the GKR arithmetic framework:** We have adapted the GKR system to include multiple independent sections in both inputs and outputs. Looking ahead, this development is designed to facilitate the construction of the circuit set and enable connections between subsets of inputs and outputs across different circuits.
- **Introduction of GKR Interactive Oracle Proofs (IOP):** This incorporates verifier-generated challenges into the witness generation process, thereby enhancing the efficiency of argument design.

For clarity and focus, the specifics of the arithmetic framework expansion are deferred to Appendix A. This section will primarily introduce and detail the GKR IOP concept.

Our protocol operates in three distinct phases. In the first phase, the prover and the verifier interact multiple rounds, exchanging witness oracles and challenges. The procedure for each round is as follows:

1. The prover presents the oracle of a new witness vector.
2. The prover receives a new challenge from the verifier. This challenge is used by the prover to construct witnesses for subsequent rounds.

In the second stage, once the oracles for all rounds are sent to the verifier and the challenges are prepared, the prover calculates the witnesses for every layer within the circuit. Following this, both parties proceed to execute the GKR protocol.

In the third stage, upon completion of the GKR protocol, the statement is condensed to a series of evaluation statements for the input layer. The verifier then queries the witness oracles to verify the correctness of these evaluations. The full protocol is shown in Multi-round GKR IOP.

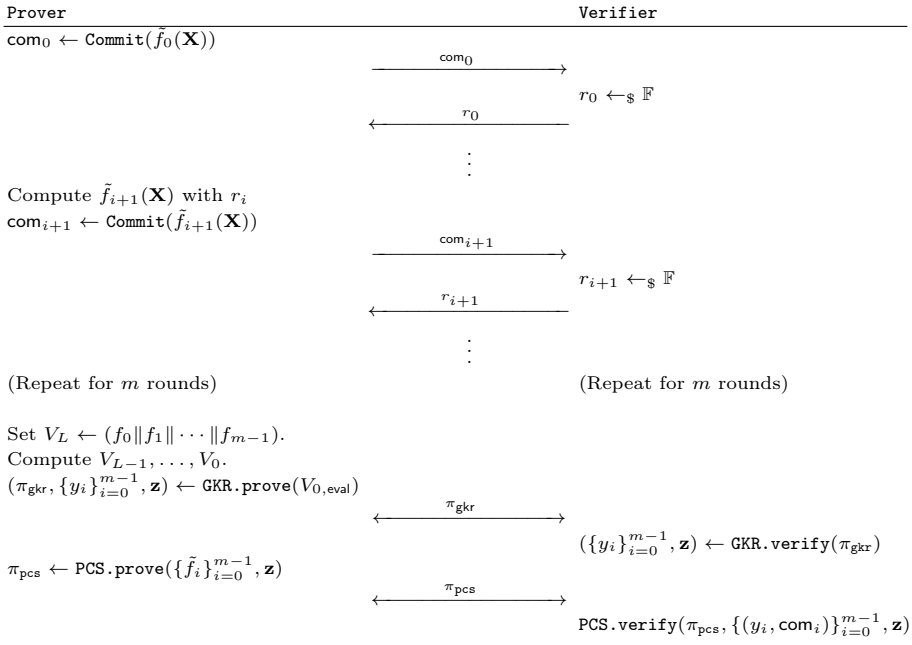
4 Asymmetric Protocols

4.1 Non-uniformity Leads to Better Performance

The prevailing SNARK schemes are primarily designed for static computational frameworks, such as arithmetic circuits. Implementing SNARKs for dynamic computations introduces additional complexities. For instance, while a sumcheck protocol is adequate for validating the summation of a fixed-length array, addressing a variable-length array—where the array length is an input parameter—requires integrating loop logic within the proving circuits. To address this challenge, the naive approach initially sets a maximum length, $N_{\max}$. It then introduces additional witnesses and constraints to handle the incremental updates of both the summation and the iterator, as well as comparing the iterator to the actual input length. Collectively, these modifications result in an $O(N_{\max})$ proving overhead relative to a conventional sumcheck protocol.

To minimize the proving cost, the verifier now handles proofs submitted by a class of provers, where each prover generates a proof for a specific, fixed

Scheme 1: Multi-round GKR IOP



computation. Upon receiving and accepting a proof, the verifier also receives auxiliary information that describes the computation the prover proved. This strategy shifts some of the computational burden from the prover to the verifier while simultaneously reducing the overall overhead.

In the context of the earlier array summation example, each prover produces a sumcheck proof for an array of fixed length $N = 2^n$. The verifier, upon receiving the proof, also obtains a n parameter, reflecting that the proof involves $O(n)$ sumcheck rounds. The verifier then uses a variable-length loop to verify the received proof. This approach ensures that each prover has minimal overhead while the verifier incurs only $O(\log N_{\max})$ overhead—significantly lower than that required by a variable-length prover system. The protocol is shown in Protocol 2.

In the scenario where a class of provers is paired with a verifier, we designate these provers as *non-uniform provers*, each generating a *non-uniform proof*, and refer to the verifier as a *uniform verifier*. When transforming the verifier into a recursive prover, the recursive prover then generates a *uniform proof*.

4.2 Asymmetric Sumcheck and GKR Protocol

We define asymmetric protocols. A pair of prover and verifier $(\mathcal{P}^{(C)}(\mathbb{x}, \mathbb{w}), \mathcal{V}^{(C)}(\mathbb{x}))$ is **symmetric** and $(\mathcal{P}^{(C)}(\mathbb{x}, \mathbb{w}), \mathcal{V}^{(\cdot)}(\langle C \rangle, \mathbb{x}))$ is **asymmetric**.

From the previous example, we can easily derive the asymmetric sumcheck (ASC) protocol with variable size, as shown in Protocol 2. By replacing SC.Verify

Protocol 2 (Asymmetric Sumcheck Protocol) *Sumcheck protocol with variable length is an interactive proof protocol between* $\mathsf{ASC}.\mathcal{P}$ *and* $\mathsf{ASC}.\mathcal{V}$ *as follows:*

- $\mathsf{ASC.Prove}_{n,d}(\sigma, F(\mathbf{X}))$*: Return* $\mathsf{SC.Prove}_{n,d}(\sigma, F(\mathbf{X}))$.
- $\mathsf{ASC.Verify}^{F(\cdot)}_{n_{\mathsf{max}},d}(n, \sigma, \pi_{\mathsf{SC}})$*: With input* $\sigma \in \mathbb{F}$, $\mathcal{V}$ *goes through the following steps:*
  1. *Generate* $\mathsf{idx} = (\mathsf{idx}_0, \ldots, \mathsf{idx}_{n_{\mathsf{max}}}) = (0, 1, \ldots, n_{\mathsf{max}})$.
  2. *Set* $\sigma_0 = \sigma$, $\mathsf{state}_{\mathsf{in\_sc}} = 1$.
  3. *For* $i = 0, \ldots, n_{\mathsf{max}} - 1$*, run the following steps:*
     (a) *If* $\mathsf{idx}_i = n$*, then set* $\mathsf{state}_{\mathsf{in\_sc}} = 0$.
     (b) *If* $\mathsf{state}_{\mathsf{in\_sc}} = 1$*:*
        i. *Compute* $f^{(i)}(0) = \sigma_i - f^{(i)}(1)$.
        ii. *Randomly generate* $x_{n-i-1} \leftarrow \mathbb{F}$ *and sends it to* $\mathcal{P}$.
        iii. *Recover* $f^{(i)}(X)$ *from* $\left(f^{(i)}(0), \ldots, f^{(i)}(d)\right)$ *and compute* $\sigma_{i+1} = f^{(i)}(x_{n-i-1})$.
  4. *Query the oracle* $F_{\mathsf{eval}} = F(x_0, \ldots, x_{n-1})$*. If* $F_{\mathsf{eval}} = \sigma_n$*, output* `1`*. Otherwise, output* `0`.

The orange highlights the differences between this and a classic sumcheck protocol in Figure 3.

Figure 4: Asymmetric Sumcheck Protocol

with ASC.Verify, we directly obtain the basic version of asymmetric GKR protocol (AGKR), which we denote by AGKR-L.Verify.

We implement the AGKR protocol across two distinct types of structured circuits: data-parallel circuits and tree-structured circuits, as detailed below:

- **AGKR for data-parallel circuit (denoted by AGKR-DP).** For in-depth information, please refer to Appendix A.
- **AGKR for tree-structured circuit (denoted by AGKR-TS).** This structure is characterized by a sequential connection of data-parallel sub-layers based on the same sub-circuit. Consider a sub-circuit C with an input size B and an output size of 1. The tree-structured circuit then comprises $N = B^n$ leaf nodes on the n-th layer. For the i-th layer, there are B^i nodes. The 0-th layer is the output. In our zkVM design, we use $\mathcal{G}_{\mathsf{prod}}$ to compute the grand product necessary for the set equality check. Here, the leaves represent the set elements, augmented by a random challenge, and the sub-circuit performs a simple operation: $\mathsf{prod}(a, b) = ab$, calculating the product of inputs from two child nodes.

The protocol AGKR-DP is derived from AGKR-L, as described in Appendix A. We construct AGKR-TS by repeatedly deploying AGKR-DP. Details of these protocols are not included here.

Both the ASC and AGKR protocols are pivotal elements of our zkVM design. Their implementation demonstrates how introducing asymmetry in the design enhances overall efficiency.

4.3 Asymmetric Protocols in Prover Chain

In practical zero-knowledge proof systems, proof recursion is often employed multiple times to compress the size of proofs. This process can be conceptualized as a sequential chain of provers, where each prover verifies the preceding proof and recursively generates another proof of this process, presented as follows:

$$\left(\mathcal{P}_0^{(C_0)}(\mathbb{x}_0, \mathbb{w}_0), \mathcal{P}_1^{(C_1)}(\mathbb{x}_1, \mathbb{w}_1), \ldots, \mathcal{P}_{r-1}^{(C_{r-1})}(\mathbb{x}_{r-1}, \mathbb{w}_{r-1})\right)$$

where $\mathcal{P}_i^{(C_i)}(\mathbb{x}_i, \mathbb{w}_i)$ is a prover who generates a proof for the computation C_i, $\mathbb{x}$ is the common input to the prover and the verifier, and $\mathbb{w}$ is the prover's private input. For the adjacent pair $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$, $\mathcal{P}_{i+1}$ is used to generate a new proof asserting the proof π_i generated by $\mathcal{P}_i$ is correct. For this reason, $\mathcal{P}_{i+1}$ is often referred to as "recursive prover" of $\mathcal{P}_i$.

Building upon the observations above, A pair of adjacent provers $(\mathcal{P}_i^{(C_i)}(\mathbb{x}_i, \mathbb{w}_i)$, $\mathcal{P}_{i+1}^{(C_{i+1})}(\mathbb{x}_{i+1}, \mathbb{w}_{i+1}))$ is **symmetric** when

$$\begin{aligned} C_{i+1}(\cdot) &= \mathcal{V}_i^{(C_i)}(\cdot) \\ \mathbb{x}_{i+1} &= (\mathbb{x}_i, \pi_i) \end{aligned}$$

for verifier $\mathcal{V}_i^{(C_i)}$ corresponding to $\mathcal{P}_i^{(C_i)}$. Conversely, the pair $(\mathcal{P}_i^{(C_i)}(\mathbb{x}_i, \mathbb{w}_i)$, $\mathcal{P}_{i+1}^{(C_{i+1})}(\mathbb{x}_{i+1}, \mathbb{w}_{i+1}))$ is **asymmetric**, when

$$\begin{aligned} C_{i+1}(\cdot) &= \mathcal{V}_i^{(\cdot)}(\cdot) \\ \mathbb{x}_{i+1} &= (\mathbb{x}_i, \pi_i, \langle C_i \rangle) \end{aligned}$$

where $\mathcal{V}^{(\cdot)}$ acts as a universal circuit that accepts a computation representation as an input, and $\langle C \rangle$ denotes the description of C.

In most existing zkVM frameworks, the initial circuit configuration is typically set with $C_0 = \mathsf{VM}$, and $\mathbb{x} = (\langle \Pi \rangle, x)$, where VM represents the virtual machine logic, Π is the program whose correctness is to be proven, and x is the program's input. Throughout the proof chain, all adjacent prover pairs are symmetric. However, in scenarios where each prover is equipped with succinct proving schemes—meaning $\mathcal{P}_i$ operates in polynomial time and $\mathcal{V}_i$ in sub-linear time—the computational cost predominantly accumulates at $\mathcal{P}_0$. Given that the circuit sizes for subsequent provers decrease progressively, with the rate of decrease varying across different proving schemes, our approach is specifically designed to minimize the proving time for $\mathcal{P}_0$. This optimization is crucial for enhancing the overall efficiency and effectiveness of the system.

In the previous section, we demonstrated that shifting the computational overhead from the prover to the verifier using an asymmetric protocol effectively reduces the prover's costs. Consequently, we expand the concept to the entire proving chain by permitting any adjacent pair to adopt an asymmetric configuration. Applying this principle to the zkVM, we initialize with $C_0 = (\mathsf{VM}, \Pi)$ and $\mathbb{x} = x$, which positions $\mathcal{P}_0$ as a non-uniform prover.

Looking forward, we propose two designs for the zkVM: *Ceno Basic* and *Ceno Pro*. In *Ceno Basic*, the verifier simply receives the count of each type of opcode used in the execution trace. Meanwhile, *Ceno Pro* takes a more detailed approach, where the verifier receives segments of opcodes representing the circuit. This design includes a method for verifying the GKR proof of the circuit represented by these opcode segments instead of the GKR arithmetic representations.

5 Chip Arguments in *Ceno Basic* and *Ceno Pro*

The architecture of zkVM is composed of several key components beyond the basic opcode execution unit. Firstly, it incorporates stacks or registers, which are essential for transmitting single values between opcodes. Secondly, it is equipped with memory capable of storing data with a large and mutable size. Lastly, the system utilizes multiple chiplets specifically designed for rapid computation of frequent, non-arithmetic tasks such as hashing, bitwise operations, and comparisons.

In this section, we collect results from several papers [10, 52], reducing all operations to the set-equality arguments.

5.1 Reductions to Set-equality Arguments

Set-equality arguments. We formalize the relation of set-equality checking as follows:

$$\mathcal{R}_{\mathsf{SE}} = \left\{ (\mathsf{record}_{\mathsf{W}}, \mathsf{record}_{\mathsf{R}}) : \begin{array}{l} \mathsf{record}_{\mathsf{W}} = \{w_0, \ldots, w_{N_w-1}\} \ , \\ \mathsf{record}_{\mathsf{R}} = \{r_0, \ldots, r_{N_r-1}\} \ , \\ \text{s.t.} \ \prod_{i=0}^{N_W-1} (w_i + X) = \prod_{i=0}^{N_R-1} (r_i + X) \ . \end{array} \right\} \tag{3}$$

where w_i (or r_i) can be a single entry or derived from a random linear combination of multiple entries.

Random-access memory with read-once values. The *random-access* memory is a type of memory that allows for reading and modification in any order. In zkVM, we typically use (a, v, t) to represent a memory access operation record, where a stands for the address, v for the value, and t for the timestamp. To ensure the integrity of operations, the consistency of read and write records, denoted by $\mathsf{record}_{\mathsf{R}}$ and $\mathsf{record}_{\mathsf{W}}$, respectively, is critical. We start by assuming that each written value is read at most once. To establish the consistency of these records, the process involves matching each write record with a corresponding read record, employing set-equality arguments. Unlike typical SE relations, it is crucial to ensure that each write operation occurs before the associated read. Additionally, it is possible that some write operations may not be canceled.

Therefore, a valid pairing of $(\mathsf{record_W}, \mathsf{record_R})$ must comply with these specific relational conditions.

$$\mathcal{R}_{\mathsf{RAM\text{-}RO}} = \left\{ (\mathsf{record_W}, \mathsf{record_R}) : \begin{array}{l} \mathsf{record_W} = \{w_0, \ldots, w_{N_w-1}\} \ , \\ \mathsf{record_R} = \{r_0, \ldots, r_{N_r-1}\} \ , \\ \text{for each } r_i.\ \exists t'_i < r_i.t, \\ \text{define } r'_i = (r_i.a, r_i.v, t'_i), \\ \qquad \mathsf{record_{R'}} = (r'_0, \ldots, r'_{N_r-1}) \ , \\ \exists \mathsf{record_{R''}} = \{(r_i.a'', r_i.v'', r_i.t'')\}_{i \in [\![N''_r]\!]} \ , \\ \text{s.t. } (\mathsf{record_W}, \mathsf{record_{R'}} \cup \mathsf{record_{R''}}) \in \mathcal{R}_{\mathsf{SE}} \end{array} \right\} \tag{4}$$

Note that we only finalize the memory cells that occur in the records

Random-access memory. When we assume that each written value can be read only once, this effectively means that writing and reading correspond to creating and erasing a value on the RAM, respectively. Upon discarding this assumption to permit a written value to be read zero or multiple times, we generate a new pair of read and write records derived from the existing ones, as aforementioned in Section 2.4. Notably, the initial read record lacks a corresponding write record for each memory address. A default value is written to memory at $\mathsf{clk} = 0$ to address this. Traditional offline memory-checking protocols necessitate initializing all memory cells. However, by employing a non-uniform prover, it becomes feasible to allow the prover to identify a dynamically sized set of addresses, requiring initialization only for the touched memory cells in the execution trace. The relationship is defined as follows:

$$\mathcal{R}_{\mathsf{RAM}} = \left\{ (\mathsf{record_W}, \mathsf{record_R}) : \begin{array}{l} \mathsf{record_W} = \{w_0, \ldots, w_{N_w-1}\} \ , \\ \mathsf{record_R} = \{r_0, \ldots, r_{N_r-1}\} \ , \\ \text{for each } r_i.\ \text{define } r'_i = r_i, w'_i = r_i, \\ \text{for each } w_i.\ \text{define } w'_{i+N_r} = w_i, \\ \qquad \exists v'_i, r'_{i+N_r} = (w_i.a, v'_i, w_i.t), \\ \text{define } \mathsf{record_{R'}} = \left\{r'_0, \ldots, r'_{N_r+N_w-1}\right\} \ , \\ \qquad \mathsf{record_{W'}} = \left\{w'_0, \ldots, w'_{N_r+N_w-1}\right\} \ , \\ \exists \mathsf{record_{W''}} = \{(w_i.a, w_i.v, 0)\}_{i \in [\![N''_w]\!]} \ , \\ \qquad \text{with unique } w_i.v, i \in [\![N''_w]\!], \\ \text{s.t. } (\mathsf{record_{W'}} \cup \mathsf{record_{W''}}, \mathsf{record_{R'}}) \in \mathcal{R}_{\mathsf{RAM\text{-}RO}} \end{array} \right\} \tag{5}$$

Lookup arguments. Setty et al. [52] have demonstrated that lookup operations resemble operations on a *write-once memory*. Therefore, lookup arguments

can be reduced to set-equality arguments. We denote the table and input records are record_T and record_A, respectively. The relation is as follows:

$$\mathcal{R}_{\mathsf{Lookup}} = \left\{ (\mathsf{record}_T, \mathsf{record}_A) : \begin{array}{l} \mathsf{record}_T = (\tau_0, \ldots, \tau_{N_\tau - 1}) \ , \\ \mathsf{record}_A = (\alpha_0, \ldots, \alpha_{N_\alpha - 1}) \ , \\ \text{for each } i \in [\![N_\alpha]\!], \\ \quad \exists t_i, r_i = (\alpha_i, t_i), w_i = (\alpha_i, t_i + 1) \ , \\ \text{define } \mathsf{record}_{\mathsf{R}} = \{r_0, \ldots, r_{N_\alpha - 1}\} \ , \\ \quad \mathsf{record}_{\mathsf{W}} = \{w_0, \ldots, w_{N_\alpha - 1}\} \ , \\ \text{for each } i \in [\![N_\tau]\!], \ w'_i = (\tau_i, 0), \exists t'_i, r'_i = (\tau_i, t'_i), \\ \text{define } \mathsf{record}_{\mathsf{W}'} = (w'_0, \ldots, w'_{N_\tau - 1}) \ , \\ \quad \mathsf{record}_{\mathsf{R}'} = (r'_0, \ldots, r'_{N_\tau - 1}) \ , \text{ s.t.} \\ (\mathsf{record}_{\mathsf{W}} \cup \mathsf{record}_{\mathsf{W}'}, \mathsf{record}_{\mathsf{R}} \cup \mathsf{record}_{\mathsf{R}'}) \in \mathcal{R}_{\mathsf{SE}} \end{array} \right\} \tag{6}$$

Unify four types in zkVM In our zkVM, we incorporate multiple specialized chips such as the stack, memory, and range chips, which correspond to $\mathcal{R}_{\mathsf{RAM}}$-RO, $\mathcal{R}_{\mathsf{RAM}}$ and $\mathcal{R}_{\mathsf{Lookup}}$, respectively. All of these can be reduced to $\mathcal{R}_{\mathsf{SE}}$, allowing us to unify them. Within each opcode circuit, we validate the correctness of execution and operate on these chips. Furthermore, we reduce the chip operations and to the entries in $\mathsf{record}_{\mathsf{W}}$ and $\mathsf{record}_{\mathsf{R}}$, highlighted in blue in the relations. Each entry is formatted as $(\mathsf{tag}, \mathsf{item}_0, \mathsf{item}_1, \ldots)$ with tag potentially being $'\mathsf{stack}'$, $'\mathsf{memory}'$ and so on.

For a more nuanced discussion on RAM-RO and RAM, despite the memory typically being constrained by $\mathcal{R}_{\mathsf{RAM}}$, permitting multiple reads of the same value, introducing more flexibility can significantly enhance concrete efficiency. For example, when the access pattern is predictable in the memory chip—such as consecutive memory operations that alternate between reading and writing to the same address with consistent intervals—we can limit ourselves to RAM-RO constraints and potentially omit the range check for timestamps.

6 *Ceno Basic*

Many current zkVM implementations process opcodes in sequence. This approach necessitates additional control witnesses, like opcode-type identifiers and indicators marking an opcode's or block's beginning and end. These control witnesses are far from trivial and can, in cases like Scroll's zkEVM [55], amount to over a hundred controller witnesses for each opcode, contributing significantly to the overall execution complexity.

In this section, we introduce a new framework for zkVM to minimize the need for such controlling witnesses.

6.1 Overview

We model a virtual machine (VM) that interprets a predefined set of opcodes, each uniquely identified within the set $[\![Q]\!] = 0, 1, \ldots, Q-1$. A program $\Pi := (\mathsf{op}_0, \ldots, \mathsf{op}_{m_\Pi - 1})$ is thus a sequence of these identifiers, making Π a member of $[\![Q]\!]^{m_\Pi}$, which maps the sequence of operations to their opcodes.

We design *Ceno Basic* to prove a program Π's execution based on given inputs and outputs. The framework of *Ceno Basic* is a circuit set. The opcode circuits input prover-provided witnesses and output chip records according to the opcode computational logic. The complete collection of chips within our system is represented by CHIP, which includes global state chips, stack, memory, and lookup-based chips. The subset of lookup-based chips, denoted by Lookup, encompasses chips such as bytecode chips and range chips. All circuits occurring in the circuit set are as follows:

- $\mathcal{G}^{(\mathsf{op})}$ is the circuit computing $\mathsf{op} \in [\![Q]\!]$, which inputs $\mathsf{in}^{(\mathsf{op})}$ and outputs chip records for each chip $(\mathsf{record}_W^{(\mathsf{op})}, \mathsf{record}_R^{(\mathsf{op})})$.
- $\mathcal{G}_{\mathsf{chip}}$ for $\mathsf{chip} \in \mathsf{Lookup}$ denotes the circuit computing the table items of chip. It inputs the witnesses denoted by $\mathsf{in}^{(\mathsf{chip})}$, and outputs the table item records $(\mathsf{record}_W^{(\mathsf{chip})}, \mathsf{record}_R^{(\mathsf{chip})})$.
- $\mathcal{G}_{\mathsf{unique}}$ is the circuit responsible for processing memory initialization witnesses. It ensures that memory addresses are unique and subsequently outputs the corresponding memory records.
- $\mathcal{G}_{\mathsf{prod}}$ is the building block of the set-equality arguments, as defined in Section 4. $\delta_\star^{(\mathsf{op})}$ denotes the product for the records of $\star \in \{W, R\}$ generated by each op. $\delta_\star^{(\mathsf{chip})}$ denotes the product of records generated by the table circuits, for $\star \in \{W, R\}$.

The prover-provided witnesses are as follows:

- $\mathsf{in}^{(\mathsf{op})}$, as described above, is the input of the circuit $\mathcal{G}^{(\mathsf{op})}$ with the copy number $N^{(\mathsf{op})} = 2^{n^{(\mathsf{op})}}$.
- $\mathsf{in}^{(\mathsf{chip})}$, as described above, is the input of the table generation circuit $\mathcal{G}_{\mathsf{chip}}$. $n^{(\mathsf{chip})}$ is the logarithm of its size.
- $(\mathsf{mem}^{(\mathsf{init})}, \mathsf{in}_{\mathsf{unique}}), \mathsf{mem}^{(\mathsf{finl})}, \mathsf{st}^{(\mathsf{finl})}$ denote the initialization and finalization of the memory and stack required by offline memory checking. $n^{(\mathsf{mem,init})}$, $n^{(\mathsf{mem,finl})}$, $n^{(\mathsf{st,finl})}$ are the logarithm of the number of entries. $\mathsf{in}_{\mathsf{unique}}$ represents the witnesses used to check the uniqueness of memory addresses.

where $n^{(\mathsf{op})}$, $n^{(\mathsf{chip})}$, $n^{(\mathsf{mem,init})}$, $n^{(\mathsf{mem,finl})}$, $n^{(\mathsf{st,finl})}$ are included in the auxiliary information.

We leverage a prover and verifier working together to establish a proof. The underlying relation, $\mathcal{R}_{Ceno\ Basic}$, defines the criteria for this verification process, based on an instance $\mathbb{x} = (\Pi, \mathsf{io})$ as follows.

$$
\mathcal{R}_{\text{Ceno-Basic}} = \tag{7}
$$

$$
\left\{ (\Pi, \mathsf{io}) : \begin{array}{l}
\exists \left\{ (\mathsf{in}^{(\mathsf{op})}, n^{(\mathsf{op})}) \right\}_{\mathsf{op} \in [\![Q]\!]}, \left\{ (\mathsf{in}^{(\mathsf{chip})}, n^{(\mathsf{chip})}) \right\}_{\mathsf{chip} \in \mathsf{Lookup}}, \mathsf{in}^{(\mathsf{bytecode})} = \Pi \ , \\
\exists ((\mathsf{mem}^{(\mathsf{init})}, \mathsf{in}_{\mathsf{unique}}), n^{(\mathsf{mem,init})}), (\mathsf{mem}^{(\mathsf{finl})}, n^{(\mathsf{mem,finl})}), (\mathsf{st}^{(\mathsf{finl})}, n^{(\mathsf{st,finl})}) \ , \\
\mathsf{io} \in \mathsf{st}^{(\mathsf{finl})} \cup \mathsf{mem}^{(\mathsf{finl})} \ , \\
(\mathsf{record}_W^{(\mathsf{op})}, \mathsf{record}_R^{(\mathsf{op})}) \leftarrow \mathcal{G}^{(\mathsf{op})}(\mathsf{in}^{(\mathsf{op})}) \text{ for each } \mathsf{op} \in [\![Q]\!] \ , \\
\delta_\star^{(\mathsf{op})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_\star^{(\mathsf{op})}) \text{ for each } \mathsf{op} \in [\![Q]\!] \text{ and } \star \in \{W, R\} \ , \\
(\mathsf{record}_W^{(\mathsf{chip})}, \mathsf{record}_R^{(\mathsf{chip})}) \leftarrow \mathcal{G}_{\mathsf{chip}}(\mathsf{in}^{(\mathsf{chip})}) \text{ for } \mathsf{chip} \in \mathsf{Lookup}, \\
\delta_\star^{(\mathsf{chip})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_\star^{(\mathsf{chip})}) \text{ for each } \mathsf{chip} \in \mathsf{Lookup}, \star \in \{W, R\} \ , \\
\mathsf{record}_{\mathsf{W}}^{(\mathsf{mem,init})} \leftarrow \mathcal{G}_{\mathsf{unique}}(\mathsf{mem}^{(\mathsf{init})}) \ , \\
\delta_W^{(\mathsf{mem,init})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_{\mathsf{W}}^{(\mathsf{mem,init})}) \ , \\
\delta_R^{(\mathsf{mem,finl})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_{\mathsf{R}}^{(\mathsf{mem,finl})} = \left\{ \mathsf{RLC}\left(\mathsf{mem}^{(\mathsf{finl})}\right) \right\}) \ , \\
\delta_R^{(\mathsf{st,finl})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_{\mathsf{R}}^{(\mathsf{st,finl})} = \left\{ \mathsf{RLC}\left(0..N^{(\mathsf{st,finl})}, \mathsf{st}^{(\mathsf{finl})}.a, \mathsf{st}^{(\mathsf{finl})}.t\right) \right\}) \ , \\
\delta_R^{(\mathsf{mem,finl})} \cdot \delta_R^{(\mathsf{st,finl})} \cdot \left(\prod_{\mathsf{op} \in [\![Q]\!]} \delta_R^{(\mathsf{op})} \right) \cdot \left(\prod_{\mathsf{chip} \in \mathsf{Lookup}} \delta_R^{(\mathsf{chip})} \right) \\
= \delta_W^{(\mathsf{mem,init})} \cdot \left(\prod_{\mathsf{op} \in [\![Q]\!]} \delta_W^{(\mathsf{op})} \right) \cdot \left(\prod_{\mathsf{chip} \in \mathsf{Lookup}} \delta_W^{(\mathsf{chip})} \right) \ .
\end{array} \right\}
$$

We now detail our circuit set with the established model as depicted in Figure 5. An overview of the opcode circuit is presented on the left side of Figure 5, with comprehensive details in Section 6.2. The classification of chip computations and the constraints for generating records within the opcode circuit are introduced in Section 6.2. Further, we discuss memory and stack initialization and finalization processes in Section 6.3, alongside creating table items for lookup operations in chips.

Proof generation for the relation $\mathcal{R}_{Ceno\ Basic}$ unfolds in two stages. Initially, the prover constructs proofs of the computation as the blue texts in Equation 7, including $\mathcal{G}^{(\mathsf{op})}$ for each $\mathsf{op} \in [\![Q]\!]$, $\mathcal{G}_{\mathsf{chip}}$ for chips in Lookup, along with $\mathcal{G}_{\mathsf{prod}}$. Subsequently, the verifier verifies the GKR proofs and PCS batch opening with auxiliary data like $n^{(\mathsf{op})}$, $n^{(\mathsf{mem,init})}$, $n^{(\mathsf{mem,finl})}$, $n^{(\mathsf{chip})}$, ..., alongside the verification of the final outputs of the circuit set, e.g., $\left(\delta_\star^{(\mathsf{op})}\right)_{\star \in \{W,R\}}, \left(\delta_\star^{(\mathsf{chip})}\right)_{\star \in \{W,R\}}$ for each opcode and lookup-based chip and $\delta_W^{(\mathsf{mem,init})}, \delta_R^{(\mathsf{mem,finl})}, \delta_R^{(\mathsf{st,finl})}$, as the black texts in Equation 7. This verification confirms the computations within $\mathcal{R}_{Ceno\ Basic}$. Section 6.4 demonstrates more details of the protocol. We claim the second stage can be transformed into a recursive proving protocol, producing a uniform proof.

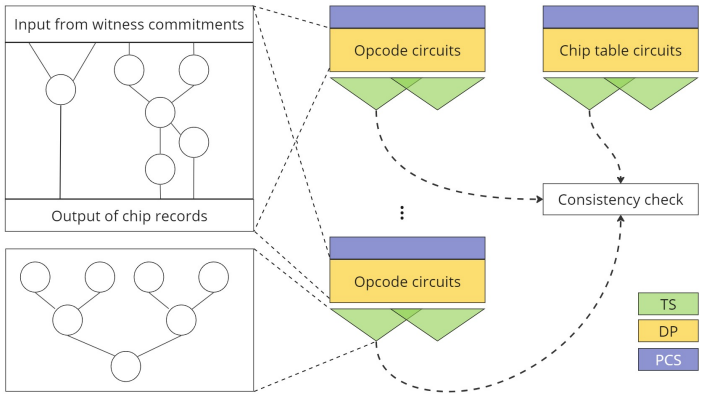


Figure 5: *Ceno Basic* Layout, where $\mathsf{PCS}, \mathsf{DP}, \mathsf{TS}$ denote the polynomial commitment scheme, data-parallel and tree-structured circuits.

6.2 Opcode Circuit Constraints

For completeness, we list all the constraints for the opcode circuit.

Opcode constraint. Suppose an opcode op updates the program from state $(\mathsf{pc}^{(\mathsf{in})}, \mathsf{clk}^{(\mathsf{in})}, \mathsf{top}^{(\mathsf{in})})$ to $(\mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})})$, pops $N^{(\mathsf{pop})}$ values and pushes $N^{(\mathsf{push})}$ values to the stack, accesses memory $N^{(\mathsf{mem})}$ times. First and foremost, we must constrain that the operation defined by the opcode itself is satisfied.

1. **Opcode related constraints.** This includes the correctness checking of $(\mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})})$, stack result computation, and other chip operations.

Program state transition. Our design differentiates itself from the existing approaches in that we choose not to prove opcodes in the sequence in which they are executed. Instead, to ensure soundness, we prove that opcodes are chained together correctly as in the original program.

A state record is formatted as a tuple $s := (\mathsf{pc}, \mathsf{clk}, \mathsf{top})$, representing the program counter, the clock, and the top of the stack. A program state transition will generate two records, $s^{(\mathsf{op},\mathsf{in})} = (\mathsf{pc}^{(\mathsf{in})}, \mathsf{clk}^{(\mathsf{in})}, \mathsf{top}^{(\mathsf{in})})$ and $s^{(\mathsf{op},\mathsf{out})} = (\mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})})$, reflecting how and when the state transition happens.

To show that the state transitions correctly reflect the executions of all opcodes, we must prove that input states $S^{(\mathsf{in})} = \bigcup_{\mathsf{op}\in[\![Q]\!]} \left\{ s_i^{(\mathsf{op},\mathsf{in})} \right\}_{i\in[\![N^{(\mathsf{op})}]\!]}$ matches all output states $S^{(\mathsf{out})} = \bigcup_{\mathsf{op}\in[\![Q]\!]} \left\{ s_i^{(\mathsf{op},\mathsf{out})} \right\}_{i\in[\![N^{(\mathsf{op})}]\!]}$ except for the program's very initial input state $s^{(0)} = (\mathsf{pc} = 0, \mathsf{clk} = 1, \mathsf{top} = 0)$ and the final output

state $s^{(\mathsf{finl})}$, i.e.

$$S^{(\mathsf{in})} \cup \left\{ s^{(\mathsf{finl})} \right\} = S^{(\mathsf{out})} \cup \left\{ s^{(0)} \right\}.$$

The opcode circuit only generates the entry and exit state records mapped into a single field element $('\mathsf{globalstate}', \mathsf{pc}, \mathsf{clk}, \mathsf{top}) \mapsto \mathbb{F}$ by linearly combining pc, clk, top with a random challenge sampled from the transcript.

Explicitly, we have the following two constraints.

2. **Generate state transition records.**

$$\mathsf{record}_{\mathsf{W}}^{(\mathsf{op})} = \mathsf{record}_{\mathsf{W}}^{(\mathsf{op})} \cup \left\{ \mathsf{RLC}\left('\mathsf{globalstate}', \mathsf{pc}^{(\mathsf{in})}, \mathsf{clk}^{(\mathsf{in})}, \mathsf{top}^{(\mathsf{in})}\right) \right\},$$

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{op})} = \mathsf{record}_{\mathsf{R}}^{(\mathsf{op})} \cup \left\{ \mathsf{RLC}\left('\mathsf{globalstate}', \mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})}\right) \right\}.$$

Stack record generation. According to Section 5, stack operations satisfy $\mathcal{R}_{\mathsf{RAM\text{-}RO}}$. The opcode circuit creates two chip record sets to reflect its stack push and pop operations. A stack push record consists of a tuple (a, v, t) where a, v, and t denote the position, value, and timestamp when the push operation happens. A stack pop record also consists of a tuple (a, v, t') where the additional variable t' denotes the timestamp when the value v is pushed into the stack. Like program state records, we use random linear combinations to map stack push and pop records (a, v, t) to field elements.

In the opcode circuit, it's also verified that tuples (a, v, t) and (a, v, t') are well-formed. This includes ensuring the stack address a avoids underflow or overflow, setting the timestamp t to $\mathsf{clk}^{(\mathsf{in})}$ for push operations, and requiring $t' < \mathsf{clk}^{(\mathsf{in})}$ for pop operations. Though out the execution of all the opcodes, we collect two sets $S^{(\mathsf{push})} = \{(a, v, t)_i\}_{i \in [\![N^{(\mathsf{push})}]\!]}$ and $S^{(\mathsf{pop})} = \{(a, v, t')_i\}_{i \in [\![N^{(\mathsf{pop})}]\!]}$.

To sum up, the stack operations consist of the following two constraints.

3. **Generate stack pop records.** Suppose the values from the stack are $v_0^{(\mathsf{pop})}, \ldots, v_{N^{(\mathsf{pop})}-1}^{(\mathsf{pop})}$, written to the stack at the clock $\mathsf{clk}_0^{(\mathsf{pop})}, \ldots, \mathsf{clk}_{N^{(\mathsf{pop})}-1}^{(\mathsf{pop})}$, then the following records are generated. For simplicity, the detailed reduction of range records is omitted here. Readers can refer to Section 5 for further information.

$$\mathsf{record}_{\mathsf{range}}^{(\mathsf{op})} = \mathsf{record}_{\mathsf{range}}^{(\mathsf{op})} \cup \left\{ (0 \le \mathsf{top}^{(\mathsf{in})} - N^{(\mathsf{pop})} < \mathsf{size}^{(\mathsf{stack})}) \right\},$$

$$\mathsf{record}_{\mathsf{range}}^{(\mathsf{op})} = \mathsf{record}_{\mathsf{range}}^{(\mathsf{op})} \cup \left\{ (\mathsf{clk}_i^{(\mathsf{pop})} < \mathsf{clk}^{(\mathsf{in})}) \right\}_{i \in [\![N^{(\mathsf{pop})}]\!]},$$

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{op})} = \mathsf{record}_{\mathsf{R}}^{(\mathsf{op})} \cup \left\{ \mathsf{RLC}\left('\mathsf{stack}', \mathsf{top}^{(\mathsf{in})} - i - 1, v_i^{(\mathsf{pop})}, \mathsf{clk}_i^{(\mathsf{pop})}\right) \right\}_{i \in [\![N^{(\mathsf{pop})}]\!]}.$$

4. **Generate stack push records.** Suppose the values from the stack are $v_0^{(\mathsf{push})}, \ldots, v_{N^{(\mathsf{push})}-1}^{(\mathsf{push})}$, then the following records are generated. We also omit

the details of range checks.

$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{range}} \cup = \left\{ (0 \leq \mathsf{top}^{(\mathsf{in})} - N^{(\mathsf{pop})} + N^{(\mathsf{push})} - 1 < \mathsf{size}^{(\mathsf{stack})}) \right\} ,$$
$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{W}} \cup = \left\{ \mathsf{RLC}\left('\mathsf{stack}', \mathsf{top}^{(\mathsf{in})} - N^{(\mathsf{pop})} + i, v_i^{(\mathsf{push})}, \mathsf{clk}^{(\mathsf{in})}\right) \right\}_{i \in [\![N^{(\mathsf{push})}]\!]} .$$

Memory record generation. According to Section 5, memory operations satisfy $\mathcal{R}_{\mathsf{RAM}}$. A memory store operation will generate a pair of records, simultaneously popping the old data and pushing the new data to the same memory address. The memory load function works identically to a memory store, where we simultaneously pop and push the same data. To sum up, memory operations have two constraints.

5. **Generate memory records (If load)**. Suppose the loaded values are $v_0^{(\mathsf{load})}, \ldots, v_{N^{(\mathsf{mem})}-1}^{(\mathsf{load})}$ with address $a_0^{(\mathsf{mem})}, \ldots, a_{N^{(\mathsf{mem})}-1}^{(\mathsf{mem})}$, written to the memory at the clock $\mathsf{clk}_0^{(\mathsf{load})}, \ldots, \mathsf{clk}_{N^{(\mathsf{mem})}-1}^{(\mathsf{load})}$:

$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{range}} = \mathsf{record}^{(\mathsf{op})}_{\mathsf{range}} \cup \left\{ (\mathsf{clk}_i^{(\mathsf{load})} < \mathsf{clk}^{(\mathsf{in})}) \right\} ,$$
$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{R}} = \mathsf{record}^{(\mathsf{op})}_{\mathsf{R}} \cup \left\{ \mathsf{RLC}\left('\mathsf{memory}', a_i^{(\mathsf{mem})}, v_i^{(\mathsf{load})}, \mathsf{clk}_i^{(\mathsf{load})}\right) \right\}_{i \in [\![N^{(\mathsf{mem})}]\!]} ,$$
$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{W}} = \mathsf{record}^{(\mathsf{op})}_{\mathsf{W}} \cup \left\{ \mathsf{RLC}\left('\mathsf{memory}', a_i^{(\mathsf{mem})}, v_i^{(\mathsf{load})}, \mathsf{clk}^{(\mathsf{in})}\right) \right\}_{i \in [\![N^{(\mathsf{mem})}]\!]} .$$

6. **Generate memory records (If store).** Suppose the stored values are $v_0^{(\mathsf{store})}, \ldots, v_{N^{(\mathsf{mem})}-1}^{(\mathsf{store})}$ with addresses $a_0^{(\mathsf{mem})}, \ldots, a_{N^{(\mathsf{mem})}-1}^{(\mathsf{mem})}$, overwrite the values $v_0^{(\mathsf{load})}, \ldots, v_{N^{(\mathsf{mem})}-1}^{(\mathsf{load})}$ written to the memory at $\mathsf{clk}_0^{(\mathsf{load})}, \ldots, \mathsf{clk}_{N^{(\mathsf{mem})}-1}^{(\mathsf{load})}$:

$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{range}} = \mathsf{record}^{(\mathsf{op})}_{\mathsf{range}} \cup \left\{ (\mathsf{clk}_i^{(\mathsf{load})} < \mathsf{clk}^{(\mathsf{in})}) \right\} ,$$
$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{R}} = \mathsf{record}^{(\mathsf{op})}_{\mathsf{R}} \cup \left\{ \mathsf{RLC}\left('\mathsf{memory}', a_i^{(\mathsf{mem})}, v_i^{(\mathsf{load})}, \mathsf{clk}_i^{(\mathsf{load})}\right) \right\}_{i \in [\![N^{(\mathsf{mem})}]\!]} ,$$
$$\mathsf{record}^{(\mathsf{op})}_{\mathsf{W}} = \mathsf{record}^{(\mathsf{op})}_{\mathsf{W}} \cup \left\{ \mathsf{RLC}\left('\mathsf{memory}', a_i^{(\mathsf{mem})}, v_i^{(\mathsf{store})}, \mathsf{clk}^{(\mathsf{in})}\right) \right\}_{i \in [\![N^{(\mathsf{mem})}]\!]} .$$

Chip lookup record generation. Lookup arguments, which verify $S \subseteq T$ given sets S and T, effectively address complex and non-linear operations. According to Section 5, lookup operations satisfy $\mathcal{R}_{\mathsf{Lookup}}$. In zkVM, several chips are effectively represented using lookup arguments. Examples include batched bytecode verification, which entails justifying correct opcode retrieval by demonstrating $(\mathsf{pc}, \mathsf{opcode}) \in \{(i, \Pi[i])\}_{i \in [\![m_\Pi]\!]}$, and batched range checking, which involves ensuring a value x falls within a specific range, $x \in [0, \mathsf{MAX})$. A bytecode chip serves as an illustrative case, with its constraints outlined as follows:

7. **Generate bytecode input records.** With a witness t,

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{op})} = \mathsf{record}_{\mathsf{R}}^{(\mathsf{op})} \cup \left\{\mathsf{RLC}\left('\mathsf{bytecode}', t, \mathsf{pc}^{(\mathsf{in})}, \mathsf{op}\right)\right\} ,$$
$$\mathsf{record}_{\mathsf{W}}^{(\mathsf{op})} = \mathsf{record}_{\mathsf{W}}^{(\mathsf{op})} \cup \left\{\mathsf{RLC}\left('\mathsf{bytecode}', t+1, \mathsf{pc}^{(\mathsf{in})}, \mathsf{op}\right)\right\} .$$

6.3 Chip Computation Constraints

Memory initialization and finalization constraints. As aforementioned in Section 5.1, we initialize the memory cells for a subset of address touched during the program execution. Assuming $N^{(\mathsf{mem},\mathsf{init})}$ memory cells with addresses $\mathsf{mem}^{(\mathsf{init})} = a_0^{(\mathsf{mem},\mathsf{init})}, \ldots, a_{N^{(\mathsf{mem},\mathsf{init})}-1}^{(\mathsf{mem},\mathsf{init})}$ need initialization, then $\mathsf{record}_{\mathsf{W}}^{(\mathsf{mem},\mathsf{init})}$ is defined as follows:

– **Memory initialization.**

$$a_i^{(\mathsf{mem},\mathsf{init})} < a_{i+1}^{(\mathsf{mem},\mathsf{init})}, \text{ for } i \in [\![N^{(\mathsf{mem},\mathsf{init})} - 1]\!] ,$$
$$\mathsf{record}_{\mathsf{W}}^{(\mathsf{mem},\mathsf{init})} = \left\{\mathsf{RLC}\left('\mathsf{memory}', a_i^{(\mathsf{mem},\mathsf{init})}, 0, 0\right)\right\}_{i \in [\![N^{(\mathsf{mem},\mathsf{init})}]\!]} .$$

Additionally, memory finalization in our zkVM involves clearing all touched addresses by opcodes. If there are $N^{(\mathsf{mem},\mathsf{finl})}$ cells at $a_0^{(\mathsf{mem},\mathsf{finl})}, \ldots, a_{N^{(\mathsf{mem},\mathsf{finl})}-1}^{(\mathsf{mem},\mathsf{finl})}$ with values $v_0, \ldots, v_{N^{(\mathsf{mem},\mathsf{finl})}-1}$ to be cleared, finally written at the clocks of $\mathsf{clk}_0^{(\mathsf{mem},\mathsf{finl})}, \ldots, \mathsf{clk}_{N^{(\mathsf{mem},\mathsf{finl})}-1}^{(\mathsf{mem},\mathsf{finl})}$, respectively. The witnesses $\mathsf{mem}^{(\mathsf{finl})} = \left(a_0^{(\mathsf{mem},\mathsf{finl})}, v_0, \mathsf{clk}_0^{(\mathsf{mem},\mathsf{finl})}\right), \ldots, \left(a_{N^{(\mathsf{mem},\mathsf{finl})}-1}^{(\mathsf{mem},\mathsf{finl})}, v_{N^{(\mathsf{mem},\mathsf{finl})}-1}, \mathsf{clk}_{N^{(\mathsf{mem},\mathsf{finl})}-1}^{(\mathsf{mem},\mathsf{finl})}\right)$, and $\mathsf{record}_{\mathsf{R}}^{(\mathsf{mem},\mathsf{finl})}$ is defined as follows:

– **Memory finalization.**

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{mem},\mathsf{finl})} = \left\{\mathsf{RLC}\left('\mathsf{memory}', a_i^{(\mathsf{mem},\mathsf{finl})}, v_i, \mathsf{clk}_i^{(\mathsf{mem},\mathsf{finl})}\right)\right\}_{i \in [\![N^{(\mathsf{mem},\mathsf{finl})}]\!]} . \tag{8}$$

Stack initialization and finalization constraints. Things are inherently simpler for the stack, given that the stack is initially empty. Upon finalization after program halt, with $N^{(\mathsf{st},\mathsf{finl})}$ values $v_0, \ldots, v_{N^{(\mathsf{st},\mathsf{finl})}-1}$ to be cleared, their addresses naturally range from 0 to $N^{(\mathsf{st},\mathsf{finl})} - 1$, finally accessed at $\mathsf{clk}_0^{(\mathsf{st},\mathsf{finl})}, \ldots, \mathsf{clk}_{N^{(\mathsf{st},\mathsf{finl})}-1}^{(\mathsf{st},\mathsf{finl})}$. Consequently, the witnesses $\mathsf{st}^{(\mathsf{finl})} = \left(v_0, \mathsf{clk}_0^{(\mathsf{st},\mathsf{finl})}\right), \ldots, \left(v_{N^{(\mathsf{st},\mathsf{finl})}-1}, \mathsf{clk}_{N^{(\mathsf{st},\mathsf{finl})}-1}^{(\mathsf{st},\mathsf{finl})}\right)$, and$\mathsf{record}_{\mathsf{R}}^{(\mathsf{st},\mathsf{finl})}$ is defined as follows:

– **Stack finalization.**

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{st},\mathsf{finl})} = \left\{\mathsf{RLC}\left('\mathsf{stack}', i, v_i, \mathsf{clk}_i^{(\mathsf{st},\mathsf{finl})}\right)\right\}_{i \in [\![N^{(\mathsf{st},\mathsf{finl})}]\!]} . \tag{9}$$

Table constraints. A circuit is required to initialize table items and create the respective table records for each chip verified through lookup arguments. For instance, in the case of the bytecode chip and its table record set $\mathsf{recordbytecode}, T$, the circuit generates the following records:

- **Generate bytecode table records.** With a witness $t_0, \ldots, t_{m_\Pi - 1}$,

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{chip})} = \left\{\mathsf{RLC}\left('\mathsf{bytecode}', 0, i, \mathsf{op}_i\right)\right\}_{i \in [\![m_\Pi]\!]},$$
$$\mathsf{record}_{\mathsf{W}}^{(\mathsf{chip})} = \left\{\mathsf{RLC}\left('\mathsf{bytecode}', t_i, i, \mathsf{op}_i\right)\right\}_{i \in [\![m_\Pi]\!]}.$$

  where m_Π is the length of the program.

where m_i is the number that (i, op_i) appears in the execution.

Additionally, the output of each chip in each opcode circuit is connected with a tree-structured circuit, depicted as green triangles in Figure 5, which calculates the product or fraction summation for record items. Subsequently, a similar computation is executed following the chip table circuit by a tree circuit. All the products and fraction summations will be forwarded to the verifier.

6.4 Proving and Verification Protocols

The detailed description of the constraint system has already been presented in Section 6.2. The circuit structure and proving process are presented in Figure 6 and Protocol 3, separately.

Proving cost analysis. Assume the summation of execution times for all opcodes in the program is N times, then

- The circuit size and the running time of the prover are $O(N)$ due to the linear-time prover of GKR.
- The running time of the verifier, proof size, and the number of rounds are both $O(\log^2 N)$, coming from the sumcheck verification protocol on the tree-structure circuits with $O(\log N)$ layers. In the most recent paper, Bünz et al. [14] proposed a variant GKR protocol with a higher verification cost and a larger proof size but fewer rounds.

Note that recursion proof, a common practice in the zero-knowledge community, can further reduce the verification and proof size.

7 *Ceno Pro*

The current arithmetic system mainly handles static computations like circuits. *Ceno Basic* capitalizes on the fixed computational patterns of each opcode, transforming them into data-parallel circuits. However, We observe more fixed patterns within a program. When a program segment lacks branches, the control and data flow within this segment remain consistent. Based on this insight, we

Protocol 3 (*Ceno Basic*) *The proving-verification process proceeds as follows, where we use the same notation as in Equation 7:*

- ***Prover:***
  1. *Prover sends auxiliary information to the verifier, including* $\left\{n^{(\mathsf{op})}\right\}_{\mathsf{op}\in[\![Q]\!]}$, $\left\{n^{(\mathsf{chip})}\right\}_{\mathsf{chip}\in\mathsf{Lookup}}$, $n^{(\mathsf{mem,init})}$, $n^{(\mathsf{mem,finl})}$ *and* $n^{(\mathsf{st,finl})}$.
  2. *Prover and verifier exchange witness commitments and challenges in several rounds, including commitments for* $\left\{\mathsf{in}^{(\mathsf{op})}\right\}_{\mathsf{op}\in[\![Q]\!]}$, $\left\{\mathsf{in}^{(\mathsf{chip})}\right\}_{\mathsf{chip}\in\mathsf{Lookup}}$, $(\mathsf{mem}^{(\mathsf{init})}, \mathsf{in}_{\mathsf{unique}})$, $\mathsf{mem}^{(\mathsf{finl})}$, *and* $\mathsf{st}^{(\mathsf{finl})}$.
  3. *Prover sends* $\left\{\delta_{\star}^{(\mathsf{op})}\right\}_{\mathsf{op}\in[\![Q]\!],\star\in\{W,R\}}$ *generated by the opcode circuits,* $\left\{\delta_{\star}^{(\mathsf{chip})}\right\}_{\mathsf{chip}\in\mathsf{Lookup},\star\in\{W,R\}}$ *generated by lookup table circuits,* $\delta_W^{(\mathsf{mem,init})}$, $\delta_R^{(\mathsf{mem,finl})}$ *and* $\delta_R^{(\mathsf{st,finl})}$ *to the verifier. In addition, the prover generates all GKR proofs for the circuits, following the flow as shown in Figure 6 and sends the proofs to the verifier.*
  4. *Prover opens polynomial commitments and sends the openings to the verifier.*
- ***Verifier:***
  1. *Verifier receives the auxiliary information.*
  2. *Verifier exchanges witness commitments and challenges with the prover in several rounds.*
  3. *Verifier receives GKR proofs and verifies them.*
  4. *Verifier also receives polynomial commitment openings and verifies them.*
  5. *Verifier checks the correctness of*

$$\begin{aligned}&\delta_R^{(\mathsf{mem,finl})}\cdot\delta_R^{(\mathsf{st,finl})}\cdot\left(\prod_{\mathsf{op}\in[\![Q]\!]}\delta_R^{(\mathsf{op})}\right)\cdot\left(\prod_{\mathsf{chip}\in\mathsf{Lookup}}\delta_R^{(\mathsf{chip})}\right)\\&=\delta_W^{(\mathsf{mem,init})}\cdot\left(\prod_{\mathsf{op}\in[\![Q]\!]}\delta_W^{(\mathsf{op})}\right)\cdot\left(\prod_{\mathsf{chip}\in\mathsf{Lookup}}\delta_W^{(\mathsf{chip})}\right)\ .\end{aligned}$$

utilize the concept of "basic blocks" from compiler design. In *Ceno Pro*, we exploit the consistent pattern of basic blocks. By defining data-parallel circuits at the basic block level, we make the program more closely resemble the structure of circuits, thereby improving the first-stage proving time.

Now, we are ready to present the details of this improvement. We note that throughout this section, we use stack-based virtual machines as examples, though our method also applies to register-based virtual machines.

7.1 Critical Observations in Program Execution

In this subsection, we start with some observations of stacks and basic blocks.

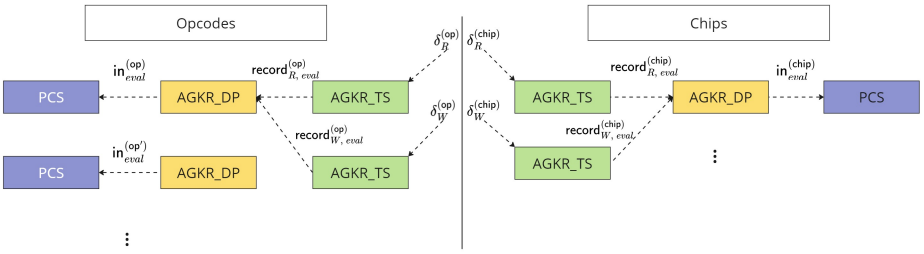


Figure 6: Circuit Structure in *Ceno Basic*

Our view of stack operations Generally speaking, a stack operation connects a value between two opcodes, where it is generated and then consumed. Consider this example: In an opcode execution sequence $\Pi := (\ldots, \mathsf{ADD}, \mathsf{MUL}, \ldots)$, one of the values popped in MUL is the result that was pushed in ADD. If we envision the opcodes as gates in a circuit, the stack operations act as the wires that link the gates. For clarity, we refer to the opcode that generates the stack element as the *generator* (i.e., the ADD gate) and the one that consumes the stack element as the *consumer*. Extending this notion over the whole trace, we can construct a circuit for a segment of program Π that lacks branches.

There is a concept known as *basic block* (BB) in the compiler design domain.

Definition 3 (Basic block [62]). *The code in a basic block has the following properties:*

- *One entry point, meaning that no code within it is the destination of a jump instruction anywhere in the program.*
- *One exit point, meaning that only the last instruction can cause the program to begin executing code in a different basic block.*

Under these circumstances, whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once and in order.

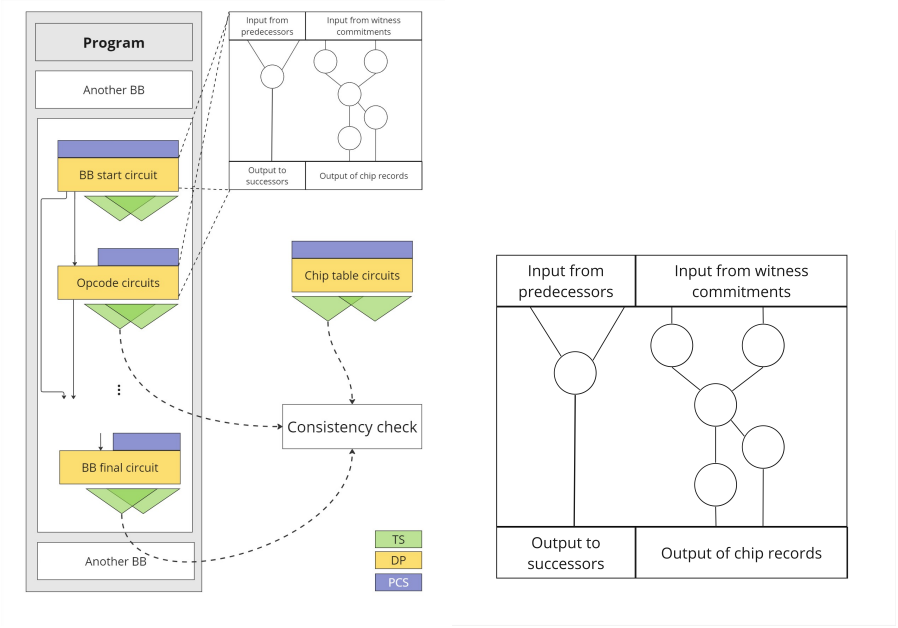
Consequently, a program can be divided into a list of basic blocks. From these, we can construct a control-flow graph in which each basic block forms a vertex, and directed edges illustrate the potential paths in the control flow. Program execution is then represented by a walk in this graph, starting at the entry block.

7.2 Overview

In this section, we demonstrate an overview of *Ceno Pro*. We model the program as a list of basic blocks, or $\Pi := (\mathsf{BB}_0, \ldots, \mathsf{BB}_{m_{\mathsf{BB}}-1})$. For a basic block BB, it is a sequence of opcodes, or $\mathsf{BB} = (\mathsf{op}_0, \ldots, \mathsf{op}_{m_{\mathsf{op}}^{(\mathsf{BB})}-1}) \in [\![Q]\!]^{m_{\mathsf{op}}^{(\mathsf{BB})}}$. We design

Ceno Pro to prove a program Π's execution based on given inputs and outputs. Similar to *Ceno Basic*, the framework of *Ceno Pro* is also a circuit set.

We now provide details of the *Ceno Pro* circuit set, building upon *Ceno Basic*. As illustrated in Figure 7, all circuits occurring in the circuit set are similar to the *Ceno Basic* shown in Figure 5, except that the opcode data-parallel circuits have been replaced with basic block data-parallel circuits. We denote the basic block circuit as $\mathcal{G}^{(\mathsf{BB})}$. On the left side, Figure 7a displays the basic block. Each basic block circuit consists of a BB start circuit, multiple non-stack opcode circuits, and a BB final circuit, denoted by $\mathsf{C}^{(\mathsf{BB})} = \{\mathsf{BB.start}, \mathsf{BB.final}\} \cup \left\{\mathcal{G}^{(\mathsf{BB.op}_i)}\right\}_{i \in [\![m_{\mathsf{op}}^{(\mathsf{BB})}]\!]}$, corresponding to the prover-provided witnesses $\left\{\mathsf{in}^{(\mathcal{G})}\right\}_{\mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}}$ and the output $\left\{(\mathsf{record}_W^{(\mathcal{G})}, \mathsf{record}_R^{(\mathcal{G})})\right\}_{\mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}}$. Additionally, every circuit, except for the BB start, receives stack outputs from its predecessors, and every circuit, except for the BB final, forwards stack outputs to its successors. These connections are determined by stack operations. The layout of each opcode is further illustrated in Figure 7b.



(a) *Ceno Pro* zkVM layout (b) *Ceno Pro* opcode layout

Figure 7: *Ceno Pro* Layouts

For a given input, a program can generate an execution trace. Each basic block BB possesses a copy number $N^{(\mathsf{BB})}$, expressed logarithmically as $n^{(\mathsf{BB})}$.

We define the underlying relation, $\mathcal{R}_{Ceno\ Pro}$, as the set of accepted instances in the form $\mathbb{x} = (\Pi, \mathsf{io})$, as detailed in Equation 10.

$$
\mathcal{R}_{\mathsf{Ceno\text{-}Pro}} = \left\{ (\Pi, \mathsf{io}) : \begin{array}{l}
\exists \left\{ \left(\left\{ \mathsf{in}^{(\mathcal{G})} \right\}_{\mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}}, n^{(\mathsf{BB})} \right) \right\}_{\mathsf{BB} \in \Pi}, \left\{ (\mathsf{in}^{(\mathsf{chip})}, n^{(\mathsf{chip})}) \right\}_{\mathsf{chip} \in \mathsf{Lookup}}, \\
\exists (\mathsf{mem}^{(\mathsf{init})}, n^{(\mathsf{mem,init})}), (\mathsf{mem}^{(\mathsf{finl})}, n^{(\mathsf{mem,finl})}), (\mathsf{st}^{(\mathsf{finl})}, n^{(\mathsf{st,finl})}), \\
\mathsf{io} \in \mathsf{st}^{(\mathsf{finl})} \cup \mathsf{mem}^{(\mathsf{finl})}, \mathsf{in}_{\mathsf{bytecode}} = \Pi, \\
\left\{ (\mathsf{record}_W^{(\mathcal{G})}, \mathsf{record}_R^{(\mathcal{G})}) \right\}_{\mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}} \leftarrow \mathcal{G}^{(\mathsf{BB})} \left(\left\{ \mathsf{in}^{(\mathcal{G})} \right\}_{\mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}} \right) \forall \mathsf{BB} \in \Pi, \\
\delta_\star^{(\mathcal{G})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_\star^{(\mathcal{G})}), \text{ for each } \mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}, \star \in \{W, R\}, \mathsf{BB} \in \Pi \\
(\mathsf{record}_W^{(\mathsf{chip})}, \mathsf{record}_R^{(\mathsf{chip})}) \leftarrow \mathcal{G}_{\mathsf{chip}}(\mathsf{in}^{(\mathsf{chip})}) \text{ for each } \mathsf{chip} \in \mathsf{Lookup}, \\
\delta_\star^{(\mathsf{chip})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_\star^{(\mathsf{chip})}) \text{ for each } \mathsf{chip} \in \mathsf{Lookup}, \star \in \{W, R\}, \\
\mathsf{record}_{\mathsf{W}}^{(\mathsf{mem,init})} \leftarrow \mathcal{G}_{\mathsf{unique}}(\mathsf{mem}^{(\mathsf{init})}), \\
\delta_W^{(\mathsf{mem,init})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_{\mathsf{W}}^{(\mathsf{mem,init})}), \\
\delta_R^{(\mathsf{mem,finl})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_{\mathsf{R}}^{(\mathsf{mem,finl})} = \left\{ \mathsf{RLC}\left(\mathsf{mem}^{(\mathsf{finl})} \right) \right\}), \\
\delta_R^{(\mathsf{st,finl})} \leftarrow \mathcal{G}_{\mathsf{prod}}(\mathsf{record}_{\mathsf{R}}^{(\mathsf{st,finl})} = \left\{ \mathsf{RLC}\left(0..N^{(\mathsf{st,finl})}, \mathsf{st}^{(\mathsf{finl})}.a, \mathsf{st}^{(\mathsf{finl})}.t \right) \right\}), \\
\delta_R^{(\mathsf{mem,finl})} \cdot \delta_R^{(\mathsf{st,finl})} \cdot \left(\prod_{\mathsf{BB} \in \Pi} \prod_{\mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}} \delta_R^{(\mathcal{G})} \right) \cdot \left(\prod_{\mathsf{chip} \in \mathsf{Lookup}} \delta_R^{(\mathsf{chip})} \right) \\
= \delta_W^{(\mathsf{mem,init})} \cdot \left(\prod_{\mathsf{BB} \in \Pi} \prod_{\mathcal{G} \in \mathsf{C}^{(\mathsf{BB})}} \delta_W^{(\mathcal{G})} \right) \cdot \left(\prod_{\mathsf{chip} \in \mathsf{Lookup}} \delta_W^{(\mathsf{chip})} \right),
\end{array} \right\} \tag{10}
$$

Proof generation for the relation $\mathcal{R}_{Ceno\ Pro}$ also unfolds two stages as in *Ceno Basic*. The prover constructs proofs of the computation as the blue texts in Equation 10, and the verifier verifies those proofs, together with checking the other constraints in the relation. We omit the other details but notice that the only difference is $\mathcal{G}^{(\mathsf{op})}$ is replaced with $\mathcal{G}^{(\mathsf{BB})}$. Using the following toy example, we illustrate the layout of $\mathcal{G}^{(\mathsf{BB})}$ with more details and demonstrate how to prove and verify it.

A toy example. Figure 8 presents a toy example. This example program, shown in Figure 8a, reads an input integer n and uses a loop with n iterations to compute the sum of a variable-length array. After compiling this program into assembly, as depicted in Figure 8b, we identify three basic blocks. The first block (B1) includes the first two opcodes, initializing the summation and loop iterator. The second block (B2), beginning with the third opcode and extending to the `CJMP` opcode, forms the loop body and increments the summation. The final block (B3), containing just the last opcode, cleans up by removing the iterator and retaining the summation result. Upon completion of B1, the transition to

B2 is automatic, while B2 can loop back to itself or proceed to B3, depending on the branch. The control-flow graph is outlined in Figure 8c.

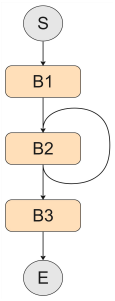
```
int n = input();

int sum = 0;
for(int i = n; i > 0; i--) {
    sum += a[i];
}
```

(a) Toy program

```
B1: PUSH 0   # [sum]
    PUSH n   # [sum, n]
B2: DUP      # [sum, i, i]
    LOAD     # [sum, i, a[i]]
    SWAP1    # [sum, a[i], i]
    SWAP2    # [i, a[i], sum]
    ADD      # [i, sum']
    SWAP1    # [sum', i]
    SUB 1    # [sum', i']
    DUP      # [sum', i', i']
    NZ       # [sum', i',
               is_zero(i')]
    CJMP B2 # [sum', i']
B3: POP
```

(b) Toy program in assembly



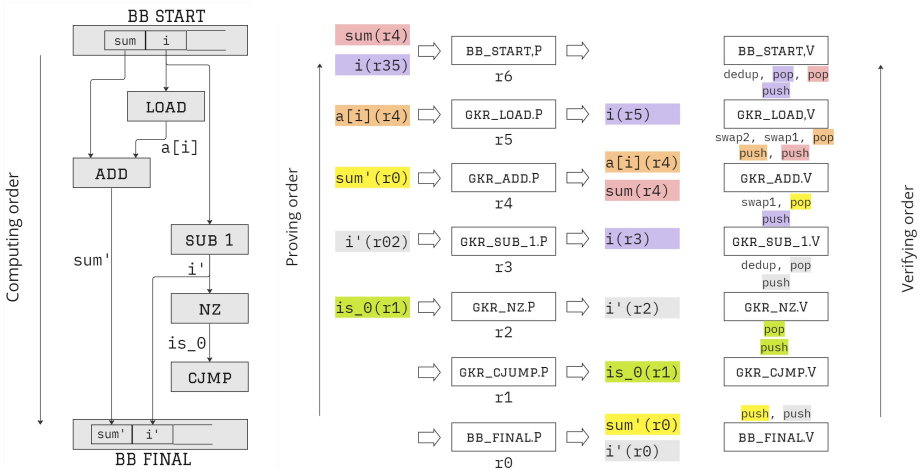
(c) Basic Block

Figure 8: Program and its Basic Block for the Toy Example

We define the circuit by representing each non-stack operation as a gate within the circuit. All values popped from the stack form the fan-ins to the gate, while values pushed onto the stack form the fan-outs. We then connect each stack value from its generator to its consumer, as determined by stack operations. This setup is depicted in Figure 9a, where the top part of the figure represents the BB start circuit, initializing stack values generated externally and consumed within the current block. The lower part of the figure shows the BB final circuit, which finalizes stack values to be consumed externally. Looking ahead, we manage a global stack to facilitate value transfer between basic blocks. Both the BB start and BB final circuits interact with the global stack, fetching and returning values, respectively.

To generate a proof for the B2 circuit using the GKR protocol, we begin at the BB final circuit and proceed backward to the BB start circuit. For each circuit, the GKR protocol confirms the correctness of the output evaluations—some entries of which are the pushed stack values—by reducing them to the evaluations of the input segments, which contain the popped stack values. Subsequently, these input evaluations are distributed to preceding gates based on the wire connections. The left section of Figure 9b illustrates this proving process, specifically highlighting the evaluations of stack values. The values on the left represent the evaluations of the outputs, while those on the right show the corresponding reduced evaluations of the inputs.

To verify the proof generated by the process, the verifier needs to know the circuit structure represented by GKR arithmetic, which indicates the evaluation transmission across layers. Under our assumption, however, the verifier is only aware of the opcode sequence for B2. To facilitate the recovery of wire connections, we equip the verifier with a stack. The verifier then scans the opcode



(a) Computing B2 Circuits

(b) Prove and Verify B2 Circuits. The terms r0, r1, etc., represent the random points used in the final step of each GKR protocol. r02 denotes the random point applied in the sum-check protocol to merge sum'(r0) and sum'(r2); similarly, r35 is used for corresponding merges.

Figure 9: Prove and Verify Basic Block in the Toy Example

sequence in reverse while simultaneously progressing through the proving process. Whenever a stack operation is encountered, the verifier applies its inverse operation to the corresponding stack value evaluation in the proof. This method allows the verifier to effectively reconstruct the connections between gates.

Additionally, since the stack operations are applied to the evaluations in the proof, each can be performed only once. To show the efficiency of this method, in our toy example, B2 includes 15 stack operations—both explicit stack opcodes and those embedded within non-stack opcodes. If B2 is executed 100 times, this results in a total of 1500 stack operations. However, in our approach, the prover performs only 2 pops and 2 pushes in the BB start and BB final circuits on the global stack, totaling 400 operations. The verifier, in contrast, only needs to perform 19 operations. This significant reduction from 1600 to 419 (400 plus 19) operations demonstrates how our method can substantially decrease the number of stack operations represented in the circuit.

7.3 Basic Block Circuit Constraints

For completeness, we list the constraints of the basic blocks in *Ceno Pro*.

Opcode-level constraints *Ceno Pro* improves the opcode-level constraints upon *Ceno Basic* in the following aspects:

- Firstly, the opcode circuits within a basic block no longer manage stack operations, nor do they check the range of the stack top or the timestamp.
- Secondly, the opcode circuits no longer require bytecode lookups, as the verifier simultaneously scans the program when verifying the proof of the basic blocks. The only exception occurs with the last opcode if it is a jump operation; some instruction set architectures mandate that the destination must be a specific type of opcode (e.g., JUMPDEST in EVM opcodes).
- Thirdly, the opcode circuit no longer handles global states. Global state records are only generated at the beginning and end of a basic block.

Basic-block-level constraints. The basic block start circuits and final circuits are used to assert the validity of stack operations and global state updates. Specifically:

- The BB start circuit retrieves all global stack values that will be accessed by the opcodes within this basic block. The quantity and positions relative to the stack top are consistent across all executions.
- The BB final circuit takes the stack values produced by the internal opcodes and updates them back to the global stack, with their number and relative positions remaining fixed.
- The BB start circuit also generates an input global state record represented by $(\mathsf{pc}^{(\mathsf{in})}, \mathsf{clk}^{(\mathsf{in})}, \mathsf{top}^{(\mathsf{in})})$.
- Similarly, the BB final circuit computes and generates an output global state record $(\mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})})$.

Suppose a basic block BB updates the program from state $(\mathsf{pc}^{(\mathsf{in})}, \mathsf{clk}^{(\mathsf{in})}, \mathsf{top}^{(\mathsf{in})})$ to $(\mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})})$, pops $N^{(\mathsf{pop})}$ values at the beginning and pushes $N^{(\mathsf{push})}$ values to the stack in the end, then it has the following constraints (we omit the details of range argument):

1. **Generate state transition records.** The BB start circuit generates a state record:

$$\mathsf{record}_{\mathsf{W}}^{(\mathsf{BB})} = \mathsf{record}_{\mathsf{W}}^{(\mathsf{BB})} \cup \left\{ \mathsf{RLC}\left('\mathsf{globalstate}', \mathsf{pc}^{(\mathsf{in})}, \mathsf{clk}^{(\mathsf{in})}, \mathsf{top}^{(\mathsf{in})}\right) \right\} ,$$

   BB final circuit computes $(\mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})})$ based on $\mathsf{pc}^{(\mathsf{in})}, \mathsf{clk}^{(\mathsf{in})}, \mathsf{top}^{(\mathsf{in})}$ according to the BB's structure, and then generates a state record:

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{BB})} = \mathsf{record}_{\mathsf{R}}^{(\mathsf{BB})} \cup \left\{ \mathsf{RLC}\left('\mathsf{globalstate}', \mathsf{pc}^{(\mathsf{out})}, \mathsf{clk}^{(\mathsf{out})}, \mathsf{top}^{(\mathsf{out})}\right) \right\} .$$

2. **Initialization of the stack.** The BB start circuit loads values from the global stack and exports them as outputs, which are then fed into the corresponding opcode circuits within the block. Denote the values from the stack by $v_0^{(\mathsf{pop})}, \ldots, v_{N^{(\mathsf{pop})}-1}^{(\mathsf{pop})}$ and let $\mathsf{clk}_0^{(\mathsf{pop})}, \ldots, \mathsf{clk}_{N^{(\mathsf{pop})}-1}^{(\mathsf{pop})}$ be the clock that stack

is written, respectively. Then,

$$\mathsf{record}_{\mathsf{range}}^{(\mathsf{BB.start})} = \mathsf{record}_{\mathsf{range}}^{(\mathsf{BB.start})} \cup \left\{ (0 \leq \mathsf{top}^{(\mathsf{in})} - N^{(\mathsf{pop})} < \mathsf{size}^{(\mathsf{stack})}) \right\} ,$$

$$\mathsf{record}_{\mathsf{range}}^{(\mathsf{BB.start})} = \mathsf{record}_{\mathsf{range}}^{(\mathsf{BB.start})} \cup \left\{ (\mathsf{clk}_i^{(\mathsf{pop})} < \mathsf{clk}^{(\mathsf{in})}) \right\}_{i \in [\![N^{(\mathsf{pop})}]\!]} ,$$

$$\mathsf{record}_{\mathsf{R}}^{(\mathsf{BB.start})} \cup = \left\{ \mathsf{RLC}\left(\mathit{'stack'}, \mathsf{top}^{(\mathsf{in})} - i - 1, v_i^{(\mathsf{pop})}, \mathsf{clk}_i^{(\mathsf{pop})} \right) \right\}_{i \in [\![N^{(\mathsf{pop})}]\!]} .$$

3. The BB final circuit takes values as the input that are computed within the block and not yet consumed. It writes back those values to the stack. Denote this values by $v_0^{(\mathsf{push})}, \ldots, v_{N^{(\mathsf{push})}-1}^{(\mathsf{push})}$, then:

$$\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \left\{ (0 \leq \mathsf{top}^{(\mathsf{in})} - N^{(\mathsf{pop})} + N^{(\mathsf{push})} - 1 < \mathsf{size}^{(\mathsf{stack})}) \right\},$$

$$\mathsf{record}_{\mathsf{W}}^{(\mathsf{BB.final})} \cup = \left\{ \mathsf{RLC}\left(\mathit{'stack'}, \mathsf{top}^{(\mathsf{in})} - N^{(\mathsf{pop})} + i, v_i^{(\mathsf{push})}, \mathsf{clk}^{(\mathsf{in})} \right) \right\}_{i \in [\![N^{(\mathsf{push})}]\!]} .$$

We have listed the basic block constraints above. For other parts of the circuit set, including the other chip operations within each opcode and chip table computation constraints, they remain identical to those in *Ceno Basic*.

7.4 Proving And Verification Protocols

The complete protocol is detailed in Protocol 4, and the prover's workflow is illustrated in Figure 10, forming a non-uniform protocol.

During verification, it is assumed that the verifier is familiar with the program Π, which comprises a list of basic blocks. To verify the non-uniform proof, the verifier scans the opcodes of each basic block in reverse order while simultaneously verifying the proofs of each block to deduce the layout from the opcode sequence. As outlined in Section 7.2, upon encountering a new basic block, the verifier begins with an empty stack, applying inverse operations to stack operations to transmit stack value evaluations from their consumer to the generator. After completing the verification of a block, the verifier checks to ensure the stack is empty. The process concludes with the verifier examining the additional constraints depicted as black text in Equation 10. Ultimately, the entire verification protocol is adaptable to a recursive proving framework.

Cost analysis. *Ceno Pro* is a structure-aware protocol optimized for specific program structures. Consider a program of size m_Π, composed of m_{BB} basic blocks. When the total execution times for all basic blocks are no more than M times, and the number of stack operations in the BB start and BB final circuits are bounded by a constant, the prover needs to demonstrate only $O(M)$ stack operations. This optimization also reduces the integrity checks for the stack to $O(M)$, which includes validating the stack top pointer and comparing the stack timestamp with the current clock. *Ceno Pro* becomes markedly more efficient when M is substantially less than N, the total number of opcode execution

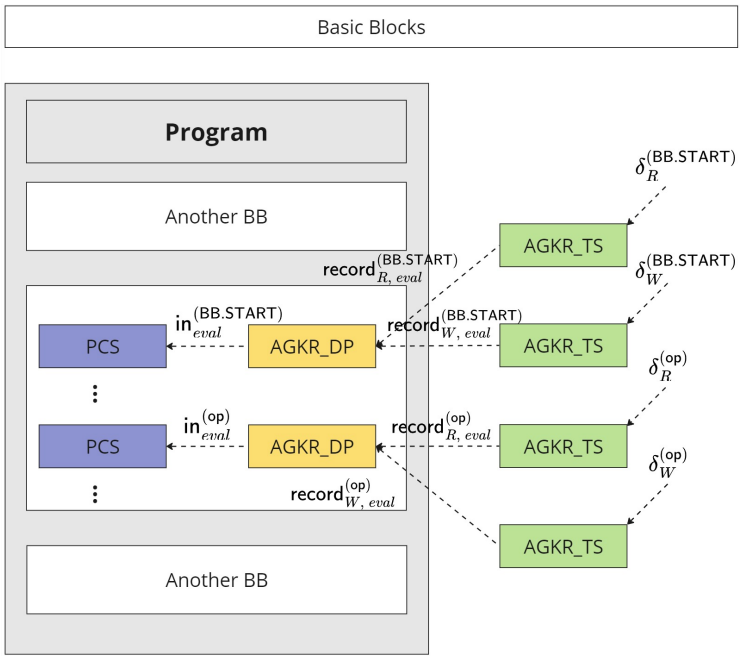


Figure 10: Structure of Basic Blocks in *Ceno Pro*

times. We assert that this is typically the case in real-world programs, as a basic block usually comprises multiple opcodes.

However, this reduction in cost for the prover corresponds with a slight increase in the verifier's workload. Specifically, the stack operations formerly handled by the opcode circuits are now transferred to the verifier, to be processed once for each op_i in Π, irrespective of repetition frequency. As a result, the verifier's workload potentially includes $O(m_{\mathsf{BB}} + m_{\Pi})$ stack operations. Note that the total number of stack operations for both the prover and verifier is reduced to $O(M + m_{\mathsf{BB}} + m_{\Pi})$, which is significantly lower than the total operations performed during program execution.

The summary of costs is as follows:

- Prover's Running Time: $O(N + M)$, yet it benefits from a significantly improved constant due to the reduced size of opcode circuits.
- Verification Cost and Proof Size: $O(\log^2 N + \log^2 M + m_{\mathsf{BB}} + m_{\Pi})$.

Additionally, this analysis extends to bytecode operations since the responsibility for bytecode verification has shifted from the prover to the verifier, further reducing the overall cost associated with bytecode operations.

8 Conclusions And Discussion

We conclude our paper with the following observations:

Protocol 4 (*Ceno Pro*) *The proving-verification process proceeds as follows, where we use the same notation as in Equation 10:*

- ***Prover:***
  1. *Prover sends auxiliary information to the verifier, including* $\left\{n^{(\mathsf{BB})}\right\}_{\mathsf{BB}\in\Pi}$, $\left\{n^{(\mathsf{chip})}\right\}_{\mathsf{chip}\in\mathsf{Lookup}}$, $n^{(\mathsf{mem,init})}$, $n^{(\mathsf{mem,finl})}$ *and* $n^{(\mathsf{st,finl})}$.
  2. *Prover and verifier exchange witness commitments and challenges in several rounds, including commitments for* $\left\{\mathsf{in}^{(\mathcal{G})}\right\}_{\mathcal{G}\in\mathsf{C}^{(\mathsf{BB})},\mathsf{BB}\in\Pi}$, $\left\{\mathsf{in}^{(\mathsf{chip})}\right\}_{\mathsf{chip}\in\mathsf{Lookup}}.\mathsf{mem}^{(\mathsf{init})}, \mathsf{mem}^{(\mathsf{finl})}, \mathsf{st}^{(\mathsf{finl})}, \mathsf{clk}^{(\mathsf{finl})}$.
  3. *Prover sends* $\left\{\delta_{\star}^{(\mathcal{G})}\right\}_{\mathcal{G}\in\mathsf{C}^{(\mathsf{BB})},\mathsf{BB}\in\Pi,\star\in\{W,R\}}$ *which are generated by the basic block circuits, and* $\left\{\delta_{\star}^{(\mathsf{chip})}\right\}_{\mathsf{chip}\in\mathsf{Lookup},\star\in\{W,R\}}$ *which are generated by lookup table circuits,* $\delta_W^{(\mathsf{mem,init})}$, $\delta_R^{(\mathsf{mem,finl})}$ *and* $\delta_R^{(\mathsf{st,finl})}$ *to the verifier. In addition, the prover generates all GKR proofs for the circuits, following the flow as shown in Figure 10 and sends the proofs to the verifier.*
  4. *Prover opens polynomial commitments and sends the openings to the verifier.*
- ***Verifier:***
  1. *Verifier receives the auxiliary information.*
  2. *Verifier exchanges witness commitments and challenges with the prover in several rounds.*
  3. *The verifier receives GKR proofs for all basic blocks and scans the opcodes in each basic block of the program in reverse order, executing the following steps:*
     (a) *Initially, verify the proof of the BB final circuit and initialize the verifier's stack with its input evaluations.*
     (b) *For each stack opcode encountered, apply the inverse operation to the stack.*
     (c) *For each non-stack opcode with* $N^{(\mathsf{pop})}$ *pops and* $N^{(\mathsf{push})}$ *pushes, pop* $N^{(\mathsf{push})}$ *evaluations from the stack as output evaluations for the corresponding circuit, verify the GKR proof of the current opcode, and then push* $N^{(\mathsf{pop})}$ *input evaluations onto the stack.*
     (d) *At the conclusion, retrieve all unutilized evaluations from the stack as the output evaluations, verify the proof of the BB start circuit, and confirm that the stack is empty.*
  4. *Verifier also receives polynomial commitment openings and verifies them.*
  5. *Verifier checks the correctness of*

$$\begin{aligned}&\delta_R^{(\mathsf{mem,finl})}\cdot\delta_R^{(\mathsf{st,finl})}\cdot\left(\prod_{\mathsf{BB}\in\Pi}\prod_{\mathcal{G}\in\mathsf{C}^{(\mathsf{BB})}}\delta_R^{(\mathcal{G})}\right)\cdot\left(\prod_{\mathsf{chip}\in\mathsf{Lookup}}\delta_R^{(\mathsf{chip})}\right)\\&=\delta_W^{(\mathsf{mem,init})}\cdot\left(\prod_{\mathsf{BB}\in\Pi}\prod_{\mathcal{G}\in\mathsf{C}^{(\mathsf{BB})}}\delta_W^{(\mathcal{G})}\right)\cdot\left(\prod_{\mathsf{chip}\in\mathsf{Lookup}}\delta_W^{(\mathsf{chip})}\right),\end{aligned}$$

Application to Register Machines In this paper, we have demonstrated how to construct a zkVM specifically designed for stack machines. This framework can

be effectively adapted for register machines due to the similarities between stack cells and registers. The main difference between these systems is in their method of addressing: register machines utilize explicit identifiers to specify which register is employed in the current opcode, whereas stack machines manipulate the stack top to determine the cell to be used.

Integrating *Ceno Basic* and *Ceno Pro* protocols We note that the *Ceno Basic* and *Ceno Pro* schemes discussed in this paper are not mutually exclusive. Since each opcode and its associated circuits operate independently—similarly to basic blocks—we can gain advantages by dividing a program into multiple segments. Some parts can be verified using *Ceno Basic*, while others may benefit from the efficiencies of *Ceno Pro*, particularly for basic blocks that are executed repeatedly. Deciding how to optimally segment the program remains an open challenge, as identifying the most effective division strategy requires further analysis and real-world testing. We leave these considerations for future research.

Parallel witness generation In the standard zkVM framework, accelerating witness generation is challenging due to its dependence on the structure of the original program. However, our framework enables parallel witness generation through the following steps:

- Execute the zkVM interpreter to generate the necessary data for the input layer of all independent circuits. This step is efficient because the opcodes operate on unsigned integers, which are natively supported by physical CPUs.
- In the stage of GKR IOP, construct the input layer field elements across multiple rounds using the preprocessed data.
- Before generating the GKR proofs, concurrently compute the intermediate layers for all independent, data-parallel circuits. The parallel computation occurs both across different independent circuits and within each circuit, specifically among various copies of the sub-circuit.

With these steps, although the program execution may be sequential, it incurs a low cost. This allows for the parallel processing of more computationally demanding field operations during witness generation.

Alternative provers We employed GKR as our backend prover due to its compatibility with data-parallel circuits. However, our framework is adaptable to various provers because the GKR arithmetic system generalizes both the Plonkish and CCS arithmetic systems, offering extra flexibility in the configuration of circuit layers. In other words, the Plonkish and CCS arithmetic systems can be considered as specialized cases of GKR circuits with a constant number of layers. Consequently, we can seamlessly incorporate other promising candidates, such as [4, 15], to optimize our independent circuits.

References

1. Spartan2. `https://github.com/microsoft/Spartan2`.
2. Kasra Abbaszadeh, Christodoulos Pappas, Dimitrios Papadopoulos, and Jonathan Katz. Zero-knowledge proofs of training for deep neural networks. *IACR Cryptol. ePrint Arch.*, page 162, 2024.
3. arkworks contributors. `arkworks` zksnark ecosystem, 2022.
4. Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217, 2023. `https://eprint.iacr.org/2023/1217`.
5. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018.
6. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 90–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
7. Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. Cryptology ePrint Archive, Paper 2016/116, 2016. `https://eprint.iacr.org/2016/116`.
8. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 276–294, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
9. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 781–796, USA, 2014. USENIX Association.
10. M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 90–99, 1991.
11. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Paper 2018/962, 2018. `https://eprint.iacr.org/2018/962`.
12. Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *SOSP*, pages 341–357, 2013.
13. Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023. `https://eprint.iacr.org/2023/620`.
14. Benedikt Bünz and Jessica Chen. Proofs for deep thought: Accumulation for large memories and deterministic computations. Cryptology ePrint Archive, Paper 2024/325, 2024. `https://eprint.iacr.org/2024/325`.
15. Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Advances in Cryptology – EUROCRYPT 2023*, pages 499–530, 2023.
16. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Advances in Cryptology – EUROCRYPT 2020*, pages 738–768, 2020.

17. Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. EOS: Efficient private delegation of zkSNARK provers. In *USENIX Security Symposium*, pages 6453–6469, 2023.
18. Shumo Chu, Brandon H. Gomes, Francisco Hernandez Iglesias, Todd Norton, and Duncan Tebbs. Uniplonk: Plonk with universal verifier. Cryptology ePrint Archive, Paper 2023/869, 2023. `https://eprint.iacr.org/2023/869`.
19. Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical Verified Computation with Streaming Interactive Proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112, 2012.
20. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — CRYPTO' 86*, pages 186–194, 1987.
21. Azetc foundation. Aztec. `https://aztec.network/`.
22. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.
23. Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zkSaaS: zero-knowledge SNARKs as a service. In *USENIX Security Symposium*, pages 4427–4444, 2023.
24. Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete stark-friendly cpu architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021. `https://eprint.iacr.org/2021/1063`.
25. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. pages 113–122, 2008.
26. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 291–304. ACM, 1985.
27. Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. `https://eprint.iacr.org/2016/260`.
28. Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. version 2022.3.8. Online, 2022. `https://zips.z.cash/protocol/protocol.pdf`.
29. Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 359–388. Springer, 2022.
30. Modulus Lab. Scaling intelligence: Verifiable decision forest interence with remainder. Github, 2022. `https://github.com/Modulus-Labs/Papers/blob/master/remainder-paper.pdf`.
31. Succinct lab. Succinct processor 1. `https://blog.succinct.xyz/introducing-sp1/`.
32. Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. In *IEEE Symposium on Security and Privacy*, pages 35–35, 2024.
33. Tianyi Liu, Xiang Xie, and Yupeng 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*, page 2968–2985, 2021.

34. C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. pages 2–10 vol.1, 1990.
35. Nexus Inc. Nexus zkVM. `https://github.com/nexus-xyz/nexus-zkvm/`.
36. Ethereum Org. Go ethereum: Official go implementation of the ethereum protocol. `https://github.com/ethereum/go-ethereum`.
37. Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using gkr. Cryptology ePrint Archive, Paper 2023/1284, 2023.
38. Paradigm. Reth: Modular, contributor-friendly and blazing-fast implementation of the ethereum protocol, in rust. `https://github.com/paradigmxyz/reth`.
39. Aztec project. noir language, 2023. `https://noir-lang.org/docs/`.
40. Monero Project. Monero. Online, 2022. `https://github.com/monero-project/monero`.
41. Polygon project. Plonky2. `https://github.com/mir-protocol/plonky2`.
42. Polygon project. Plonky3. `https://github.com/Plonky3/Plonky3`.
43. Polygon project. Polygon Hermez. `https://polygon.technology/solutions/polygon-hermez/`.
44. Polygon project. Polygon Miden. `https://polygon.technology/polygon-miden`.
45. Zcash project. The halo2 book.
46. Zcash project. PLONKish arithmetization. link, 2022.
47. ZkSync project. ZkSync. `https://zksync.io/`.
48. Risc-Zero project. Risc-Zero. `https://www.risczero.com/`.
49. SCIPR Lab. `libiop`: a C++ library for IOP-based zkSNARKs. `https://github.com/scipr-lab/libiop`, 2021.
50. Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Advances in Cryptology – CRYPTO 2020*, pages 704–737, 2020.
51. Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023.
52. Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023.
53. Starkware. Starknet. `https://www.starknet.io/en`.
54. StarkWare Team. `ethSTARK`. `https://github.com/starkware-libs/ethSTARK`, 2021.
55. Scroll tech. Scroll. `https://scroll.io/`.
56. Justin Thaler. Time-Optimal Interactive Proofs for Circuit Evaluation. In *Advances in Cryptology – CRYPTO 2013*, pages 71–89, 2013.
57. Justin Thaler. Proofs, arguments, and zero-knowledge, 2020.
58. Justin Thaler. Proofs, Arguments, and Zero-Knowledge. December 2022.
59. Valida. Valida, a stark-based virtual machine, 2023. `https://github.com/valida-xyz/valida`.
60. R Wahby, S Setty, Z Ren, AJ Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium (NDSS)*, February 2015.
61. Riad S. Wahby, Ye Ji, Andrew J. Blumberg, abhi shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. Cryptology ePrint Archive, Paper 2017/242, 2017. `https://eprint.iacr.org/2017/242`.
62. Wikipedia contributors. Basic block — Wikipedia, the free encyclopedia, 2023.
63. Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology – CRYPTO 2019*, pages 733–764, 2019.

64. Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 3003–3017. ACM, 2022.
65. Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. Veri-zexe: Decentralized private computation with universal setup. Cryptology ePrint Archive, Paper 2022/802, 2022. `https://eprint.iacr.org/2022/802`.
66. Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 2039–2053. ACM, 2020.
67. Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng 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 (CCS)*, pages 159–177, 2021.
68. Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Xiaodong Song. Transparent polynomial delegation and its applications to zero knowledge proof. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876, 2020.
69. Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vram: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 908–925, 2018.
70. Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880, 2017.

A Generalized GKR Arithmetics

We start by introducing a generalization to the GKR protocol. We extend both the gates and the layer structures to a broader notion. Our generalized version of GKR will form the foundation of our zkVM design.

A.1 Gate Design

GKR arithmetics were initially designed for arithmetic circuits, consisting of 2-to-1 multiplication gates and addition gates, as per Equation 1. Then, one can construct circuits for arbitrary polynomial computations, for which various functions can be approximated to any desired degree of accuracy (as per the Stone-Weierstrass theorem in the context of continuous functions on closed intervals).

As shown in Equation 1, the polynomials $\tilde{\mathsf{mul}}(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \mathbf{b}_x^{(1)})$ and $\tilde{\mathsf{add}}(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \mathbf{b}_x^{(1)})$ act as selectors. For a given index point $(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \mathbf{b}_x^{(1)})$, a mul (add, respectively) gate is *switched on* with inputs $(\mathbf{b}_x^{(0)}, \mathbf{b}_x^{(1)})$ and output $\mathbf{b}_y$, if and only if the polynomial $\tilde{\mathsf{mul}}$ ($\tilde{\mathsf{add}}$, respectively) evaluates to 1 over the boolean hypercube. Theoretically, these two gates are already sufficient to build any circuit. Nonetheless, our generalization allows for better expressive gates and concrete performance improvement.

High-degree gates. High-degree gates are a handy tool in zero-knowledge circuit designs. [15] showed how to build such a gate in the Plonkish arithmetics, and [65] reported concrete performance improvements for various circuits when deploying high-degree gates.

GateA: Linear combination of product gates. A degree-d gate is a generalization of the aforementioned `add` and `mul` gates: it is represented by a polynomial

$$\tilde{G}(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) = \begin{cases} 1, & \text{If } \mathsf{out}(\mathbf{b}_y) = \prod_{k=0}^{d-1} \mathsf{in}(\mathbf{b}_x^{(k)}). \\ 0, & \text{Otherwise.} \end{cases} \tag{11}$$

where in and out are this layer's input and output. The above gate is a product of several entries from the input layer. $\tilde{G}(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) = 1$ is interpreted as "the output indexed by $\mathbf{b}_y$ is *equal to* the product $\prod_{k=0}^{d-1} \mathsf{in}(\mathbf{b}_x^{(k)})$", and therefore for each $\mathbf{b}_y$, there should exist exact one $\mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}$ such that $G(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) = 1$.

However, exploiting the nature of GKR protocol, it is convenient to extend this interpretation by allowing arbitrary number of tuples $(\mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)})$ such that given $\mathbf{b}_y$, $G(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) = 1$, which is reinterpreted as "the product $\prod_{k=0}^{d-1} \mathsf{in}(\mathbf{b}_x^{(k)})$ is *added to* the output indexed by $\mathbf{b}_y$". As a result, $\mathsf{out}(\mathbf{b}_y)$ equals the summation of all gates, taking it as the output wire.

This idea was initially proposed in [33]. This gate can be very expressive for the linear combination of products, such as inner product functions.

GateB: Product of linear combination gates. An alternative high-degree gate to the above design is to switch the order of linear combination and product. In this case, we represent the linear combination with d matrices, i.e., $\{G^{(j)}(\mathbf{b}_y, \mathbf{b}_x^{(j)})\}_{j\in[\![d]\!]}$. Then we can compute d vectors, where the j-th vector is given by $f^{(j)}(\mathbf{b}_y) = \sum_{\mathbf{b}_x^{(j)}} G^{(j)}(\mathbf{b}_y, \mathbf{b}_x^{(j)}) \cdot \mathsf{in}(\mathbf{b}_x^{(j)})$. Finally, $\mathsf{out}(\mathbf{b}_y)$ equals the product $\prod_{j\in[\![d]\!]} f^{(j)}(\mathbf{b}_y)$.

Note that the above two types of high-degree gates are not mutually exclusive: we can use both gates in the same protocol, optimizing different components of the same circuit when applicable.

A.2 Data-Parallel Instantiations for Our Gates

Goldwasser et al. [25] proved that log-space uniform circuits can be verified in sublinear time. However, this lower bound is not always guaranteed in practice as log-space uniform circuits are non-trivial to construct for arbitrary operations. Instead, we focus on circuits with specific structures for which efficient verifiers can be easily derived. The data-parallel operations are all we need to design an efficient zkVM.

Data-Parallel Instantiation for GateA. Suppose in and out are one layer's input and output wires. Without loss of generality, we assume $|\mathsf{in}| = |\mathsf{out}| = N = 2^n$, the computation consists of $M = 2^m$ copies of $\frac{N}{M}$-size sub-circuits, and there is only one gate $G(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)})$ in the sub-circuit. GateA's data-parallel instantiation is given by the following sumcheck formula:

$$\begin{aligned}\tilde{\mathsf{out}}(\mathbf{Y}\|\mathbf{T}) = \sum_{\substack{\mathbf{b}_s^{(0)},\\ \mathbf{b}_x^{(0)}}} \cdots \sum_{\substack{\mathbf{b}_s^{(d-1)},\\ \mathbf{b}_x^{(d-1)}}} & \tilde{eq}(\mathbf{T}, \mathbf{b}_s^{(0)}, \ldots, \mathbf{b}_s^{(d-1)}) G(\mathbf{Y}, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) \\ & \cdot \tilde{\mathsf{in}}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s^{(0)}) \cdots \tilde{\mathsf{in}}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s^{(d-1)})\end{aligned} \tag{12}$$

where $\mathbf{b}_s^{(0)}, \ldots, \mathbf{b}_s^{(d-1)} \in \{0,1\}^m$ indicating the indices of the sub-circuit copy, and $\mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)} \in \{0,1\}^{n-m}$, indicating the wire indices inside a sub-circuit.

Data-Parallel Instantiation for GateB. With the same data-parallel layer model as above, GateB's sumcheck protocol instantiation is as follows:

$$\begin{aligned}\tilde{\mathsf{out}}(\mathbf{Y}\|\mathbf{T}) = \sum_{\mathbf{b}_s} \sum_{\mathbf{b}_z} & \tilde{eq}(\mathbf{T}, \mathbf{b}_s) \cdot \tilde{eq}(\mathbf{Y}, \mathbf{b}_z) \cdot \\ & \left(\sum_{\mathbf{b}_x^{(0)}} G^{(0)}(\mathbf{b}_z, \mathbf{b}_x^{(0)}) \cdot \tilde{\mathsf{in}}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s)\right) \cdots \\ & \left(\sum_{\mathbf{b}_x^{(d-1)}} G^{(d-1)}(\mathbf{b}_z, \mathbf{b}_x^{(d-1)}) \cdot \tilde{\mathsf{in}}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s)\right) .\end{aligned} \tag{13}$$

A.3 Unlayered Circuit and Generalized GKR Protocol

Zhang et al. [67] proposed a GKR protocol for unlayered circuits. Such an unlayered circuit is constructed for gates with fan-in-2 inputs, where a layer number is assigned for each gate according to its topological order in the circuit. Under this assignment procedure, a gate denoted as $G_{i,j}$ will have its output in the i-th layer and inputs in the $(i+1)$-th layer and the j-th layer ($j > i+1$), respectively. For $j \neq i+1$, let $S_{j\to i}$ be the subset of used wires in the j-th layer that enter the sumcheck relation of the i-th layer, i.e., there exists some gate $G_{i,j}$. The wire values on this subset are collected into a vector $V_{j\to i}(\cdot)$, while the wire values on the whole layer j are denoted by $V_j(\cdot)$.

To prevent the proving complexity from blowing up with the number of possible $G_{i,j}$s, Zhang et al. [67] also propose to re-index the wires in $S_{j\to i}$ according to their order in this subset. During the GKR reduction process, the original indexes of wires in $S_{j\to i}$ inside the j-th layer are recovered during a sumcheck protocol, converting a subset evaluation $V_{j\to i,\mathsf{eval}}$ to an evaluation $V_{j,\mathsf{eval}}$.

Their approaches have several limitations:

1. The topological sorting based method makes it hard to support a self-defined layer structure. They use topological sorting to automatically assign layer numbers for gates. However, to embed particular sumcheck protocols in the GKR circuit, all candidate wires must be allocated in the same layer, which is hard for the automated approach.
   Furthermore, in their design, exactly one wire must come from the previous layer. This is guaranteed by their layer assignment method. However, such an enforcement excludes other optimal layer assignment opportunities.
2. Their complicated gate design doesn't support more than two fan-in gates naively. For those high fan-in gates, the order of input wires matters (for example, subtraction). Their solution to this issue is to process the same gate multiple times with different input orders.

In this paper, we resolve the above issue by proposing a simplified circuit structure based on the following principles:

1. We restrict the inputs to the gates at layer i to be from the previous layer $i+1$;
2. When a gate at layer i needs an input from any of the layer j's with $j > i+1$, we copy this input to layer $i+1$.

Similar to the structure in [67], we have two types of wire vectors: We define the complete wire values in layer i as vector V_i, and $V_{j\to i}(j > i+1)$ as the collected wire values of the subset $S_{j\to i}$.

Thus, our GKR circuit's i-th layer is structured with the following attributes:

- Gates from layer $i+1$ to layer i; they follow either of the two GKR layer protocols in the previous subsection.
- $\mathsf{paste\text{-}from}_{j\to i}(\mathbf{b}_y, \mathbf{b}_x)$; indicating the $\mathbf{b}_x$-th wire value in the vector $V_{j\to i}$ is pasted to $V_i(\mathbf{b}_y)$.

– $\mathsf{copy\text{-}to}_{i\to k}(\mathbf{b}_y, \mathbf{b}_x)$; indicating the $\mathbf{b}_x$-th wire value in V_i is collected in the vector $V_{i\to k}$, with the new index $\mathbf{b}_y$.

For our zkVM design, we need to support input witnesses from different sources (committed by polynomial commitment scheme or other predecessor circuits) and output to different targets (different successor circuits). Hence, we define $\mathsf{input\text{-}paste\text{-}from}$ and $\mathsf{output\text{-}copy\text{-}to}$ to split the input layer and output layer into subsets.

Based on this circuit structure, we introduce the Generalized GKR Protocols, detailed from Protocol 5 through Protocol 10. Each layer i starts by merging evaluations reduced from the computation in the $(i-1)$-th layer and $V_{i\to k}$. We then apply the sumcheck protocol to both prove and verify computations that include the gates between the i-th and $(i+1)$-th layers, as well as copies from other layers into the i-th layer. We organize the GKR layer protocol into two primary phases:

1. Merge evaluations from the subsequent layers. This is done in Protocol 5 and Protocol 6 and is based on the following identity:

$$\begin{aligned}&\sum_{k=0}^{c-1} \alpha^k \widetilde{V}_i^{(k)}(\mathbf{Y}_k \| \mathbf{T}_k) + \sum_{k=c}^{c+c'-1} \alpha^k \widetilde{V}_{i\to \ell_{k-c}}(\mathbf{Y}_k \| \mathbf{T}_k) \\ &= \sum_{\mathbf{b}_t} \sum_{\mathbf{b}_y} \left(\sum_{k=0}^{c-1} \tilde{eq}(\mathbf{T}_k, \mathbf{b}_t) \tilde{eq}(\mathbf{Y}_k, \mathbf{b}_y) + \sum_{k=c}^{c+c'-1} \tilde{eq}(\mathbf{T}_k, \mathbf{b}_t) \widetilde{\mathsf{copy\text{-}to}}_{i\to \ell_{k-c}}(\mathbf{Y}_k, \mathbf{b}_y) \right) \widetilde{V}_i(\mathbf{b}_y).\end{aligned} \tag{14}$$

2. Check the correctness of the wire values. At layer i, one of the two data-parallel instantiations for our gates, i.e. (Equation 12) or (Equation 13), is proved/verified.
The layerwise adaptation of (Equation 12) is given by

$$\begin{aligned}&\widetilde{V}_i(\mathbf{Y} \| \mathbf{T}) = \sum_{\substack{\mathbf{b}_s^{(0)}, \cdots \mathbf{b}_s^{(d-1)}, \\ \mathbf{b}_x^{(0)} \quad\quad \mathbf{b}_x^{(d-1)}}} \tilde{eq}(\mathbf{T}, \mathbf{b}_s^{(0)}, \ldots, \mathbf{b}_s^{(d-1)}) G_A(\mathbf{Y}, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) \cdot \widetilde{V}_{i+1}(\mathbf{b}_x^{(0)} \| \mathbf{b}_s^{(0)}) \\ &\cdots \widetilde{V}_{i+1}(\mathbf{b}_x^{(d-1)} \| \mathbf{b}_s^{(d-1)}) + \textstyle\sum_{j=0}^{c''} \tilde{eq}(\mathbf{T}, \mathbf{b}_s^{(0)}) \widetilde{\mathsf{paste\text{-}from}}_{\ell'_j}(\mathbf{Y}, \mathbf{b}_x^{(0)}) \widetilde{V}_{\ell'_j \to i}(\mathbf{b}_x^{(0)} \| \mathbf{b}_s^{(0)})\end{aligned} \tag{15}$$

To prove this summation, we launch a d-phase GKR layer protocol shown in Protocol 7 and Protocol 8. In each phase, the initialization of each book-keeping table requires $O(N)$ operations. Then, the sumcheck protocol is applied over a degree-2 equation and will terminate in n rounds.
The layer-wise adaptation of (Equation 13) is given by

$$\begin{aligned}\widetilde{V}_i(\mathbf{Y} \| \mathbf{T}) = \textstyle\sum_{\mathbf{b}_s} \sum_{\mathbf{b}_z} \Big(&\tilde{eq}(\mathbf{T}, \mathbf{b}_s) \cdot \tilde{eq}(\mathbf{Y}, \mathbf{b}_z) \cdot \Big(\textstyle\sum_{\mathbf{b}_x^{(0)}} G_B^{(0)}(\mathbf{b}_z, \mathbf{b}_x^{(0)}) \cdot \widetilde{V}_{i+1}(\mathbf{b}_x^{(0)} \| \mathbf{b}_s) \Big) \\ &\cdots \Big(\textstyle\sum_{\mathbf{b}_x^{(d-1)}} G_B^{(d-1)}(\mathbf{b}_z, \mathbf{b}_x^{(d-1)}) \cdot \widetilde{V}_{i+1}(\mathbf{b}_x^{(d-1)} \| \mathbf{b}_s) \Big) \\ &+ \textstyle\sum_{\mathbf{b}_x^{(0)}} \sum_{j=0}^{c''} \widetilde{\mathsf{paste\text{-}from}}_{\ell'_j}(\mathbf{b}_z, \mathbf{b}_x^{(0)}) \widetilde{V}_{\ell'_j \to i}(\mathbf{b}_x^{(0)} \| \mathbf{b}_s) \Big)\end{aligned} \tag{16}$$

To prove this summation, we present a $(d+1)$-phase GKR layer protocol as shown in Protocol 9 and Protocol 10. The initialization of the book-keeping

tables requires $O(N)$ operations, and the sumcheck protocol is applied to a degree-$(d+1)$ equation with $O(n)$ rounds. Applying the results from [15], the total cost will be $O(d \log^2 dN)$.

B GKR Layer Protocols

B.1 GKR Layer Phase 1 Protocols

Protocol 5 (GKR Layer Prover Protocol, Phase 1) *Suppose L_i is the structure of the i-th layer in the circuit. Before proving current layer, there are c evaluations $\left\{(\widetilde{V}_i(\mathbf{y}_k \| \mathbf{t}_k), (\mathbf{y}_k \| \mathbf{t}_k))\right\}_{0 \le k < c}$ generated by the gate computation of the next layer, and c' evaluations $\left\{(\widetilde{V}_{i \to \ell_{k-c}}(\mathbf{y}_k \| \mathbf{t}_k), (\mathbf{y}_k \| \mathbf{t}_k))\right\}_{c \le k < c+c'}$ corresponding to the subsets $S_{i \to \ell_k}$ copied to the ℓ_k-th layer for $0 \le k < c'$.*

– $\mathsf{GKR\text{-}L}^{(\mathsf{Phase1})}.\mathsf{Prove}_n^{(L_i)}\left(\left\{(\widetilde{V}_i(\mathbf{y}_k \| \mathbf{t}_k), (\mathbf{y}_k \| \mathbf{t}_k))\right\}, \left\{(\widetilde{V}_{i \to \ell_{k-c}}(\mathbf{y}_k \| \mathbf{t}_k), (\mathbf{y}_k \| \mathbf{t}_k))\right\}, \widetilde{V}_i(\mathbf{Y}, \mathbf{T})\right)$*: The prover runs the following steps:*

1. *Set σ and $F_1(\mathbf{Y}) = \sum_{k=0}^{c+c'} f^{(k)}(\mathbf{Y}) g^{(k)}(\mathbf{Y})$, where*

$$\begin{aligned}
\sigma &= \sum_{k=0}^{c-1} \alpha^k \widetilde{V}_i^{(k)}(\mathbf{y}_k \| \mathbf{t}_k) + \sum_{k=c}^{c+c'-1} \alpha^k \widetilde{V}_{i \to \ell_{k-c}}(\mathbf{y}_k \| \mathbf{t}_k) \ , \\
f_1^{(k)}(\mathbf{Y}) &= \widetilde{V}_i(\mathbf{Y} \| \mathbf{t}_k), \ \textit{for } 0 \le k < c + c' \ , \\
g_1^{(k)}(\mathbf{Y}) &= \alpha^k \tilde{eq}(\mathbf{y}_j, \mathbf{Y}), \ \textit{for } 0 \le k < c \ , \\
g_1^{(k)}(\mathbf{Y}) &= \alpha^k \widetilde{\mathsf{copy\text{-}to}}_{i \to \ell_{k-c}}(\mathbf{y}_j, \mathbf{Y}), \ \textit{for } c \le k < c + c',
\end{aligned}$$

2. *Run $\mathsf{SC.Prove}_{n-m,2}(\sigma, F_1(\mathbf{Y}))$. Set $\mathbf{y}$ to be the random variables received from the verifier during the sumcheck protocol, and $F_{1,\mathsf{eval}}$ as the last step evaluation.*
3. *Send $f_{1,\mathsf{eval}}^{(0)}, \ldots, f_{1,\mathsf{eval}}^{(c+c'-1)}$ to the verifier.*
4. *Set $\sigma = F_{1,\mathsf{eval}}$, and $F_2(\mathbf{T}) = f_2(\mathbf{T}) g_2(\mathbf{T})$, where*

$$\begin{aligned}
f_2(\mathbf{T}) &= \widetilde{V}_i(\mathbf{y}, \mathbf{T}) \ , \\
g_2^{(k)}(\mathbf{T}) &= \alpha^k \tilde{eq}(\mathbf{y}_j, \mathbf{y}) \tilde{eq}(\mathbf{t}_k, \mathbf{T}) \ \textit{for } 0 \le k < c \ , \\
g_2^{(k)}(\mathbf{Y}) &= \alpha^k \widetilde{\mathsf{copy\text{-}to}}_{i \to \ell_{k-c}}(\mathbf{y}_j, \mathbf{y}) \tilde{eq}(\mathbf{t}_k, \mathbf{T}), \ \textit{for } c \le k < c + c',
\end{aligned}$$

5. *Run $\mathsf{SC.Prove}_{m,2}(\sigma, F_2(\mathbf{T}))$. Set $\mathbf{t}$ to be the random variables received from the verifier during the sumcheck protocol, and $F_{2,\mathsf{eval}}$ as the last step evaluation.*
6. *Send $f_{2,\mathsf{eval}}$ to the verifier.*

Output $(\widetilde{V}_i(\mathbf{y}, \mathbf{t}), (\mathbf{y}, \mathbf{t}))$

Protocol 6 (GKR Layer Verifier Protocol, Phase 1) *This is the verifier protocol corresponding to Protocol 5.*

– $\mathsf{GKR\text{-}L}^{(\mathsf{Phase1})}.\mathsf{Verify}_n^{(L_i)}\left(\left\{(\widetilde{V}_i(\mathbf{y}_k\|\mathbf{t}_k),(\mathbf{y}_k\|\mathbf{t}_k))\right\},\left\{(\widetilde{V}_{i\to\ell_{k-c}}(\mathbf{y}_k\|\mathbf{t}_k),(\mathbf{y}_k\|\mathbf{t}_k))\right\}\right)$: *The verifier runs the following steps:*

1. *Set* σ *as*
$$\sigma=\sum_{k=0}^{c-1}\alpha^k\widetilde{V}_i^{(k)}(\mathbf{y}_k\|\mathbf{t}_k)+\sum_{k=c}^{c+c'-1}\alpha^k\widetilde{V}_{i\to\ell_{k-c}}(\mathbf{y}_k\|\mathbf{t}_k).$$
2. *Run* $\mathsf{SC.Verify}_{n-m,2}(\sigma)$. *Set* $\mathbf{y}$ *to be the random variables send to the prover during the sumcheck protocol, and* $F_{1,\mathsf{eval}}$ *as the last step evaluation.*
3. *Receive* $f_{1,\mathsf{eval}}^{(0)},\ldots,f_{1,\mathsf{eval}}^{(c+c'-1)}$ *from the prover.*
4. *Compute* $g_{1,\mathsf{eval}}^{(k)}(\mathbf{y})$ *as follows:*
$$\begin{aligned}g_{1,\mathsf{eval}}^{(k)}&=\alpha^k\tilde{eq}(\mathbf{y}_j,\mathbf{y}),\ \text{for } 0\le k<c\ ,\\ g_{1,\mathsf{eval}}^{(k)}&=\alpha^k\widetilde{\mathsf{copy\text{-}to}}_{i\to\ell_{k-c}}(\mathbf{y}_j,\mathbf{y}),\ \text{for } c\le k<c+c',\end{aligned}$$
5. *Check whether* $F_{1,\mathsf{eval}}=\sum_{k=0}^{c+c'}f_{1,\mathsf{eval}}^{(k)}g_{1,\mathsf{eval}}^{(k)}$.
6. *Set* $\sigma=F_{1,\mathsf{eval}}$.
7. *Run* $\mathsf{SC.Verify}_{m,2}(\sigma)$. *Set* $\mathbf{t}$ *to be the random variables send to the prover during the sumcheck protocol, and* $F_{2,\mathsf{eval}}$ *as the last step evaluation.*
8. *Receive* $f_{2,\mathsf{eval}}$ *from the verifier.*
9. *Compute*
$$\begin{aligned}g_{2,\mathsf{eval}}^{(k)}&=\alpha^k\tilde{eq}(\mathbf{y}_j,\mathbf{y})\tilde{eq}(\mathbf{t}_k,\mathbf{t})\ \text{for } 0\le k<c\ ,\\ g_{2,\mathsf{eval}}^{(k)}&=\alpha^k\widetilde{\mathsf{copy\text{-}to}}_{i\to\ell_{k-c}}(\mathbf{y}_j,\mathbf{y})\tilde{eq}(\mathbf{t}_k,\mathbf{t}),\ \text{for } c\le k<c+c',\end{aligned}$$
10. *Check whether* $F_{2,\mathsf{eval}}=\sum_{k=0}^{c+c'}g_{2,\mathsf{eval}}^{(k)}\cdot f_{2,\mathsf{eval}}$.

B.2 GKR Layer Phase 2 Protocols for GateA

Protocol 7 (GKR Layer Prover Protocol, Phase 2, Based on Equation 15) *Suppose* L_i *is the structure of the* i*-th layer in the circuit. Previously, Phase 1 has generated* $(\widetilde{V}_i(\mathbf{y}\|\mathbf{t}),(\mathbf{y}\|\mathbf{t}))$ *as the evaluation and the random point of the layer output. In the end of this phase, the protocol will generated evaluations of the previous layer and the subsets copied from layers in front. Without loss of generality, we assume there is only one type of gate* G *with degree* d *in the layer.*

– $\mathsf{GKR\text{-}L}^{(\mathsf{Phase2})}.\mathsf{Prove}_{n'}^{(L_i)}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}),(\mathbf{y}\|\mathbf{t})),\widetilde{V}_{i+1}(\mathbf{X},\mathbf{S}),\left\{\widetilde{V}_{\ell'_j\to i}(\mathbf{X},\mathbf{S})\right\}\right)$: *the prover will go through the following steps:*

1. *Set* $\sigma=\widetilde{V}_i(\mathbf{y}\|\mathbf{t})$ *and* $F_0(\mathbf{X}\|\mathbf{S})=\sum_{j=0}^{c''}f_{0,j}(\mathbf{X},\mathbf{S})g_{0,j}(\mathbf{X},\mathbf{S})$, *where*
$$\begin{aligned}f_{0,j}(\mathbf{X}\|\mathbf{S})&=\widetilde{V}_{\ell'_j\to i}(\mathbf{X}\|\mathbf{S})\ ,\ \text{for } 0\le j<c''\\ g_{0,j}(\mathbf{X}\|\mathbf{S})&=\tilde{eq}(\mathbf{t},\mathbf{S})\widetilde{\mathsf{paste\text{-}from}}_{\ell'_j\to i}(\mathbf{y},\mathbf{X})\ ,\ \text{for } 0\le j<c''\\ f_{0,c''}(\mathbf{X}\|\mathbf{S})&=\widetilde{V}_{i+1}(\mathbf{X}\|\mathbf{S})\\ g_{0,c''}(\mathbf{X}\|\mathbf{S})&=\sum_{\substack{\mathbf{b}_s^{(1)}\cdots\mathbf{b}_s^{(d-1)}\\ \mathbf{b}_x^{(1)}\cdots\mathbf{b}_x^{(d-1)}}}\tilde{eq}(\mathbf{t},\mathbf{S})\tilde{G}(\mathbf{y},\mathbf{X},\mathbf{b}_x^{(1)},\ldots)\\ &\qquad\widetilde{V}_{i+1}(\mathbf{b}_x^{(1)}\|\mathbf{b}_s^{(1)})\cdots\widetilde{V}_{i+1}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s^{(d-1)})\end{aligned}$$

2. *Run* $\mathsf{SC.Prove}_{n',2}(\sigma, F_0(\mathbf{X}\|\mathbf{S}))$. *Let* $(\mathbf{x}^{(0)}, \|\mathbf{s}^{(0)})$ *be the random challenges received from the verifier.*
3. *Compute* $f_{0,j,\mathsf{eval}} = f_{0,j}(\mathbf{x}^{(0)}, \|\mathbf{s}^{(0)})$ *for* $0 \le j \le c''$ *and* $g_{0,c'',\mathsf{eval}} = g_{0,c''}(\mathbf{x}^{(0)}, \|\mathbf{s}^{(0)})$. *Send those messages to the verifier.*
4. *Set* $\sigma = g_{0,c'',\mathsf{eval}}$.
5. *For* $w = 1..d$, *run the following steps:*
   (a) *Set* $F_w(\mathbf{X}\|\mathbf{S}) = f_w(\mathbf{X}\|\mathbf{S})g_w(\mathbf{X}\|\mathbf{S})$, *where*

$$\begin{aligned} f_w(\mathbf{X}\|\mathbf{S}) &= \widetilde{V}_i(\mathbf{X}\|\mathbf{S}) \\ g_w(\mathbf{X}\|\mathbf{S}) &= \sum_{\substack{\mathbf{b}_s^{(w+1)} \cdots \mathbf{b}_s^{(d-1)} \\ \mathbf{b}_x^{(w+1)} \cdots \mathbf{b}_x^{(d-1)}}} \tilde{eq}(\mathbf{t}, \mathbf{s}_0, \ldots, \mathbf{s}_{w-1}, \mathbf{S}, \mathbf{b}_s^{(w+1)}, \ldots) \\ &\quad \tilde{G}(\mathbf{y}, \mathbf{x}_0, \ldots, \mathbf{x}_{w-1}, \mathbf{X}, \mathbf{b}_x^{(w+1)}, \ldots) \\ &\quad \widetilde{V}_{i+1}(\mathbf{b}_x^{(w+1)}\|\mathbf{b}_s^{(w+1)}) \cdots \widetilde{V}_{i+1}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s^{(d-1)}) \end{aligned}$$

   (b) *Run* $\mathsf{SC.Prove}_{n',2}(\sigma, F_w(\mathbf{X}\|\mathbf{S}))$. *Let* $(\mathbf{x}^{(w)}, \|\mathbf{s}^{(w)})$ *be the random challenges received from the verifier.*
   (c) *Compute* $f_{w,\mathsf{eval}} = f_w(\mathbf{x}^{(w)}, \|\mathbf{s}^{(w)})$ *and if* $w \ne d-1$, *compute* $g_{w,\mathsf{eval}} = g_w(\mathbf{x}^{(w)}, \|\mathbf{s}^{(w)})$. *Send those messages to the verifier.*
   (d) *Set* $\sigma = g_{w,\mathsf{eval}} = g_w(\mathbf{x}^{(w)}, \|\mathbf{s}^{(w)})$.

Output $\left\{(\widetilde{V}_{i+1}(\mathbf{x}^{(j)}, \mathbf{s}^{(j)}), (\mathbf{x}^{(j)}, \mathbf{s}^{(j)}))\right\}_{0 \le j < d}$, $\left\{(\widetilde{V}_{\ell'_j \to i}(\mathbf{x}^{(0)}, \mathbf{s}^{(0)}), (\mathbf{x}^{(0)}, \mathbf{s}^{(0)}))\right\}_{0 \le j < c''}$.

Protocol 8 (GKR Layer Verifier Protocol, Phase 2, Based on Equation 15) *This is the verifier protocol corresponding to Protocol 7.*

– $\mathsf{GKR\text{-}L}^{(\mathsf{Phase2})}.\mathsf{Verify}_{n'}^{(L_i)}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t}))\right)$: *the verifier will go through the following steps:*

1. *Set* $\sigma = \widetilde{V}_i(\mathbf{y}\|\mathbf{t})$.
2. *Run* $\mathsf{SC.Verify}_{n',2}(\sigma)$. *Let* $(\mathbf{x}^{(0)}, \|\mathbf{s}^{(0)})$ *be the random challenges sent to the prover. Let* $F_{0,\mathsf{eval}}$ *be the last evaluation received form the prover.*
3. *Received* $f_{0,j,\mathsf{eval}}$ *for* $0 \le j \le c''$ *and* $g_{0,c'',\mathsf{eval}}$ *from the prover.*
4. *Set*

$$g_{0,j}(\mathbf{x}^{(0)}\|\mathbf{s}^{(0)}) = \tilde{eq}(\mathbf{t}, \mathbf{s}^{(0)})\widetilde{\mathsf{paste\text{-}from}}_{\ell'_j \to i}(\mathbf{y}, \mathbf{x}^{(0)}) \text{ , for } 0 \le j < c''$$

5. *Check* $F_{0,\mathsf{eval}} = \sum_{j=0}^{c''} f_{0,j,\mathsf{eval}} \cdot g_{0,j,\mathsf{eval}}$.
6. *Set* $\sigma = g_{0,c'',\mathsf{eval}}$.
7. *For* $w = 1..d$, *run the following steps:*
   (a) *Run* $\mathsf{SC.Verify}_{n',2}(\sigma)$. *Let* $(\mathbf{x}^{(w)}, \|\mathbf{s}^{(w)})$ *be the random challenges sent to the prover. Let* $F_{w,\mathsf{eval}}$ *be the last evaluation received form the prover.*
   (b) *Receive* $f_{w,\mathsf{eval}}$ *and if* $w \ne d-1$, *then also* $g_{w,\mathsf{eval}}$ *from the prover.*
   (c) *Check* $F_{w,\mathsf{eval}} = f_{w,\mathsf{eval}} \cdot g_{w,\mathsf{eval}}$, *where if* $w = d-1$

$$g_{w,\mathsf{eval}} = \tilde{eq}(\mathbf{t}, \mathbf{s}_0, \ldots, \mathbf{s}_{d-1})\tilde{G}(\mathbf{y}, \mathbf{x}_0, \ldots, \mathbf{x}_{d-1})$$

   (d) *Set* $\sigma = g_{w,\mathsf{eval}}$.

B.3 GKR Layer Phase 2 Protocols for GateB

Protocol 9 (GKR Layer Prover Protocol, Phase 2, Based on Equation 16) *Suppose L_i is the structure of the i-th layer in the circuit. Previously, Phase 1 has generated $(\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t}))$ as the evaluation and the random point of the layer output. In the end of this phase, the protocol will generated evaluations of the previous layer and the subsets copied from layers in front.*

– $\mathsf{GKR\text{-}L^{(}Phase2).Prove}_{n'}^{(L_i)}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t})), \widetilde{V}_{i+1}(\mathbf{X}, \mathbf{S}), \left\{\widetilde{V}_{\ell'_j \to i}(\mathbf{X}, \mathbf{S})\right\}\right)$*: the prover will go through the following steps:*

1. *Set $\sigma = \widetilde{V}_i(\mathbf{y}\|\mathbf{t})$.*
2. *Set $F_\star(\mathbf{Z}\|\mathbf{S}) = g_\star(\mathbf{Z}\|\mathbf{S}) \cdot \left(\prod_{j=0}^{d-1} f_\star^{(j)}(\mathbf{Z}\|\mathbf{S}) + \sum_{j=0}^{c''-1} h_\star^{(j)}(\mathbf{Z}\|\mathbf{S})\right)$, where*

$$\begin{aligned} g_\star(\mathbf{Z}\|\mathbf{S}) &= \tilde{eq}(\mathbf{S}, \mathbf{t})\tilde{eq}(\mathbf{Z}, \mathbf{y}) \\ f_\star^{(j)}(\mathbf{Z}\|\mathbf{S}) &= \textstyle\sum_{\mathbf{b}_x^{(j)}} \tilde{G}^{(j)}(\mathbf{Z}, \mathbf{b}_x^{(j)})\widetilde{V}_{i+1}(\mathbf{b}_x^{(j)}\|\mathbf{S}) \\ h_\star^{(j)}(\mathbf{Z}\|\mathbf{S}) &= \textstyle\sum_{\mathbf{b}_x^{(0)}} \widetilde{\mathsf{paste\text{-}from}}_{\ell'_j \to i}(\mathbf{Z}, \mathbf{b}_x^{(0)})\widetilde{V}_{\ell'_j \to i}(\mathbf{b}_x^{(0)}\|\mathbf{S}) \end{aligned}$$

3. *Run $\mathsf{SC.Prove}_{n',d+1}(\sigma, F_\star(\mathbf{Z}\|\mathbf{S}))$. Let $(\mathbf{z}, \|\mathbf{s})$ be the random challenges received from the verifier, and $F_{\star,\mathsf{eval}}$ be the evaluation computed by the last step.*
4. *Compute the evaluation $f_{\star,\mathsf{eval}}^{(j)} = f_\star^{(j)}(\mathbf{z}\|\mathbf{s})$ and $h_{\star,\mathsf{eval}}^{(j)} = h_\star^{(j)}(\mathbf{z}\|\mathbf{s})$. Send those messages to the verifier.*
5. *Set $\sigma = F_{\star,\mathsf{eval}} \cdot (g_{\star,\mathsf{eval}})^{-1}$, where $g_{\star,\mathsf{eval}} = g_\star(\mathbf{z}\|\mathbf{s})$.*
6. *Set $F_0(\mathbf{X}) = f'_0(\mathbf{X})g'_0(\mathbf{X}) + \sum_{j=0}^{c''-1} f_0^{(j)}(\mathbf{X})g_0^{(j)}(\mathbf{X})$, where*

$$\begin{aligned} f'_0(\mathbf{X}) &= \widetilde{V}_{i+1}(\mathbf{X}\|\mathbf{s}) \\ g'_0(\mathbf{X}) &= \tilde{G}^{(0)}(\mathbf{z}\|\mathbf{X}) \cdot \textstyle\prod_{j=1}^{d-1} \left(\sum_{\mathbf{b}_x^{(j)}} \widetilde{V}_{i+1}(\mathbf{b}_x^{(j)}\|\mathbf{s})\tilde{G}^{(j)}(\mathbf{z}, \mathbf{b}_x^{(j)})\right) \\ f_0^{(j)}(\mathbf{X}) &= \widetilde{V}_{\ell'_j \to i}(\mathbf{X}\|\mathbf{s}) \\ g_0^{(j)}(\mathbf{X}) &= \widetilde{\mathsf{paste\text{-}from}}_{\ell'_j \to i}(\mathbf{z}, \mathbf{X}). \end{aligned}$$

7. *Run $\mathsf{SC.Prove}_{n'-m,2}(\sigma, F_0(\mathbf{X}))$. Let $\mathbf{x}^{(0)}$ be the random challenges received from the verifier.*
8. *Compute the evaluations $f'_{0,\mathsf{eval}} = f'_0(\mathbf{x}^0)$, $g'_{0,\mathsf{eval}} = g'_0(\mathbf{x}^0)$, and $f_{0,\mathsf{eval}}^{(j)} = f_0^{(j)}(\mathbf{x}^0)$ for $0 \le j < c''$. Send those messages to the verifier.*
9. *Set $\sigma = g'_{0,\mathsf{eval}} \cdot (G^{(0)}(\mathbf{z}\|\mathbf{x}^{(0)}))^{-1}$.*
10. *For $w = 1..d$, run the following steps:*
    (a) *Set $F_w(\mathbf{X}) = f_w(\mathbf{X})g_w(\mathbf{X})$ where*

$$\begin{aligned} f_w(\mathbf{X}) &= \widetilde{V}_{i+1}(\mathbf{X}\|\mathbf{s}) \\ g_w(\mathbf{X}) &= \tilde{G}^{(w)}(\mathbf{z}\|\mathbf{X}) \cdot \textstyle\prod_{j=w+1}^{d-1} \left(\sum_{\mathbf{b}_x^{(j)}} \widetilde{V}_{i+1}(\mathbf{b}_x^{(j)}\|\mathbf{s})\tilde{G}^{(j)}(\mathbf{z}, \mathbf{b}_x^{(j)})\right). \end{aligned}$$

    (b) *Run $\mathsf{SC.Prove}_{n'-m,2}(\sigma, F_w(\mathbf{X}))$. Let $\mathbf{x}^{(w)}$ be the random challenges received from the verifier.*

(c) *Compute the evaluations* $f_{w,\mathsf{eval}} = f_w(\mathbf{x}^w)$*, and if* $w \neq d-1$*, compute* $g_{w,\mathsf{eval}} = g_w(\mathbf{x}^w)$*. Send those messages to the verifier.*
(d) *Set* $\sigma = g_{w,\mathsf{eval}} \cdot (G^{(w)}(\mathbf{z}\|\mathbf{x}^{(w)}))^{-1}$.

Output $\left\{(\widetilde{V}_{i+1}(\mathbf{x}^{(j)}, \mathbf{s}), (\mathbf{x}^{(j)}, \mathbf{s}))\right\}_{0 \leq j < d}$, $\left\{(\widetilde{V}_{\ell'_j \to i}(\mathbf{x}^{(0)}, \mathbf{s}), (\mathbf{x}^{(0)}, \mathbf{s}))\right\}_{0 \leq j < c''}$.

Protocol 10 (GKR Layer Verifier Protocol, Phase 2, Based on Equation 16)
This is the verifier protocol for Protocol 9.

– $\mathsf{GKR\text{-}L}^{(}\mathsf{Phase2}).\mathsf{Verify}_{n'}^{(L_i)}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t}))\right)$*: the verifier will go through the following steps:*

1. *Set* $\sigma = \widetilde{V}_i(\mathbf{y}\|\mathbf{t})$.
2. *Run* $\mathsf{SC.Verify}_{n',d+1}(\sigma)$*. Let* $(\mathbf{z}, \|\mathbf{s})$ *be the random challenges sent to the prover, and* $F_{\star,\mathsf{eval}}$ *be the evaluation received by the last step.*
3. *Receive the evaluations* $f^{(j)}_{\star,\mathsf{eval}}$ *and* $h^{(j)}_{\star,\mathsf{eval}}$ *from the prover.*
4. *Check* $F_{\star,\mathsf{eval}} = g_{\star,\mathsf{eval}} \cdot \left(\prod_{j=0}^{d-1} f_{\star,\mathsf{eval}} + \sum_{j=0}^{c''-1} h^{(j)}_{\star,\mathsf{eval}}\right)$*, where*

$$g_{\star,\mathsf{eval}} = \tilde{eq}(\mathbf{s}, \mathbf{t})\tilde{eq}(\mathbf{z}, \mathbf{y})$$

5. *Set* $\sigma = F_{\star,\mathsf{eval}} \cdot (g_{\star,\mathsf{eval}})^{-1}$.
6. *Run* $\mathsf{SC.Verify}_{n'-m,2}(\sigma)$*. Let* $\mathbf{x}^{(0)}$ *be the random challenges sent to the prover. Let* $F_{0,\mathsf{eval}}$ *be the last step evaluation from the prover.*
7. *Receive the evaluations* $f'_{0,\mathsf{eval}}$*,* $g'_{0,\mathsf{eval}}$*, and* $f^{(j)}_{0,\mathsf{eval}}$ *for* $0 \leq j < c''$.
8. *Check* $F_{0,\mathsf{eval}} = f'_{0,\mathsf{eval}} g'_{0,\mathsf{eval}} + \sum_{j=0}^{c''-1} f^{(j)}_{0,\mathsf{eval}} g^{(j)}_{0,\mathsf{eval}}$*, where*

$$g^{(j)}_{0,\mathsf{eval}} = \widetilde{\mathsf{paste\text{-}from}}_{\ell'_j \to i}(\mathbf{z}, \mathbf{x}^{(0)}).$$

9. *Set* $\sigma = g'_{0,\mathsf{eval}} \cdot (G^{(0)}(\mathbf{z}\|\mathbf{x}^{(0)}))^{-1}$.
10. *For* $w = 1..d$*, run the following steps:*
    (a) *Run* $\mathsf{SC.Verify}_{n'-m,2}(\sigma)$*. Let* $\mathbf{x}^{(w)}$ *be the random challenges sent to the prover. Let* $F_{w,\mathsf{eval}}$ *be the last step evaluation from the prover.*
    (b) *Receive the evaluations* $f_{w,\mathsf{eval}}$*, and if* $w \neq d-1$*, receive* $g_{w,\mathsf{eval}}$ *from the prover.*
    (c) *Check* $F_{w,\mathsf{eval}} = f_{w,\mathsf{eval}} \cdot g_{w,\mathsf{eval}}$ *where if* $w = d-1$

$$g_{w,\mathsf{eval}} = \tilde{G}^{(d-1)}(\mathbf{z}\|\mathbf{x}^{(d-1)}).$$

    (d) *Set* $\sigma = g_{w,\mathsf{eval}} \cdot (G^{(w)}(\mathbf{z}\|\mathbf{x}^{(w)}))^{-1}$.