Parallel Zero-knowledge Virtual Machine

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

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

Abstract. Zero-knowledge virtual machine (zkVM) is a novel application of succinct and non-interactive zero-knowledge proof protocols that allows for verifiable computation over arbitrary codes. In this paper, we present a new paradigm to build such a zkVM via data parallel circuits. Our parallelization happens at the opcode and the basic block level. Such a design allows us to eliminate almost all of the circuit overhead for opcodes, including the control flow and the data flow which can be substantial in a zero-knowledge circuit. The size of our circuit is dynamic and optimal in the sense that it only costs the *sum of all individual opcodes* that are executed in the program, (i.e., you only pay as much as you use); while in all previous designs, the circuit is of a constant size, determined by the product of a) the upper limit of the number of opcodes, and b) the size of a super opcode circuit that is capable of handling every type of opcode.
Further, we present an asymmetric GKR prover that is tailored to the data parallelism in our zkVM design, accompanied by various optimized constraint gadgets. The use of GKR prover also leads to significant reductions in the number of witnesses that are to be committed: GKR allows us to commit only the input and output of the circuits, whereas in previous Plonkish-based solutions, the prover needs to commit to all the witnesses.

1 Introduction

Zero-knowledge proof (ZKP) protocols [17] 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 size of computation and enables faster validation. The past decade has witnessed fruitful results of zk-SNARKs, in terms of both theoretical breakthroughs [6, 7, 13, 16, 20, 41], and practical and concrete

** Alphabetical Order.

instantiations [2,32,36,40,45], enabling various applications, such as anonymous payment protocols [12,19,31], distributed private computations [5,9,14,23,54], zero-knowledge machine learning (zkML) [21,24,55] 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 assert that a piece of program, in the form of a list of opcodes over a target virtual machine, takes in a given input and outputs a given result, the prover generates a proof of the correct execution of the opcodes over the committed inputs and the committed 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; whereas 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 [26,39] and WASM. On the other hand, active research is conducted to invent zero-knowledge friendly programming languages and their intermediate representations [15,30]. The main challenge of this route remains efficiency, with multiple research directions in the form of better and dedicated VM designs such as [15,34,50] and high performance proof systems [33,36].

zkEVM. Another typical use case of zkVM is its application to the Ethereum virtual machine, also known as zero-knowledge Ethereum Virtual machines (zkEVMs) [38,46]. The EVM is the execution environment that runs on the Ethereum blockchain. In the Ethereum blockchain, each program is a smart contract that is committed publicly, and executed automatically upon receiving transactions. It has been widely applied to the fields of finance, supply chain, voting system, legal industry, and so on. The EVM is a computational engine that functions as a decentralized computer, hosting and executing smart contracts on the Ethereum blockchain. The EVM 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 the limitation of block data and network throughput.

zkEVM is one of the two major candidates for Ethereum scalability, and the only one that is backed by cryptography; with the other one 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 build directly from EVM's opcode, or indirectly from another zkVM. As illustrated in Figure Figure 1, one can compile the Go or Rust implementation of EVMs [27,29] into RISC-V opcodes, and then use zkVM to prove the RISC-V opcodes. At a glance this method may look more complex than directly building a zkEVM, as it involves more components in its workflow. However, in practice, most of 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, originated from the fact that the EVM itself is a fast moving target and is under active development. Recent progress in [22] shows that this route actually delivers compelling performance compared with the first method.

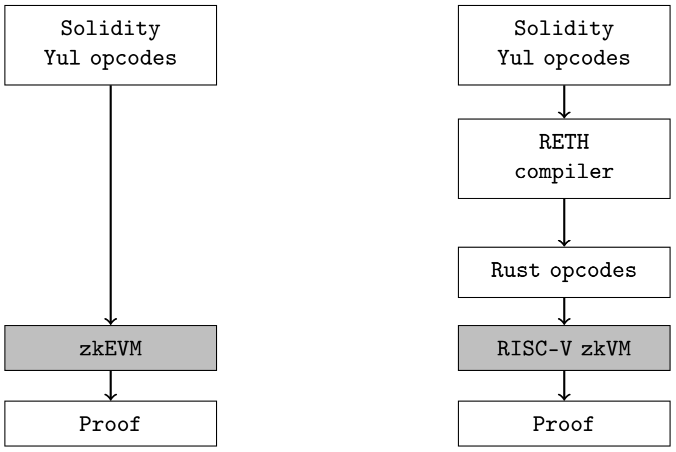


Fig. 1: Two typical ways to build zero-knowledge Ethereum virtual machines

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

1.1 Related work

Zero knowledge proof systems Zero-knowledge proofs were invented in the seminar work of [17]. Modern zk-SNARKs are constructed by compiling an information-theoretic object called an Interactive Oracle Proof (IOP) [3] 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 [16]. Converting GKR into a non-interactive protocol is a direct application of the classic Fiat-Shamir transformation [11]. For the rest of the paper, for the ease of presentation, we will focus on the interactive version. 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 via repetitively invoking a sumcheck protocol [25], 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 linear time for the prover and verifier. Thaler [47] showed that GKR is friendly to data parallel circuits, i.e., circuits consisting of a repetitive pattern. The GKR circuits are sometimes referred to as the *data parallel circuits* due to this finding; it is the key differentiator between GKR circuits and punkish or R1CS circuits. Zhang et al. [56] improves the concrete performance by allowing non-consecutive layer constraints.

There have also been many optimizations of GKR for dedicated applications. The works [21, 24] and [1] show that GKR has great potential for machine learning circuits, specifically, convolutional neural network, decision trees, and training. The works [28, 43] build concretely efficient lookup tables from GKR protocols. ZK-Bridge [53] reports great performance numbers for repetitive ECDSA circuits, attributing to GKR's friendliness to data parallel circuits.

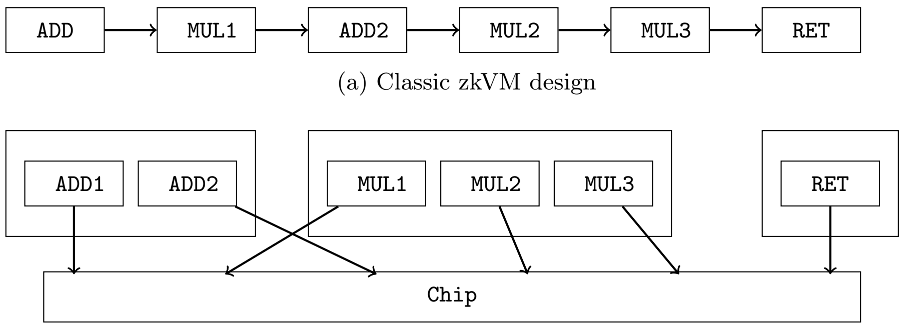
zkVMs. In the literature, there exists a list of zkVM solutions, such as Scroll [46] and Polygon [34, 35]. They all follow a same paradigm:

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 [37], rank one constraint system (R1CS) or customizable constraint system [42];
2. Apply a SNARK protocol (not necessarily zero-knowledge) to the constraint system to generate a proof. Candidates are Plonky [32], Starky [44] or Halo2 [36] for Plonkish arithmetizations, and Groth16 [18], Marlin [8] or Spartan [41] 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 [5].

Note that when the program is huge, the proving time becomes problematic. A workaround is *execution continuation*: a long list of opcodes is divided into multiple small lists, each small list is proved separately and a macro proof is also present to glue the small lists together. Folding schemes such as [6, 20] are considered a perfect candidate for such use cases when instances from multiple small lists can be folded into a single one.

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 are asking, 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

Fig. 2: Classical zkVM vs Our design

In this paper, we present a novel approach to build zero-knowledge virtual machines. Based on this framework, we present two schemes.

GKR-zkVM. The first scheme is named GKR-zkVM as it is very parallel, like the circuits for our zkVMs. In contrast to existing solutions where program opcodes are proved sequentially as they are executed (see Figure 2a), we group the identical opcodes together as in Figure 2b and prove them in batches. In more details, we execute the program in full and obtain all the opcodes, and their corresponding inputs and outputs. Then, we group opcodes and generate proofs for correct executions for each group. At a high level, the task of proving a program is then divided into two sub-tasks:

1. Prove the correct execution of each opcode. We refer to this circuit as the opcode circuit.
2. Prove that the opcode circuits are executed in the correct order, and the global states are updated resepectively. We refer to the circuit for fulfilling this task as the chip circuit.

For the first task, we use a repetition friendly prover as each group now consists of identical opcodes. We use the GKR backend and utilize its data-parallel circuit in our paper.

Having proved the correctness of each opcode, we use a chip circuit that glues all the opcodes together, proving that the opcodes, though proved out of their order, are derived truthfully from the original program. The main tools we need for this statement are set equality checks and lookup arguments, both instantiated using the GKR protocol.

The above framework already improves the existing, sequential solutions in many respects. To name a few, our protocol avoids the heavy overhead from the universal opcode circuit, i.e., one opcode circuit that supports all possible opcodes as a requirement for the static zero-knowledge proof circuit, whereas our prover is adaptive to dynamic execution trace. We also eliminate the branching overhead in selecting opcodes during a sequential proving. In the mean time, our dynamic circuit proves as many opcodes as in the original program, while classical solutions require their static circuits to handle max possible number of opcodes that any supported program may be compiled into.

GKR-zkVM Pro. On top of GKR-zkVM, we further investigate the scenario where the program structure can be exploitable. We present GKR-zkVM Pro, which is empowered by a *structure-aware prover* that allows us to take advantage of decades of advancement from the compiler community. Modern compilers compile programs into chunks of likely repetitive opcodes for optimizations. Each chunk is called a basic block. Our improved version of the scheme works over the basic blocks. We design our circuit so that data propagation through opcodes within the same basic block is implicitly guaranteed. This allows us to further reduce the stack and the memory circuits.

Asymmetric GKR prover. To accommodate the fact that the circuits in GKR-zkVM and GKR-zkVM Pro are dynamic, we present an asymmetric GKR protocol where the prover dynamically adjusts to the circuit while the verifier remains static.

During verification, the verifier will receive a piece of auxiliary information, which is a description of the program that has been executed. However, this auxiliary input to the verifier is a leak of information, and makes the verifier non-static. To mitigate this issue, following the common practice, we use a GKR verification circuit to generate a recursive proof. This recursive proof no longer leaks information about the original program. Meanwhile, the circuit verifying this recursive proof is static, and therefore, a static verifier.

Compared with other leading candidates in this domain, such as Plonk [13,32] 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 Plonk-ish arithmetizations, one is required to commit to all those intermediate data;

whereas in GKR these costs are avoided. Table 1 shows the number of witness cells that our protocol commits per opcode.

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.

opcode	GKR-zkVM	GKR-zkVM Pro
JUMP	16	0
POP	32	-
SWAP2	64	-
DUP1	32	-
ADD	128	32
GT	128	32
JUMPI	64	16
PUSH1	16	-
MSTORE	64	64
RETURN	32	32

Table 1: Witness sizes for some opcodes in GKR-zkVM and GKR-zkVM Pro, where sizes are padded to the power of 2s for simplicity. We use - to indicate that the opcode is not needed in GKR-zkVM Pro.

1.3 Organization of This Paper

In Section 2, we introduce several definitions related to our paper. In Section 3, we present the GKR arithemtics, introducing the high degree custom gates. Then, we present our variant of GKR protocol in Section 4. In Section 5 and Section 6 we present the core design of our protocol. Finally we show practical tricks to handle ROM and RAM in Section 7.

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, [48, 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 that 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 concatenation of bit strings, random points and variables.

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

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

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

$$\tilde{eq}(\mathbf{X},\mathbf{Y}^{(0)},\dots,\mathbf{Y}^{(d-1)}) = \prod_{i=0}^{n-1}\left((1-X_i)(1-Y_0^{(i)})\cdots(1-Y_{d-1}^{(i)})+X_iY_0^{(i)}\cdots Y_{d-1}^{(i)}\right)\ .$$

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* $\mathcal{A}$

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

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

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

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

If the interactive argument of knowledge protocol ARG is public-coin, is has been shown that by the Fiat-Shamir transformation [11], 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).*

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 $i = 0, \dots, n-1$, run the following steps:*
     (a) *Set*
     $$f^{(i)}(X) = \sum_{\mathbf{b} \in \{0,1\}^{n-i-1}} F(\mathbf{b}, X, x_{n-i}, \dots, x_{n-1}).$$
     (b) *Compute $f^{(i)}(1), \dots, f^{(i)}(d)$ 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 $i = 0, \dots, n-1$, run the following steps:*
     (a) *Receive $f^{(i)}(1), \dots, f^{(i)}(d)$ from the verifier and compute $f^{(i)}(0) = \sigma_i - \sum_{j=1}^{d} f^{(i)}(j)$.*
     (b) *Randomly generate $x_{n-i-1} \leftarrow \mathbb{F}$ and send to the prover.*
     (c) *Recover $f^{(i)}(X)$ from $\left(f^{(i)}(0), \dots, f^{(i)}(d)\right)$ and compute $\sigma_{i+1} = f^{(i)}(x_{n-i-1})$.*
  3. *Query the oracle $F_{\mathsf{eval}} = F(x_0, \dots, x_{n-1})$. 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.

Fig. 3: Sumcheck Protocol

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., $c = \sum_{b_0,\dots,b_{n-1} \in \{0,1\}} f(b_0, \dots, b_{n-1})$. Calculating the sum directly requires exponential time in n, as there are 2^n combinations of $b_0, \dots, b_{n-1}$. Lund et al. [25] 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. We present, in Protocol 1, the non-interactive version of the sumcheck protocol after applying the Fiat-Shamir transform.

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

Definition 2 (Multilinear Extension [10]). *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:*

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

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. The state-of-the-art algorithm is proposed by Xie et al. [52], 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.*

GKR Protocol *GKR* [16] 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 [10, 47, 52, 57, 58], 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. [52]. 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. [56] 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. There is an interactive proof for the function computed by C with soundness $O(d\log|C|/|\mathbb{F}|)$. The running time of the prover $\mathcal{P}$ is $O(|C|)$. The proof size is $\min\left\{O(d\log C + d^2), O(|C|)\right\}$. 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 Constraints toolbox

Our protocol uses the following constraints as subroutines.

Set equality checking 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

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

Lookup argument For two multisets $A = \left\{a_0, \ldots, a_{|A|-1}\right\}$ and $T = \left\{t_0, \ldots, t_{|T|-1}\right\}$, with an extra challenge τ, the constraints to check $A \subseteq T$ is

$$\sum_{i=0}^{|A|-1} \frac{1}{a_i + \tau} = \sum_{i=0}^{|T|-1} \frac{m_i}{t_i + \tau}, \tag{3}$$

where m_i denotes the number of occurrences of t_i in A.

LogUp is an efficient lookup argument protocol that uses GRK as the backend [28]. The GKR circuit of LogUp is designed for checking Equation 3 without involving any inverting operation. The field inverting operation is expensive in circuit, because it requires the prover to supply the inverted result as an input to the GKR circuit. LogUp avoids those expensive inversions by computing the sum of fractions instead. It exploits the fraction addition formula $a/b + c/d = (ad + bc)/(bd)$ to compute the sum of fractions, and then checks the equality $ad = bc$ instead of $a/b = c/d$. As a result, LogUp completely removed the division and inversion.

3 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.

3.1 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 polynomials computations, for which a wide variety of 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 $\mathbb{F}_2$. These two gates are already sufficient to build any circuit, from a theoretical perspective. Nonetheless, our generalization allows for better expressive gates and concrete performance improvement.

High-degree gates. High degree gates are a very useful tool in zero-knowledge circuit designs. [7] showed how to build such a gate in the Plonkish arithmetization, and [54] reported concrete performance improvements for various circuits when high degree gates are deployed.

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

$$\tilde{G}(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) = \begin{cases} 1, & \text{If } \mathsf{out}(\mathbf{b}_y) = \prod_{k=0}^{d-1} \mathsf{in}(\mathbf{b}_x^{(k)}). \\ 0, & \text{Otherwise.} \end{cases} \tag{4}$$

where in and out are respectively the input and the output of this layer.

The above gate is a product of several entries from the input layer. It is simple as it mimics the original $\tilde{\mathsf{add}}$ and $\tilde{\mathsf{mul}}$ design; yet it is elegant as we can derive an expressive gates from $\tilde{G}$. In the simple case, similarly to the traditional GKR,

$\tilde{G}(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) = 1$ is interpreted as "the output indexed by $\mathbf{b}_y$ is *equal to* the definition, in this case, 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 satisfying $\mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}$, and reinterpret $G(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)}) = 1$ 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 [24]. For parallel operations such as inner product functions, this gate can be very expressive.

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 $f(\mathbf{b}_y)^{(j)} = \sum_{\mathbf{b}_x^{(j)}} G^{(j)}(\mathbf{b}_y, \mathbf{b}_x^{(j)}) \cdot \mathsf{in}(\mathbf{b}_x^{(j)})$. Finally, $\mathsf{out}(\mathbf{b}_y)$ equals the product $\prod_{j \in [\![d]\!]} f(\mathbf{b}_y)^{(j)}$.

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

3.2 Data-Parallel Instantiations for Our Gates

Goldwasser et al. [16] proved that log-space uniform circuits can be verified in sublinear time. However, in practice, this lower bound is not always guaranteed as log-space uniform circuits are non-trivial to construct for arbitrary operations. Instead, we focus on circuits with specific structures for which it is easy to derive efficient verifiers. Particularly, we consider data-parallel computation [47] that is to be supported in our GKR protocol. Looking ahead, we also claim that data-parallel operations are all we need to design an efficient zkVM.

Data-Parallel Instantiation for GateA. Suppose in and out are the input and output wires of one layer. 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. We use $(\mathbf{b}_x \| \mathbf{b}_s)$ to index the wires in a layer, in which $\mathbf{b}_x$ indicates the wire index inside a sub-circuit and $\mathbf{b}_s$ indicates the index of the sub-circuit copy. A gate in the sub-circuit is denoted as $G(\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)})$. For $\mathbf{b}_y, \mathbf{b}_x^{(0)}, \ldots, \mathbf{b}_x^{(d-1)} \in \{0,1\}^{n-m}$ indexing the wires in a sub-circuit, and $\mathbf{b}_t \in \{0,1\}^m$ indexing different sub-circuit copies, GateA's data-parallel instantiation is given by the following sumcheck formula:

$$\tilde{\mathsf{out}}(\mathbf{Y}\|\mathbf{T}) = \sum_{\substack{\mathbf{b}_s^{(0)},\\ \mathbf{b}_x^{(0)}}} \cdots \sum_{\substack{\mathbf{b}_s^{(d-1)},\\ \mathbf{b}_x^{(d-1)}}} \tilde{eq}(\mathbf{T}, \mathbf{b}_s^{(0)}, \dots, \mathbf{b}_s^{(d-1)}) G(\mathbf{Y}, \mathbf{b}_x^{(0)}, \dots, \mathbf{b}_x^{(d-1)}) \cdot \tilde{\mathsf{in}}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s^{(0)}) \cdots \tilde{\mathsf{in}}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s^{(d-1)}) \tag{5}$$

where $\mathbf{b}_s^{(0)}, \dots, \mathbf{b}_s^{(d-1)} \in \{0,1\}^m$ and $\mathbf{b}_x^{(0)}, \dots, \mathbf{b}_x^{(d-1)} \in \{0,1\}^{n-m}$.

Data-Parallel Instantiation for GateB. With the same data-parallel layer model as above, GateB's data-parallel instantiation is given by the following sum-check formula:

$$\tilde{\mathsf{out}}(\mathbf{Y}\|\mathbf{T}) = \sum_{\mathbf{b}_s} \sum_{\mathbf{b}_z} \tilde{eq}(\mathbf{T}, \mathbf{b}_s) \cdot \tilde{eq}(\mathbf{Y}, \mathbf{b}_z) \cdot \left(\sum_{\mathbf{b}_x^{(0)}} G^{(0)}(\mathbf{b}_z, \mathbf{b}_x^{(0)}) \cdot \tilde{\mathsf{in}}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s) \right) \cdots \left(\sum_{\mathbf{b}_x^{(d-1)}} G^{(d-1)}(\mathbf{b}_z, \mathbf{b}_x^{(d-1)}) \cdot \tilde{\mathsf{in}}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s) \right) . \tag{6}$$

3.3 Unlayered Circuit and Our Genaralized GKR Protocol

Zhang et al. [56] 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. [56] 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 requirement of input wires makes it hard to support self-defined layer structure. They use topological sorting to automatically assign layer numbers for gates. However, if we want to embed special sumcheck protocols in the GKR circuit, all candidate wires must be allocated in a same layer. This 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 2 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 [56], 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 one of the two GKR layer protocols in the previous subsection.
– $\mathsf{paste\_from}_{j\to i}(\mathbf{b}_y, \mathbf{b}_x)$; indicating the $\mathbf{b}_x$-th wire value in the vector $V_{j\to i}$ is pasted to $V_i(\mathbf{b}_y)$.
– $\mathsf{copy\_to}_{i\to k}(\mathbf{b}_y, \mathbf{b}_x)$; 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 introduced in the later sections, we need to support input witness from different sources (committed by polynomial commitment scheme or other predecessor circuits) and output to different target (different successor circuits). Hence we define input_paste_from and output_copy_to to split the input layer and output layer into subsets.

Based on the above circuit structure, we propose the Generalized GKR Protocols in Protocol 5 to Protocol 10, where for each layer i we first consider merging evaluations from both V_i and $V_{i\to k}$, then we use sumcheck to prove/verify the consecutive layer computations that involve both types of wire vectors discussed above. The rest of the protocol follows the same scheme as the original GKR protocol. Specifically, we divide the GKR layer protocol into the following two phases:

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

$$\begin{aligned}&\sum_{k=0}^{c-1} \alpha^k \widetilde{V}_i^{(k)}(\mathbf{Y}_k \| \mathbf{T}_k) + \sum_{k=c}^{c+c'-1} \alpha^k \widetilde{V}_{i\to \ell_{k-c}}(\mathbf{Y}_k \| \mathbf{T}_k)\\ &= \sum_{\mathbf{b}_t} \sum_{\mathbf{b}_y} \left(\sum_{k=0}^{c-1} \tilde{eq}(\mathbf{T}_k, \mathbf{b}_t) \tilde{eq}(\mathbf{Y}_k, \mathbf{b}_y) + \sum_{k=c}^{c+c'-1} \tilde{eq}(\mathbf{T}_k, \mathbf{b}_t) \widetilde{\mathsf{copy\_to}}_{i\to \ell_{k-c}}(\mathbf{Y}_k, \mathbf{b}_y) \right) \widetilde{V}_i(\mathbf{b}_y).\end{aligned} \tag{7}$$

2. Check the correctness of the wire values. At layer i, one of the two data-parallel instantiations for our gates, i.e. (5) or (6), is proved/verified.
   The layerwise adaptation of (5) is given by

$$\widetilde{V}_i(\mathbf{Y}\|\mathbf{T}) = \sum_{\substack{\mathbf{b}_s^{(0)},\ldots\mathbf{b}_s^{(d-1)},\\ \mathbf{b}_x^{(0)}\ \ \mathbf{b}_x^{(d-1)}}} \tilde{eq}(\mathbf{T},\mathbf{b}_s^{(0)},\ldots,\mathbf{b}_s^{(d-1)})G_A(\mathbf{Y},\mathbf{b}_x^{(0)},\ldots,\mathbf{b}_x^{(d-1)})\cdot\widetilde{V}_{i+1}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s^{(0)}) \cdots \widetilde{V}_{i+1}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s^{(d-1)}) + \sum_{j=0}^{c''}\tilde{eq}(\mathbf{T},\mathbf{b}_s^{(0)})\widetilde{\mathsf{paste\_from}}_{\ell'_j}(\mathbf{Y},\mathbf{b}_x^{(0)})\widetilde{V}_{\ell'_j\to i}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s^{(0)}) \tag{8}$$

   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 layerwise adaptation of (6) is given by

$$\widetilde{V}_i(\mathbf{Y}\|\mathbf{T}) = \sum_{\mathbf{b}_s}\sum_{\mathbf{b}_z}\Big(\tilde{eq}(\mathbf{T},\mathbf{b}_s)\cdot\tilde{eq}(\mathbf{Y},\mathbf{b}_z)\cdot\Big(\sum_{\mathbf{b}_x^{(0)}} G_B^{(0)}(\mathbf{b}_z,\mathbf{b}_x^{(0)})\cdot\widetilde{V}_{i+1}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s)\Big) \cdots \Big(\sum_{\mathbf{b}_x^{(d-1)}} G_B^{(d-1)}(\mathbf{b}_z,\mathbf{b}_x^{(d-1)})\cdot\widetilde{V}_{i+1}(\mathbf{b}_x^{(d-1)}\|\mathbf{b}_s)\Big) + \sum_{\mathbf{b}_x^{(0)}}\sum_{j=0}^{c''}\widetilde{\mathsf{paste\_from}}_{\ell'_j}(\mathbf{b}_z,\mathbf{b}_x^{(0)})\widetilde{V}_{\ell'_j\to i}(\mathbf{b}_x^{(0)}\|\mathbf{b}_s)\Big) \tag{9}$$

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 [7], the total cost will is $O(d\log^2 dN)$.

3.4 Multi-round GKR IOP

We can further extend the power of the GKR circuit by introducing verifier-sampled randomnesses, or challenges, to the inputs of the circuit. As illustrated by Thaler in his textbook [49], randomness significantly improves the performance of certain tasks, with negligible sacrifice to soundness. A typical example is the set equality check. The deterministic algorithm requires a quadratic complexity via naive comparison, or a quasi-linear complexity via sorting, both are very expensive for arithmetic circuits. Using randomness, however, one can leverage the grand-product argument and check the set equality with a simple circuit consisting of a linear number of gates, e.g., as in PLONK [13].

To allow the GKR circuit to take verifier challenges as inputs, we split the circuit witnesses (more precisely, inputs) into multiple segments, such that the prover commits one segments in each round. Then, we introduce several rounds called *witness-commitment rounds*, in which the parties interact with each other as follows:

1. The prover commits to the input wires of the first witness-commitment round and sends the commitments to the verifier.

2. The verifier samples the first challenge and sends to the prover.
3. The parties repeat the above steps for the other input wires in the order as they were split.

After the inputs for all the phases are committed and the challenges are ready, the prover computes all the witnesses for all the layers in the circuit, and the two parties execute the GKR protocol. At the end of the GKR protocol, the statement is reduced to an evaluation statement for the input layer, which is then reduced into the evaluation statements for the committed polynomials.

In general, the input wires are split according to their reliance on the challenges. Each wire should be committed as early as possible, i.e., in the round immediately after all the relied challenges have been sampled.

Multi-round GKR IOP

`Prover`		`Verifier`
$\mathsf{com}_0 \leftarrow \mathtt{Commit}(\tilde{f}_0(\mathbf{X}))$		
	$\xrightarrow{\mathsf{com}_0}$	
		$r_0 \leftarrow_\$ \mathbb{F}$
	$\xleftarrow{r_0}$	
Compute $\tilde{f}_{i+1}(\mathbf{X})$ with r_i		
$\mathsf{com}_{i+1} \leftarrow \mathtt{Commit}(\tilde{f}_{i+1}(\mathbf{X}))$		
	$\xrightarrow{\mathsf{com}_{i+1}}$	
		$r_{i+1} \leftarrow_\$ \mathbb{F}$
	$\xleftarrow{r_{i+1}}$	
Repeat for m rounds		Repeat for m rounds
Set $V_{\mathsf{in}} \leftarrow (f_0 \Vert f_1 \Vert \cdots \Vert f_{m-1})$.		
Compute $V_{L-1}, \ldots, V_0$.		
$(\pi_{\mathsf{gkr}}, \{y_i\}_{i=0}^{m-1}, \mathbf{z}) \leftarrow \mathtt{GKR.prove}(V_{0,\mathsf{eval}})$		
	$\xrightarrow{\pi_{\mathsf{gkr}}}$	
		$(\{y_i\}_{i=0}^{m-1}, \mathbf{z}) \leftarrow \mathtt{GKR.verify}(\pi_{\mathsf{gkr}})$
$\pi_{\mathsf{pcs}} \leftarrow \mathtt{PCS.prove}(\{\tilde{f}_i\}_{i=0}^{m-1}, \mathbf{z})$		
	$\xrightarrow{\pi_{\mathsf{pcs}}}$	
		$\mathtt{PCS.verify}(\pi_{\mathsf{pcs}}, \{(y_i, \mathsf{com}_i)\}_{i=0}^{m-1}, \mathbf{z})$

4 Asymmetric Protocols

In real world zero-knowledge proof systems, proof recursions are performed for multiple times, in order to reduce proof size. The complete procedure can be represented as a chain of provers

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

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 both 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 $\mathcal{P}_i$.

We say that the adjacent 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 **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 $\mathcal{P}_i^{(C_i)}$.

In contrary, we say 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 $\mathcal{V}_i^{(\cdot)}$ is a universal circuit that additionally takes a computation representation as input, and $\langle C_i \rangle$ denotes a representation of C_i.

Most of previous zkVM solutions set $C_0 = \mathsf{VM}$, $\mathbb{x} = (\langle \Pi \rangle, x)$, where VM is the virtual machine logic, Π is the program to be proved and x is the input of the program. Through the whole proving chain, all pairs are symmetric. However, considering the case where all provers are instantiated with succinct proving schemes, i.e., $\mathcal{P}_i$ runs in polynomial time while $\mathcal{V}_i$ runs in sub-linear time, the total cost will be dominated by $\mathcal{P}_0$, asymptotically speaking. The circuit sizes of the remaining provers will decrease continuous, where the decreasing factor varies for different proving scheme. For this reason, our solution aims to improve the performance of $\mathcal{P}_0$.

We generalize the proving chain by allowing any pair in the chain to be asymmetric. When applying this notion to zkVM, we set $C_0 = (\mathsf{VM}, \Pi)$ and $\mathbb{x} = x$, then $\mathcal{P}_0$ becomes a non-uniform prover.

For the rest of this section, we first present out intuition why a non-uniform prover is beneficial. Then we specify asymmetric protocols for two sub-circuits that our zkVM design builds on top of.

4.1 Non-uniformity Leads to Better Performance

In this section, we illustrate, via a simple example, why asymmetric protocols and non-uniform provers can be more efficient than the traditional proof systems with uniform provers.

Suppose our goal is to generate a uniform proof at the final step. In real world analogy, people deploy verifiers in smart contracts so that the proofs can be verified on-chain. A simple verification scheme for uniform proofs will result in a shorter smart contract and thus less costly. Considering the case where we want to prove a simple program that inputs an integer N and a vector $\mathbf{f}$ of size N, and outputs the sum of the elements in this vector. This is a program with a variable-length for-loop, therefore, we need extra constraints and witnesses

Protocol 2 (Asymmetric Sumcheck Protocol) *Sumcheck protocol with variable length is an interactive proof protocol between a* $\mathsf{ASC}.\mathcal{P}$ *and* $\mathsf{ASC}.\mathcal{V}$ *which is described 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* $\mathsf{idx} = (\mathsf{idx}_0, \dots, \mathsf{idx}_{n_{\mathsf{max}}}) = (0, 1, \dots, n_{\mathsf{max}})$*.*
  2. *Set* $\sigma_0 = \sigma$*,* $\mathsf{state}_{\mathsf{in\_sc}} = 1$*.*
  3. *For* $i = 0, \dots, n_{\mathsf{max}} - 1$*, 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* $f^{(i)}(0) = \sigma_i - \sum_{j=1}^{d} f^{(i)}(j)$*.*
        ii. *Randomly generate* $x_{n-i-1} \leftarrow \mathbb{F}$ *and sends it to* $\mathcal{P}$*.*
        iii. *Recover* $f^{(i)}(X)$ *from* $\left(f^{(i)}(0), \dots, f^{(i)}(d)\right)$ *and compute* $\sigma_{i+1} = f^{(i)}(x_{n-i-1})$*.*
  4. *Query the oracle* $F_{\mathsf{eval}} = F(x_0, \dots, x_{n-1})$*. If* $F_{\mathsf{eval}} = \sigma_n$*, output* $\mathtt{1}$*. Otherwise, output* $\mathtt{0}$*.*

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

Fig. 4: Asymmetric Sumcheck Protocol

to support loop control. Furthermore, to generate a uniform proof, our prover should work for all possible inputs, which, in the worst case, has a size $N_{\max}$ that can potentially be much larger than N. That is, the uniform prover will run $O(N_{\max})$ time with an extra overhead linear to $N_{\max}$ to handle loop control logic.

However, this computation can be easily proved with a non-uniform sumcheck prover in $O(N)$ time, without dealing with the for-loop logic. Although non-uniform proofs may be of variable sizes and structures, we can leverage a uniform recursive verifier to prove the verification circuit whose size is now $O(\log N_{\max})$. Note that although the verifier must pay additional (constant) overhead to handle the aforementioned variable proof sizes, the gains still overweight as long as $O(N + \log N_{\max}) < O(N_{\max})$, which is true under our assumption.

4.2 Asymmetric Sumcheck and GKR Protocol

Inspired by the above observation above, we introduce the asymmetric sumcheck (ASC) protocol and the asymmetric GKR (AGKR) protocol, where ASC implies AGKR, just as the classic sum-check implies GKR. ASC and AGKR are not only building blocks of our zkVM design. They are also good illustrations of how asymmetry leads to better efficiency. We start by describing the ASC protocol.

Asymmetric GKR (AGKR) Protocol. By replacing $\mathsf{SC.Verify}$ with $\mathsf{ASC.Verify}$, we directly obtain the basic version of AGKR, which we denote by $\mathsf{AGKR}_{\mathsf{layered}}.\mathsf{Verify}$. In our zkVM, we will be applying AGKR to two types of structured circuits, namely, the data-parallel circuits and the tree-structured circuits, explained as follows.

- **AGKR for data-parallel circuit (denoted by $\mathsf{AGKR}_{\mathsf{data\_par}}$).** We refer to Section 3 for more details.
- **AGKR for tree-structured circuit (denoted by $\mathsf{AGKR}_{\mathsf{tree}}$).** A tree-structured circuit is defined by a chain of connections of identical data-parallel sub-circuits. Suppose a sub-circuit C has a input size B and a output size 1. The corresponding tree-structured circuit consists of $N = B^n$ leaves from those data-parallel circuits. We use this structure for the following two checks:
  - the lookup argument. We implement a LogUp circuit for the sum of N fraction numbers. This is a tree-structured circuit where the leaves are pairs of denominators and numerators. Each sub-circuit $\mathcal{G}_{f_{\mathsf{sum}}}$ computes the fractional sum function $f_{\mathsf{sum}}(a, b, c, d) := (ad + bc, bd)$.
  - the set equality check. The leaves are the set elements (added a random challenge), and the sub-circuits $\mathcal{G}_{f_{\mathsf{product}}}$ are simply computing the product of two children, i.e., $f_{\mathsf{prod}}(a, b) = ab$.

We derive $\mathsf{AGKR}_{\mathsf{data\_par}}$ from $\mathsf{AGKR}_{\mathsf{layered}}$ as already described in Section 3. We construct $\mathsf{AGKR}_{\mathsf{tree}}$ by repetitively invoking $\mathsf{AGKR}_{\mathsf{data\_par}}$. We omit the details of these two protocols.

5 GKR-zkVM

In all existing zkVM designs, opcodes are executed sequentially. Additional controller witnesses are padded as prefixes and suffixes to the actual witnesses during the execution. Those include flags for the beginning and end of the execution, as well as the opcode identifiers. Such an overhead is none negligible, and can sometimes be substantial, compared with the main body of the execution. As an example, Scroll's zkEVM [46] uses over one hundred witnesses per each opcode.

In this section, we propose a new framework for zkVM. Our goal is to prove opcodes *in parallel*, and minimizing the aforementioned controlling overhead.

In our framework, the opcode list is categorised into multiple groups; each group consists of multiple instances of identical opcodes. Utilizing the data-parallel circuit, we can efficiently prove the constraints within each group. We add a program state transition logic to the opcode circuit, and use a global state to track the state transition. Finally, the global state is also proved via a chip circuit to ensure consistency across all opcodes.

Intuitively, our framework reduces the control flow from the following aspects:

- **Avoiding opcode selectors.** In a classical design, for a given opcode, the circuit needs to first assert the type of the opcode so that the corresponding sub-circuit is correctly loaded. This can be done by first looking up the opcode from a prefixed table, or via an encoding mechanism with additional selectors, and then branching through various opcode's sub-circuit. In our framework, the opcodes within the same group are of the same type. The expensive look-ups and branching are avoided. Proving an opcode execution does not incur any additional overhead.

– **Avoiding universal opcode circuit.** zkVM uses a universal, static circuit to handle various programs. That means the circuit is fixed for an unknown sequence of opcodes. To achieve this goal, in a classical design, a universal opcode circuit is adopted to host all potential opcodes. Such an opcode circuit needs to maintain orders of magnitude redundancy so that different opcodes, for instance, `add` and `mload`, share the same circuit layout. It is straightforward to see that our design eliminates such redundancy and each sub-circuit is dedicated to the given opcode.

We stress that our circuit is dynamic, in that the number of opcodes in each group varies for different programs. We use our asymmetric GKR protocol in Section 4 to handle the dynamic circuits and achieve a static verifier.

5.1 Virtual Machine Layout

We model a virtual machine as follows. A virtual machine supports a set of opcodes, where each opcode is associated with an identifier $i \in [0, Q)$. Then, a program $\Pi := (\mathsf{op}_0, \ldots, \mathsf{op}_{N_\Pi - 1}) \in [0, Q)^{N_\Pi}$ is represented by a sequence of identifiers, denoted by op_i. During the execution, we use `pc` to iterate through the list of Π. With this model in mind, we proceed to the overall picture of our framework, illustrated in Figure 5.

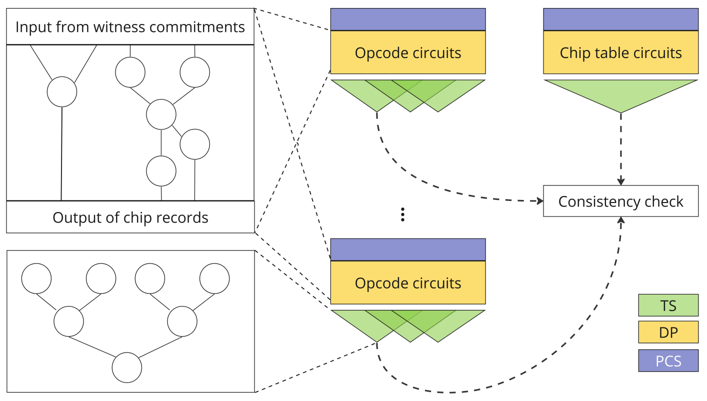


Fig. 5: GKR-zkVM Layout

1. Set op to the current opcode.
2. Read the data from the stack and the memory.
3. Apply the opcode's logic over the data.
4. Update the stack and the memory if necessary.
5. Update the program counter as defined by the current pc.

Fig. 6: Subroutine to process an opcode

The opcode computation is to read operands from the storage (stack and memory), compute results, and write them back to the storage. An opcode circuit therefore consists of similar steps as shown in Figure 6. The detailed protocol for parallel opcode computation will be presented in Section 5.2, while a brief description is shown to the left of Figure 5.

Upon collecting all chip records, the chip circuit will perform consistency checks among those records, ensuring the state transitions, memory accesses, etc. are valid and consistent across all the records. Our toolbox for such checks is set equality checks and lookup arguments. Looking ahead, we present a unified ROM and RAM circuit design in Section 7 as a practical optimization for lookup tables.

5.2 Opcode Circuit Layout and 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_{pop} values and pushes n_{push} values to the stack, accesses memory n_{mem} times. First and foremost, we need to 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 $s_{\mathsf{in}} := (\mathsf{pc}_{\mathsf{in}}, \mathsf{clk}_{\mathsf{in}}, \mathsf{top}_{\mathsf{in}})$, representing the program counter, the clock, and the top of the stack. A program state transition will generate two records, $s_{\mathsf{in}} = (\mathsf{pc}_{\mathsf{in}}, \mathsf{clk}_{\mathsf{in}}, \mathsf{top}_{\mathsf{in}})$ and $s_{\mathsf{out}} = (\mathsf{pc}_{\mathsf{out}}, \mathsf{clk}_{\mathsf{out}}, \mathsf{top}_{\mathsf{out}})$, reflecting how and when the state transition happens.

Next, to show that the state transitions correctly reflects the executions of all opcodes, we prove that all input states $S_{s_{\mathsf{in}}} = \left\{ s_{\mathsf{in}}^{(q)} \right\}_{q \in [\![N_\Pi]\!]}$ matches all output states $S_{s_{\mathsf{out}}} = \left\{ s^i_{\mathsf{out}} \right\}_{i \in [\![N_\Pi]\!]} S_{s_{\mathsf{out}}} = \left\{ s^{(q)}_{\mathsf{out}} \right\}_{q \in [\![N_\Pi]\!]}$ except for the program's very initial input state s_0 and the final output state s_{final}, i.e., we invoke a set equality check (see Section 2.4) to check that

$$S_{s_{\mathsf{in}}} \cup \{s_{\mathsf{final}}\} = S_{s_{\mathsf{out}}} \cup \{s_0\} .$$

One last caveat is that the state record is formatted as a tuple $(\mathsf{pc_{in}}, \mathsf{clk_{in}}, \mathsf{top_{in}})$. One can map this tuple into a single field element $(\mathsf{pc_{in}}, \mathsf{clk_{in}}, \mathsf{top_{in}}) \mapsto \mathbb{F}$ by linearly combining $\mathsf{pc_{in}}$, $\mathsf{clk_{in}}$ and $\mathsf{top_{in}}$ with a random challenge sampled from the transcript. As such, the elements in both sets are single field elements over which we can directly apply the set equality check.

Explicitly, we have the following two constraints.

2. **Generate state transition records.**

$$\begin{aligned} \mathsf{record}_{s_{\mathsf{in}}} &= \mathsf{RLC}\left(\mathsf{pc_{in}}, \mathsf{clk_{in}}, \mathsf{top_{in}}\right) , \\ \mathsf{record}_{s_{\mathsf{out}}} &= \mathsf{RLC}\left(\mathsf{pc_{out}}, \mathsf{clk_{out}}, \mathsf{top_{out}}\right) . \end{aligned}$$

Stack constraints. We adopt the techniques from Blum et al. [4], which also involves set equality checks. We use two sets to track all stack push and pop operations. A stack push record consists of a tuple (a, v, t) where a, v, 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. Notice that, in the opcode circuit we have already asserted that (a, v, t) and (a, v, t') are well-formed, including assertions such as the stack address a does not under or overflow, and the time stamp $t < \mathsf{clk_{in}}$. 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 $S_{\mathsf{pop}} = \{(a, v, t')_i\}_{i \in [\![N_{\mathsf{pop}}]\!]}$. Similar to program state records, we also use random linear combinations to map stack push and pop records (a, v, t) to field elements before the actual set equality checks. And finally a set equality check

$$S_{\mathsf{push}} = S_{\mathsf{pop}}$$

ensures the stack is consistent.

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

3. **Generate stack pop records.** Suppose the values from the stack are $v_{\mathsf{pop}}^{(0)}, \dots, v_{\mathsf{pop}}^{(n_{\mathsf{pop}}-1)}$, written to the stack at the clock $\mathsf{clk}_{\mathsf{pop}}^{(0)}, \dots, \mathsf{clk}_{\mathsf{pop}}^{(n_{\mathsf{pop}}-1)}$:

$$\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \{(0 \le \mathsf{top}_{\mathsf{in}} - n_{\mathsf{pop}} < \mathsf{size}_{\mathsf{stack}})\},$$
$$\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \left\{(\mathsf{clk}_{\mathsf{pop}}^{(i)} < \mathsf{clk}_{\mathsf{in}})\right\}$$
$$\mathsf{record}_{\mathsf{pop}} = \mathsf{record}_{\mathsf{pop}} \cup \left\{\mathsf{RLC}\left(\mathsf{top}_{\mathsf{in}} - i - 1, v_{\mathsf{pop}}^{(i)}, \mathsf{clk}_{\mathsf{pop}}^{(i)}\right)\right\}, \forall 0 \le i < n_{\mathsf{pop}}.$$

4. **Generate stack push records.** Suppose the values from the stack are $v_{\mathsf{push}}^{(0)}, \dots, v_{\mathsf{push}}^{(n_{\mathsf{push}}-1)}$:

$$\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \{(0 \le \mathsf{top}_{\mathsf{in}} - n_{\mathsf{pop}} + n_{\mathsf{push}} - 1 < \mathsf{size}_{\mathsf{stack}})\},$$
$$\mathsf{record}_{\mathsf{push}} = \mathsf{record}_{\mathsf{push}} \cup \left\{\mathsf{RLC}\left(\mathsf{top}_{\mathsf{in}} - n_{\mathsf{pop}} + i, v_{\mathsf{push}}^{(i)}, \mathsf{clk}_{\mathsf{in}}\right)\right\}, \forall 0 \le i < n_{\mathsf{push}}.$$

Memory constraints. The memory checking mechanism also stems from Blum et al. [4], and is built upon the stack checking constraints in the previous subsection. A memory record is also of the form (a, v, t). A memory write operation will generate a pair of records, popping the old data and pushing the new data to the same memory address simultaneously. The memory load function works identical to memory write, where we pop and push the same data simultaneously. Accordingly, a set equality check over the push record set and the pop record set, together with the same random linear combination method in the previous paragraph, ensures the memory is consistent.

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

5. **Generate memory records (If load)**. Suppose the loaded values are $v_{\mathsf{load}}^{(0)}, \dots, v_{\mathsf{load}}^{(n_{\mathsf{mem}}-1)}$ with address $a_{\mathsf{mem}}^{(0)}, \dots, a_{\mathsf{mem}}^{(n_{\mathsf{mem}}-1)}$, written to the memory at the clock $\mathsf{clk}_{\mathsf{load}}^{(0)}, \dots, \mathsf{clk}_{\mathsf{load}}^{(n_{\mathsf{mem}}-1)}$:

$$\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \left\{(\mathsf{clk}_{\mathsf{load}}^{(i)} < \mathsf{clk}_{\mathsf{in}})\right\},$$
$$\mathsf{record}_{\mathsf{load}} = \mathsf{record}_{\mathsf{load}} \cup \left\{\mathsf{RLC}\left(a_{\mathsf{mem}}^{(i)}, v_{\mathsf{load}}^{(i)}, \mathsf{clk}_{\mathsf{load}}^{(i)}\right)\right\}, \forall 0 \le i < n_{\mathsf{mem}},$$
$$\mathsf{record}_{\mathsf{store}} = \mathsf{record}_{\mathsf{store}} \cup \left\{\mathsf{RLC}\left(a_{\mathsf{mem}}^{(i)}, v_{\mathsf{load}}^{(i)}, \mathsf{clk}_{\mathsf{in}}^{(i)}\right)\right\}, \forall 0 \le i < n_{\mathsf{mem}}.$$

6. **Generate memory records (If store).** Suppose the stored values are $v_{\mathsf{store}}^{(0)}, \dots, v_{\mathsf{store}}^{(n_{\mathsf{mem}}-1)}$ with addresses $a_{\mathsf{mem}}^{(0)}, \dots, a_{\mathsf{mem}}^{(n_{\mathsf{mem}}-1)}$, overwrite the values $v_{\mathsf{load}}^{(0)}, \dots, v_{\mathsf{load}}^{(n_{\mathsf{mem}}-1)}$ written to the memory at the clock $\mathsf{clk}_{\mathsf{load}}^{(0)}, \dots, \mathsf{clk}_{\mathsf{load}}^{(n_{\mathsf{mem}}-1)}$:

$$\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \left\{(\mathsf{clk}_{\mathsf{load}}^{(i)} < \mathsf{clk}_{\mathsf{in}})\right\},$$
$$\mathsf{record}_{\mathsf{load}} = \mathsf{record}_{\mathsf{load}} \cup \left\{\mathsf{RLC}\left(a_{\mathsf{mem}}^{(i)}, v_{\mathsf{load}}^{(i)}, \mathsf{clk}_{\mathsf{load}}^{(i)}\right)\right\}, \forall 0 \le i < n_{\mathsf{mem}},$$
$$\mathsf{record}_{\mathsf{store}} = \mathsf{record}_{\mathsf{store}} \cup \left\{\mathsf{RLC}\left(a_{\mathsf{mem}}^{(i)}, v_{\mathsf{store}}^{(i)}, \mathsf{clk}_{\mathsf{in}}^{(i)}\right)\right\}, \forall 0 \le i < n_{\mathsf{mem}}.$$

Chip lookup constraints. Lookup arguments are a common method to deal with complex and non-linear operations in zkVMs. The prover pre-compute a lookup table, in the form of a set of key-value store. For the rest of this subsection, we use bytecode chip as an example. A bytecode table's record is of the format $T := \{(\texttt{pc}_i, \texttt{opcode}_i)\}$. During the execution, the prover maintains an append only set S for the actual $\texttt{pc}$ and $\texttt{opcode}$ that is been executed. Then, the prover append the key and value pair $(\texttt{pc}_{\texttt{cur}}, \texttt{opcode}_{\texttt{cur}})$ to S. The lookup is correct if $S \subseteq T$.

Concretely we use the *LogUp* protocol introduced by Papini et al. [28], which is constructed based on the circuit $\mathcal{G}_{\mathsf{fsum}}$ (see Section 4.2) to compute fractional summation, therefore can be instantiated with our $\mathsf{AGKR}_{\mathsf{tree}\mathcal{G}_{\mathsf{fsum}}}$. The constraints are defined as follows.

7. **Generate bytecode chip lookup.**

$$\mathsf{record}^{A}_{\mathsf{bytecode}} = \mathsf{RLC}\left(\mathsf{pc}_{\mathsf{in}}, \mathsf{op}\right).$$

5.3 Chip Computation Constraints

Memory initialization and finalization constraints. Depending on the architecture, some virtual machine initializes the memory with 0s at program initialization, and allows for access without declaration; others requires each memory address to be accessible only after a manual declaration and initialization. In our zkVM, we unify the two architectures via writing a default value to the memory when it is empty at the time of the first load operation. This is done at $\mathsf{clk}_{\mathsf{in}} = 0$. Suppose there are $n_{\mathsf{mem,init}}$ of cells with addresses $a^{(0)}_{\mathsf{mem,init}}, \dots, a^{(n_{\mathsf{mem,init}}-1)}_{\mathsf{mem,init}}$ to be initialized, then we have a circuit with $\mathsf{record}_{\mathsf{store}}$ defined as

$$\mathsf{record}_{\mathsf{store}} = \mathsf{record}_{\mathsf{store}} \cup \left\{(a^{(i)}_{\mathsf{mem,init}}, 0, 0)\right\}, \forall 0 \le i < n_{\mathsf{mem,init}}. \tag{10}$$

We also finalize memory by clearing the memory addresses whose values have not been manually removed by the opcodes. In our zkVM, this is done at $\mathsf{clk}_{\mathsf{final}}$. Suppose there are $n_{\mathsf{mem,finl}}$ of cells with addresses $a^{(0)}_{\mathsf{mem,finl}}, \dots, a^{(n_{\mathsf{mem,finl}}-1)}_{\mathsf{mem,finl}}$ and values $v^{(0)}, \dots, v^{(n_{\mathsf{mem,finl}}-1)}$ to be removed from the memory, then we have a circuit with the output $\mathsf{record}_{\mathsf{load}}$ defined as

$$\mathsf{record}_{\mathsf{load}} = \mathsf{record}_{\mathsf{load}} \cup \left\{(a^{(i)}_{\mathsf{mem,finl}}, v^{(i)}, \mathsf{clk}_{\mathsf{final}})\right\}, \forall 0 \le i < n_{\mathsf{mem,finl}}. \tag{11}$$

Stack initialization and finalization constraints. Handling stacks is a bit simpler, because stack is always empty at the initiation. When finalizing the stack with $n_{\mathsf{pop,finl}}$ elements after the program halt, obviously its address is from 0 to $n_{\mathsf{pop,finl}} - 1$, then we have an extra circuit with the output $\mathsf{record}_{\mathsf{pop}}$ defined as

$$\mathsf{record}_{\mathsf{pop}} = \mathsf{record}_{\mathsf{pop}} \cup \left\{(i, v^{(i)}, \mathsf{clk}_{\mathsf{final}})\right\}, \forall 0 \le i < n_{\mathsf{pop,finl}}. \tag{12}$$

Table constraints. For each chips proved by lookup arguments, we need a circuit to initialize the table items and generate the corresponding table records. For example, for the bytecode chip with the table record set $\mathsf{record}^T_{\mathsf{bytecode}}$, we have the following constraints in that circuit:

$$\mathsf{record}^T_{\mathsf{bytecode}} = \mathsf{record}^T_{\mathsf{bytecode}} \cup \{(i, \mathsf{op}_i)\}, \forall 0 \le i < N_\Pi \tag{13}$$

Each opcode additionally has several tree structured circuits, shown as the the green triangle as in Figure 5. Circuits are used to compute the product or sum of every chip. Later on, within the chip table circuit, there is also a tree circuit which performance a same computation. The products, denoted as $\left\{\delta^{(\mathsf{chip})}\right\}$ for all chips, and the summation, denoted as $\left\{\sigma^{(\mathsf{chip})}\right\}$, will be send to the verifier. A consistency check is then performed across different trees by the verifier.

5.4 Proving and Verification Protocols

The detailed description of the constraint system has already been presented in Section 5.2. The circuit structure and proving process in presented in Figure 7.

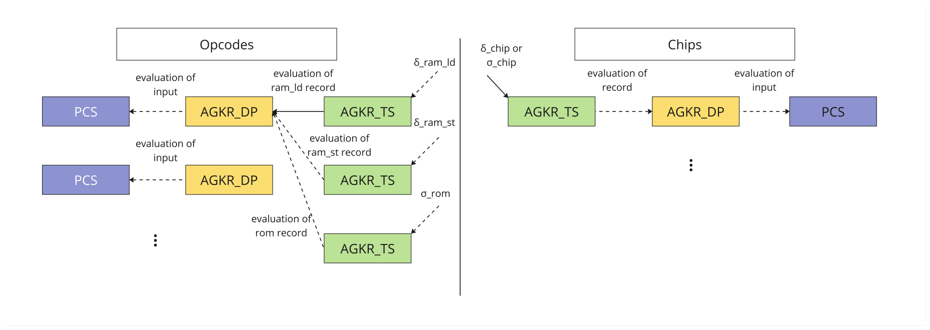


Fig. 7: Circuit Structure in GKR-zkVM

Proving cost analysis. Suppose for each program the number of types of opcodes is a constant, which is common in real-world scenarios. Assume any opcode in a program is executed for at most 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 and the proof size are both $O(\log^2 N)$, where one $O(\log N)$ comes from the sum-check and another factor of $\log N$ comes from the tree-structure circuits that have $O(\log N)$ layers.

Note that the verification and proof size can be further reduced by recursion, which is a common practice in the zero-knowledge community.

Protocol 3 (GKR-zkVM) *The proving-verification process proceeds as follows:*

- ***Prover:***
  1. *Prover sends auxiliary information to the verifier, including the program's input and output length, the length of the public instance and the bytecode length. Notice that some data field, such the bytecode, may be already known by the verifier.*
  2. *Prover and verifier exchange witness commitments and challenges in several rounds.*
  3. *Prover sends* $\left\{\delta^{(\mathsf{chip})}\right\}$, $\left\{\sigma^{(\mathsf{chip})}\right\}$, *and generates all GKR proofs for the circuits, following the flow as shown in Figure 7. Prover sends those messages to the verifier.*
  4. *Prover opens polynomial commitments and send the openings to the verifier.*
- ***Verifier:***
  1. *Verifier receives the auxiliary information.*
  2. *Verifier exchanges witness commitments and challenges with 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* $\left\{\delta^{(\mathsf{chip})}\right\}$ *and* $\left\{\sigma^{(\mathsf{chip})}\right\}$.

6 GKR-zkVM Pro

In this section, we present an optimized design building on top of previous section.

The design in the previous section has two minor drawbacks. First, our parallelization was done at the opcode level. Most of the opcodes, such as ADD, MUL, etc. have *very shallow circuits*. One feature of GKR, when it is turned into a non-interactive protocol, is that one is only required to commit to the input and output layers of the protocol, not the intermediate layers. Shallow circuits mean we have to commit to more witnesses.

The other direction of optimization is to build *structure awareness* for the prover. The design in the previous version treats every opcode separately, and does not exploit the fact that most programs have repeated patterns. As alluded earlier, structure awareness is a common means in programming language for compilers to optimize execution. We borrow this intuition and apply it to our prover.

When applying this intuition to our prover, we have the following observations. In a typical virtual machine, programs use a stack to handle data flow among opcodes. These data flow are currently represented by records, checked by our chip circuit. Notice that when a segment of program does not contain branches, the data flow remains identical within this segment. This leads to the idea of handling data flow at the basic block level.

Now we are ready to present the details of this improvement. We note that cross this section we are using stack based virtual machines as examples, though our method is also applicable to register based virtual machines.

6.1 Critical Observations in The Program Execution

Consider that GKR protocol is originally to deal with fixed circuits instead of dynamic execution trace, we try to convert the execution trace "more like" a circuit. In this subsection, we start from some observations of stacks and basic blocks and then propose our enhanced design GKR-zkVM Pro. Finally, we demonstrate that how GKR is applied to reduce the stack operations to be proved.

Our view of stack operations Stack operation, generally speaking, is to connect a value to two opcodes where the value is generated and consumed. Consider the following example. For an opcode execution sequences $\Pi := (\ldots, \mathsf{ADD}, \mathsf{MUL}, \ldots)$, the element popped in MUL is the result pushed in ADD. Therefore, if we visualize the opcodes as gates in a circuit, the stack operations will form the wires connecting the ADD and MUL gates. For simplicity, we call the opcode that generates the stack element the *generator* (i.e., the ADD gate) and the one that consumes the stack element the *consumer*. Extending this notion over the whole trace, we can construct a circuit for the program Π.

However, the existences of branches make the circuit dynamic. That is the stack operation will connect a generator to multiple consumers depending on the branching condition. Our solution to the branching issue is to work only with chunks of opcode that does not contain branches. This notion is known as *basic block* (BB) in the compiler design domain, and is useful to partition a program. Formally,

Definition 3 (Basic block [51]). *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.

Further analyzing of basic blocks leads to the following observations:

- The stack behaviors are identical for executions of a same basic block. Then, if we treat a BB as a fixed sub-circuit; a program will contain multiple identical BB sub-circuits, also known as data parallel circuits. A typical example here is the code within a loop that gets executed multiple times.

– The opcodes are always executed sequentially inside a basic block. Therefore, we only need to check the state validation in the first and last opcodes in a basic block.

In summary, our key intuition is that basic blocks can be processed with data-parallel circuits.

6.2 A Toy Example

Figure 8 presents a toy example. This example program (Figure 8a) reads an input integer n and executes a loop with n repetitions to compute the sum of a variable-length array. After compiling this program into assembly (see Figure 8b), we identify three basic blocks in the bytecode: the first basic block (B1) consists of the first two opcodes. Notice that the third opcode is the destination of a `CJMP` opcode. The second basic block (B2) starts from the third opcode and finishes at the `CJMP` opcode. Finally, the last basic block (B3) consists of the last opcode. Note that when B1 is finished, the only possible destination is B2; whereas B2 may be followed by either itself or B3, i.e., a branch. A complete workflow is illustrated in Figure 8c.

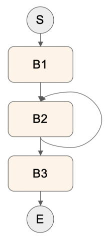
```c
int n = input();

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

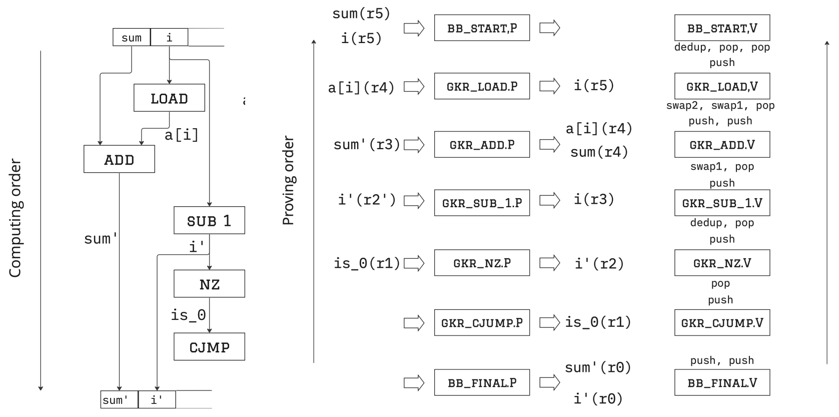


(c) Basic Block

Fig. 8: Program and its Basic Block for the Toy Example

Next, we explain how to generate a proof for this program exploiting this structure. The entire procedure follows the same workflow as GKR-zkVM, except that the opcode circuits are replaced with basic block circuits. We show how to prove B2, the most complex basic block in this example. As shown in Figure 9a, the circuits accomplish three tasks: add the current entry to the sum, decrement the counter, and decide whether to jump. To prove this computation, the prover traverses the instructions in the reverse order, and invokes the GKR prover to reduce the outputs to the inputs, as illustrated in Figure 9b, where the left side is the output for each circuit.

Finally, the verifier validates the proof in the same order as the proving.



(a) Computing B2 Circuits (b) Prove and Verify B2 Circuits

Fig. 9: Prove and Verify Basic Block in the Toy Example

6.3 Virtual Machine Layout and Opcode Layout

Before proceed further, let us first review the difference between this new design and GKR-zkVM. Recall that in GKR-zkVM, we have already modelled $\Pi := (\mathsf{op}_0, \ldots, \mathsf{op}_{N_\Pi - 1}) \in [0, Q)^{N_\Pi}$ as in Section 5.1, and same type of opcodes are handle by a single data-parallel circuit.

Let us further assume $\Pi := (b_0, \ldots, b_{B_\Pi - 1}) \in [0, R)^{B_\Pi}$ where B_Π is the total number of basic blocks in Π, R is the number of unique basic blocks, and b_i is the identifier for the i-th basic block. Each basic block BB is represented by its pc counters ($q_{\mathsf{start}}...q_{\mathsf{end}}$), a sub-vector of Π.

Similar to GKR-zkVM, our view of a virtual machine circuit still consists of the opcode circuits and the chip circuits. The difference here is that the opcode circuits are now grouped by the basic blocks rather than their individual opcode type. Specifically, for each basic block b_i in the program, we have a BB start circuit, a BB final circuit, and a list of opcode circuits, each for one opcode in the basic block, respectively. We expect a basic block circuit to be repeated for multiple times, and forms a data parallel circuit, and eventually get proved via the AGKR_DP protocol.

6.4 Constraints

Following the above basic block abstraction, we claim that we no longer need to handle the stack and global states at the opcode level. We define a basic block start circuit and end circuit to handle the storage. Within each basic block, the opcodes are executed sequentially, and the state transitions are implicitly guaranteed by the gates connecting the opcodes, as explained in Section 6.1.

GKR-zkVM Pro improves upon GKR-zkVM in the following aspects:

- First, the opcode circuits inside a basic block no longer handles the stack operations, also avoids checking the range of stack top and timestamp.
- Second, the opcodes inside a basic block are guaranteed to be executed sequentially, and therefore we no longer need to handle bytecode lookups for the current opcode. The only exception is the last opcode, which is always a jump operation.
- Thirds, the opcode circuit does not need to handle global states. The global state records are generated only at the beginning and the end of a basic block.

Constraints in Basic Block Circuits. The basic block start circuits and final circuits are used to assert the validity of stack operations and global state updates.

An heuristic here is that from a basic block point of view, the stack changes, in terms of pop and push operations, will be much smaller than the sum of all its opcodes'. The justification is that we expect that consecutive opcodes' stack operations to cancel each other.

Our basic block's stack circuit is therefore designed as follows:

- The basic block start circuit reads all the values in the stack that will be touched by the opcodes inside this basic block.
- The relative positions of these values to the stack top are fixed and remain constant for this basic block when generating all its opcode circuit.
- The basic block final circuit is responsible for updating the global state, including the new program counter, the clock and the stack top.
- The basic block final circuit also finalizes the operations on the stack.

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_{pop} values at the beginning and pushes n_{push} values to the stack in the end, then it has the following constraints:

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

$$\mathsf{record}_{s_{\mathsf{in}}} = \mathsf{RLC}\left(\mathsf{pc}_{\mathsf{in}}, \mathsf{clk}_{\mathsf{in}}, \mathsf{top}_{\mathsf{in}}\right).$$

   BB final circuit checks the correctness of $(\mathsf{pc}_{\mathsf{out}}, \mathsf{clk}_{\mathsf{out}}, \mathsf{top}_{\mathsf{out}})$ according to the BB's structure, and then generate a state record:

$$\mathsf{record}_{s_{\mathsf{out}}} = \mathsf{RLC}\left(\mathsf{pc}_{\mathsf{out}}, \mathsf{clk}_{\mathsf{out}}, \mathsf{top}_{\mathsf{out}}\right).$$

2. **Initialization of the stack.** The BB start circuit loads values from the 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 $v_{\mathsf{pop}}^{(0)}, \ldots, v_{\mathsf{pop}}^{(n_{\mathsf{pop}}-1)}$ and let $\mathsf{clk}_{\mathsf{pop}}^{(0)}, \ldots, \mathsf{clk}_{\mathsf{pop}}^{(n_{\mathsf{pop}}-1)}$ be the clock that stack is written, respectively. Then,

$$\begin{aligned}
&\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \{(0 \leq \mathsf{top}_{\mathsf{in}} - n_{\mathsf{pop}} < \mathsf{size}_{\mathsf{stack}})\},\\
&\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \left\{(\mathsf{clk}_{\mathsf{pop}}^{(i)} < \mathsf{clk}_{\mathsf{in}})\right\}\\
&\mathsf{record}_{\mathsf{pop}} = \mathsf{record}_{\mathsf{pop}} \cup \left\{\mathsf{RLC}\left(\mathsf{top}_{\mathsf{in}} - i - 1, v_{\mathsf{pop}}^{(i)}, \mathsf{clk}_{\mathsf{pop}}^{(i)}\right)\right\}, \forall 0 \leq i < n_{\mathsf{pop}}.
\end{aligned}$$

3. The BB final circuit takes values as the input that are computed within the block and not yet consumed. It writes the, back those values to the stack. Denote this values by $v_{\mathsf{push}}^{(0)}, \ldots, v_{\mathsf{push}}^{(n_{\mathsf{push}}-1)}$, then:

$$\begin{aligned}
&\mathsf{record}_{\mathsf{range}} = \mathsf{record}_{\mathsf{range}} \cup \{(0 \leq \mathsf{top}_{\mathsf{in}} - n_{\mathsf{pop}} + n_{\mathsf{push}} - 1 < \mathsf{size}_{\mathsf{stack}})\},\\
&\mathsf{record}_{\mathsf{push}} = \mathsf{record}_{\mathsf{push}} \cup \left\{\mathsf{RLC}\left(\mathsf{top}_{\mathsf{in}} - n_{\mathsf{pop}} + i, v_{\mathsf{push}}^{(i)}, \mathsf{clk}_{\mathsf{in}}\right)\right\}, \forall 0 \leq i < n_{\mathsf{push}}.
\end{aligned}$$

6.5 Proving And Verification Protocols

The complete protocol is presented in Protocol 4. The circuit structure is illustrated in Figure 10, implying a non-uniform proving protocol. Note that we cannot derive the verification process via a simple combination of AGKR protocols. The connections of opcodes within a basic block influence the circuit layout, and a uniform verifier should be able to infer the layout via the opcode sequence.

Cost analysis. GKR-zkVM Pro is a structure-aware protocol for programs with applicable structures. Suppose the program is of size N_Π, consisting of B_Π basic blocks. When the number of iterations of each basic block is bounded by M, and every basic block executes at most a constant number of stack operations, the prover proves $O(B_\Pi M)$ number of stack operations. Furthermore, the number of integrity checks associated with the stack is also reduced to $O(B_\Pi M)$, including range checks on the stack top pointer, and the comparison between stack timestamp and the current clock. GKR-zkVM Pro becomes increasingly efficient when $B_\Pi M$ is significantly smaller than N.

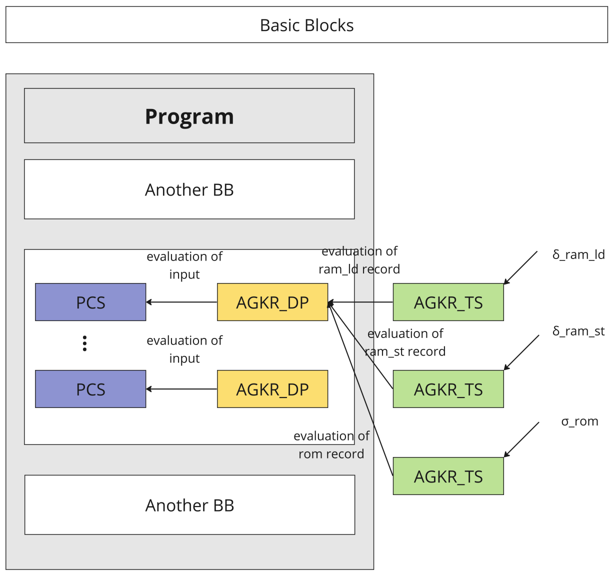


Fig. 10: Structure of Basic Blocks in GKR-zkVM Pro

This cost improvement on the prover side comes at a cost of a slight increment to the verifier's work. In general, the stack operations that was previously done by the opcode circuits are now performed by the verifier's. This is executed once for every $\mathsf{op}_i \in \Pi$, regardless of the number of repetitions. Consequently, the verifier's cost is increased by $O(N_\Pi)$ in the worst case.

In summary,

- The running time of the prover is $O(N_\Pi M)$.
- The verification cost and proof size are $O(N_\Pi \log^2 M)$.

Note that M here is usually smaller than the N, the number of opcode repetitions in GKR-zkVM. Furthermore, the total number of stack operations on the prover and the verifier sides has been reduced to $O(N_\Pi + B_\Pi \cdot M)$, which, for most programs, is significantly less than the number of operations in their executions. Also, we moved bytecode checkings from the prover to the verifier. Similar to the above analysis, the combined cost for bytecode operations are reduced.

7 GKR-zkVM's ROM and RAM

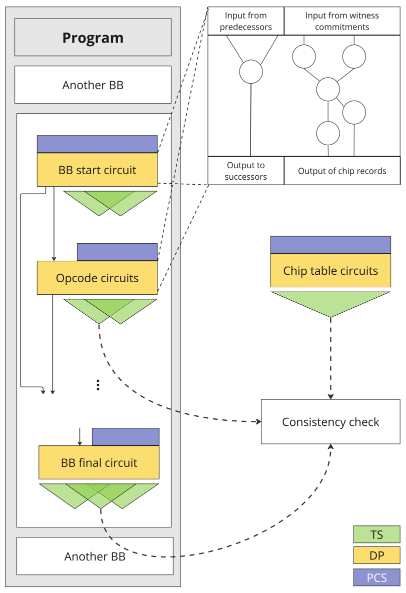
In both GKR-zkVM and GKR-zkVM Pro protocols, we have introduced multiple tables for memory constraints, chip lookup constraints, etc. In this section, we

Protocol 4 (GKR-zkVM Pro) *The proving and verification proceed as follows:*

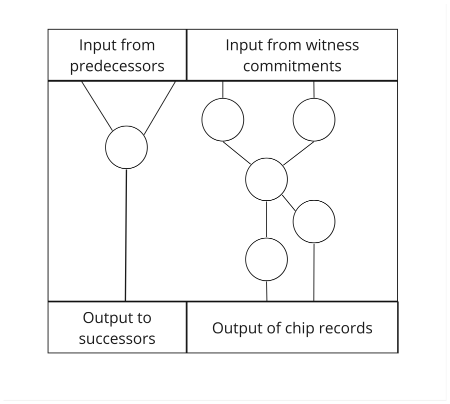
- ***Prover:***
  1. *Prover sends auxiliary information to the verifier, including the program's input and output length, the length of the public instance and the basic block information. Notice that some data field, such the bytecode, may be already known by the verifier.*
  2. *Prover and verifier exchange witness commitments and challenges in several rounds.*
  3. *Prover sends* $\left\{\delta^{(\mathsf{chip})}\right\}$, $\left\{\sigma^{(\mathsf{chip})}\right\}$*, and generates all GKR proofs for the circuits, following the flow as shown in Figure 10. This consists of basic block circuits, including BB start and BB Final, opcode circuits, chip table circuits and tree circuits for computing summations and products. Prover sends those messages to the verifier.*
  4. *Prover opens polynomial commitments and send the openings to the verifier.*
- ***Verifier:***
  1. *Verifier receives the auxiliary information.*
  2. *Verifier exchanges witness commitments and challenges with prover in several rounds.*
  3. *Verifier receives GKR proofs from the prover in the order corresponding to the program. The verifier iterates through the program and the corresponding circuits reversely, from the last opcode to the first.*
     (a) *If the circuit is a BB final circuit, the verifier verifies it and, initialize the verifier's stack using the input evaluations.*
     (b) *If the circuit is a opcode circuit, the verifier fetches the evaluations from the stack, verify the GKR proof, and update the stack.*
     (c) *If the circuit is a BB start circuit, the verifier fetches all the remaining evaluations in the stack, verifies the GKR proof, and ensures the stack is empty.*
  4. *Verifier receives polynomial commitment openings and verifies them.*
  5. *Verifier checks the correctness of the product of* $\left\{\delta^{(\mathsf{chip})}\right\}$*, and* $\left\{\sigma^{(\mathsf{chip})}\right\}$.

show how to reduce those tables into merely two tables: a ROM table and a RAM table. At a high level, the ROM table is the concatenation of all read-only tables, with an additional key indicating which table the entry is originated from. The RAM table is used for operations on mutable tables. In the case of zkVM, these particularly refer to the stack, memory, and global state checking operations. The reduction in the number of tables leads to less number of circuits for the set equality check and lookup arguments.

ROM Table. The ROM table collects all the lookup tables into a single table, thus merging multiple lookup arguments into one. This optimization is formalized into the following fact.



(a) GKR-zkVM Pro zkVM layout



(b) GKR-zkVM Pro opcode layout

Fig. 11: GKR-zkVM Pro Layouts

Fact 1 *Given a set* $A = \{\mathbf{a}_i\}_{i=0}^{m_a-1}$ *of input vectors and set* $T = \{\mathbf{t}_i\}_{i=0}^{m_t-1}$ *of tables, for every* i *let* $\mathbf{a}'_i = \{('\mathsf{tag}'_i, a_{i,j})\}_{j=0}^{|\mathbf{a}_i|}$ *and* $\mathbf{t}'_i = \{('\mathsf{tag}'_i, t_{i,j})\}_{j=0}^{|\mathbf{t}_i|}$. *Let* $\mathbf{A} := \mathbf{a}'_1 \| \cdots \| \mathbf{a}'_{m_a}$ *and* $\mathbf{T} := \mathbf{t}'_1 \| \cdots \| \mathbf{t}'_{m_t}$. *Then all the pairs in* $\mathbf{V}$ *is a subset of all the pairs in* $\mathbf{T}$ *if and only if for every* i *from* 0 *to* $m-1$*, the entries of* $\mathbf{v}_i$ *is a subset of the entries in* $\mathbf{t}_i$.

Based on this fact, we introduce the ROM table whose constraints are defined as follows. We first initialize $\mathsf{record}_T^{\mathsf{ROM}}$ as the empty list. Suppose then for each chip operation with record a on the chip $'\mathsf{tag}'$, we introduce the following constraints

$$\mathsf{record}_T^{\mathsf{ROM}} = \mathsf{record}_T^{\mathsf{ROM}} \cup \{\mathsf{RLC}\,('\mathsf{tag}' \| a)\} \tag{14}$$

For example, suppose we have a range check chip and a bytecode chip, with tables defined as $[0, 2^b)$ and $\{(i, \mathsf{op})\}_{0 \le i < N_\Pi}$, respectively. We initialize an empty list $\mathsf{record}_A^{\mathsf{ROM}}$, and the constraints for checking these two operations are respectively

$$\begin{aligned} \mathsf{record}_A^{\mathsf{ROM}} &= \mathsf{record}_A^{\mathsf{ROM}} \cup \{\mathsf{RLC}\,('\mathsf{range}', a)\} && \text{for a range check on } a \\ \mathsf{record}_A^{\mathsf{ROM}} &= \mathsf{record}_A^{\mathsf{ROM}} \cup \left\{\mathsf{RLC}\left('\mathsf{bytecode}', \mathsf{pc}, \mathsf{op}\right)\right\} && \text{for looking up } (\mathsf{pc}, \mathsf{op}) \end{aligned} \tag{15}$$

Merging multiple lookup arguments into a single one leads to less number of circuits. Consider two lookup arguments, with a lookup vector size of N and a table size also of N. Executing the two lookup arguments separately requires $2 \log N$ rounds, whereas the merged lookup argument requires $\log 2N = 1 + \log N$ rounds. This significantly reduces the cost for the verifier.

RAM Table. We use a RAM table to handle the memory, the stack operations, and the global state checking. Unlike the ROM table where the data is stored once and remains static for its whole lifetime, the RAM table must handle the cases where the data are mutable. Therefore, we use the offline memory checking [4] technique to transform these checks into a set-equality argument described in Section 5.2.

Now, let us explain how to perform all checks with a single RAM table. First, recall that in the offline memory checking, both the reading and the writing operations generate two records: one read and one write at the same address. This inspires us to define a new memory model, which we refer to as *once access memory* (OAM) which is a special kind of read-write memory.

Definition 4. *An OAM with address space* $0, \cdots, M-1$ *is a random-access, read-write memory of this address space, with the following restriction: an OAM's address can only be accessed by write and read operations in an interleaved sequence, starting from write and ending with read. In another word, for each address, the operations to this address must follow the write, read, write, read,* $\cdots$*, write, read pattern.*

Let $\mathsf{record}_{\mathsf{load}}^{\mathsf{OAM}}$ and $\mathsf{record}_{\mathsf{store}}^{\mathsf{OAM}}$ be the tables that store the sequence of OAM operations. Then the constraints for these two operations are defined as follows:

- OAM read with a value v at an address a that was written at a timestamp clk, with an original chip tag $'\mathsf{tag}'$:

$$\mathsf{record}_{\mathsf{load}}^{\mathsf{OAM}} = \mathsf{record}_{\mathsf{load}}^{\mathsf{OAM}} \cup \{\mathsf{RLC}\,('\mathsf{tag}', a, v, \mathsf{clk})\} \tag{16}$$

- OAM write with a value v to an address a at a timestamp clk, with an original chip tag $'\mathsf{tag}'$:

$$\mathsf{record}_{\mathsf{store}}^{\mathsf{OAM}} = \mathsf{record}_{\mathsf{store}}^{\mathsf{OAM}} \cup \{\mathsf{RLC}\,('\mathsf{tag}', a, v, \mathsf{clk})\} \tag{17}$$

Intuitively, we view each value as an object generated by a write operation and consumed by a read operation.

Under our OAM model, storage operations, including read-write memory operations, stack operations, and global state updates, can all be simulated with OAM operations. Although here we will explicitly specify some dedicated address for the stack operations, this address is actually maintained by the push and pop logic in the opcodes.

We omit the details of range check operations that are to be used in the following constraints. They can be directly instantiated from ROM as shown in Equation 15.

- For a memory read operation at address a and a timestamp clk, reading a value v_{load} that was written at $\mathsf{clk}_{\mathsf{load}}$,

$$\begin{aligned}
\mathsf{record}_{\mathsf{range}} &= \mathsf{record}_{\mathsf{range}} \cup \{(\mathsf{clk}_{\mathsf{load}} < \mathsf{clk})\} \\
\mathsf{record}^{\mathsf{OAM}}_{\mathsf{load}} &= \mathsf{record}^{\mathsf{OAM}}_{\mathsf{load}} \cup \{\mathsf{RLC}\,('\mathsf{memory}', a_{\mathsf{mem}}, v_{\mathsf{load}}, \mathsf{clk}_{\mathsf{load}})\} \\
\mathsf{record}^{\mathsf{OAM}}_{\mathsf{store}} &= \mathsf{record}^{\mathsf{OAM}}_{\mathsf{store}} \cup \{\mathsf{RLC}\,('\mathsf{memory}', a_{\mathsf{mem}}, v_{\mathsf{load}}, \mathsf{clk})\}\,.
\end{aligned}$$

- For a memory write operation at address a and a timestamp clk, overwriting the value v_{load} that was written at $\mathsf{clk}_{\mathsf{load}}$ with the new value v_{store}

$$\begin{aligned}
\mathsf{record}_{\mathsf{range}} &= \mathsf{record}_{\mathsf{range}} \cup \{(\mathsf{clk}_{\mathsf{load}} < \mathsf{clk})\} \\
\mathsf{record}^{\mathsf{OAM}}_{\mathsf{load}} &= \mathsf{record}^{\mathsf{OAM}}_{\mathsf{load}} \cup \{\mathsf{RLC}\,('\mathsf{memory}', a_{\mathsf{mem}}, v_{\mathsf{load}}, \mathsf{clk}_{\mathsf{load}})\} \\
\mathsf{record}^{\mathsf{OAM}}_{\mathsf{store}} &= \mathsf{record}^{\mathsf{OAM}}_{\mathsf{store}} \cup \{\mathsf{RLC}\,('\mathsf{memory}', a_{\mathsf{mem}}, v_{\mathsf{store}}, \mathsf{clk})\}\,.
\end{aligned}$$

- For a stack PUSH operation that writes value v_{push} to a at time clk

$$\begin{aligned}
\mathsf{record}_{\mathsf{range}} &= \mathsf{record}_{\mathsf{range}} \cup \{(0 \le a < \mathsf{size}_{\mathsf{stack}})\}\,, \\
\mathsf{record}^{\mathsf{OAM}}_{\mathsf{store}} &= \mathsf{record}^{\mathsf{OAM}}_{\mathsf{store}} \cup \left\{\mathsf{RLC}\left('\mathsf{stack}', a, v_{\mathsf{push}}, \mathsf{clk}\right)\right\}\,.
\end{aligned}$$

- For a stack POP operation that reads a value v_{pop} that was pushed at time $\mathsf{clk}_{\mathsf{load}}$ from stack address a at time clk,

$$\begin{aligned}
\mathsf{record}_{\mathsf{range}} &= \mathsf{record}_{\mathsf{range}} \cup \{(0 \le a < \mathsf{size}_{\mathsf{stack}})\}\,, \\
\mathsf{record}_{\mathsf{range}} &= \mathsf{record}_{\mathsf{range}} \cup \{(\mathsf{clk}_{\mathsf{pop}} < \mathsf{clk})\} \\
\mathsf{record}^{\mathsf{OAM}}_{\mathsf{load}} &= \mathsf{record}^{\mathsf{OAM}}_{\mathsf{load}} \cup \left\{\mathsf{RLC}\left('\mathsf{stack}', a, v_{\mathsf{pop}}, \mathsf{clk}_{\mathsf{pop}}\right)\right\}\,.
\end{aligned}$$

- For a stack SWAP operation that swaps v, v' in addresses a, a' that was pushed at time $\mathsf{clk}_{\mathsf{pop}}, \mathsf{clk}'_{\mathsf{pop}}$, respectively, currently at time clk, it is equivalent to execute two stack pop operations for $(a, v, \mathsf{clk}_{\mathsf{pop}})$ and $(a', v', \mathsf{clk}'_{\mathsf{pop}})$, followed by two stack push operations for (a, v', clk) and (a', v, clk).
- For a stack DUP operation that duplicates the value v at stack address a that was pushed at timestamp $\mathsf{clk}_{\mathsf{pop}}$, currently at time clk with the stack top address a_{top}. it is equivalent to execute one stack pop operations for $(a, v, \mathsf{clk}_{\mathsf{pop}})$, followed by two stack push operations for (a, v, clk) and $(a_{\mathsf{top}}, v, \mathsf{clk})$.

In summary, we unify all the lookup arguments into a single ROM table, and the stack and memory operations into a RAM table. From the circuit point of view, the chip circuit now only consists of one LogUp circuit and one set equality check, thus reducing the number of rounds in the GKR protocol for the chip circuit.

8 Conclusions And Discussion

We conclude our paper with the following remarks.

Application to register machines Throughout the paper, we demonstrated how to build a zkVM for a stack machine. Our framework can also be applied to register machines, although extra care need to be taken when dealing with registers. We leave this to future work.

Alternative provers We used GKR as our backend prover for its data-parallel circuit friendliness. Nonetheless, our framework is generic for various provers, so long as they can prove the opcode circuit, basic block circuit and chip circuit efficiently. To this end, we may plug-and-play other candidates, such as [32], which, although have a slower asymptotic complexity, has shown great performance in practice.

Opportunistic proving We note that the GKR-zkVM and GKR-zkVM Pro schemes in this paper are not mutually exclusive. In certain scenarios, it may be beneficial to segment a program into multiple pieces, with parts proved via GKR-zkVM and others proved via GKR-zkVM Pro. We may opportunistically apply GKR-zkVM Pro to the basic blocks that are repeated for many times; and leave the rest to GKR-zkVM. How to dynamically segment the program is still an open problem, while finding a sweet spot requires further analysis and real world experiments. We also leave this to future work.

References

1. 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.
2. arkworks contributors. `arkworks` zksnark ecosystem, 2022.
3. Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. Cryptology ePrint Archive, Paper 2016/116, 2016. `https://eprint.iacr.org/2016/116`.
4. 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.
5. 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`.
6. 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`.
7. 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.
8. 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.

9. Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. EOS: Efficient private delegation of zkSNARK provers. In *USENIX Security Symposium*, pages 6453–6469, 2023.
10. 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.
11. 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.
12. Azetc foundation. Aztec. `https://aztec.network/`.
13. 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.
14. 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.
15. 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`.
16. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. pages 113–122, 2008.
17. 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.
18. Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. `https://eprint.iacr.org/2016/260`.
19. 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`.
20. 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.
21. Modulus Lab. Scaling intelligence: Verifiable decision forest interence with remainder. Github, 2022. `https://github.com/Modulus-Labs/Papers/blob/master/remainder-paper.pdf`.
22. Succinct lab. Succinct processor 1. `https://blog.succinct.xyz/introducing-sp1/`.
23. 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.
24. 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.
25. C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. pages 2–10 vol.1, 1990.

26. Nexus Inc. Nexus zkVM. `https://github.com/nexus-xyz/nexus-zkvm/`.
27. Ethereum Org. Go ethereum: Official go implementation of the ethereum protocol. `https://github.com/ethereum/go-ethereum`.
28. Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using gkr. Cryptology ePrint Archive, Paper 2023/1284, 2023.
29. Paradigm. Reth: Modular, contributor-friendly and blazing-fast implementation of the ethereum protocol, in rust. `https://github.com/paradigmxyz/reth`.
30. Aztec project. noir language, 2023. `https://noir-lang.org/docs/`.
31. Monero Project. Monero. Online, 2022. `https://github.com/monero-project/monero`.
32. Polygon project. Plonky2. `https://github.com/mir-protocol/plonky2`.
33. Polygon project. Plonky3. `https://github.com/Plonky3/Plonky3`.
34. Polygon project. Polygon Hermez. `https://polygon.technology/solutions/polygon-hermez/`.
35. Polygon project. Polygon Miden. `https://polygon.technology/polygon-miden`.
36. Zcash project. The halo2 book.
37. Zcash project. PLONKish arithmetization. link, 2022.
38. ZkSync project. ZkSync. `https://zksync.io/`.
39. Risc-Zero project. Risc-Zero. `https://www.risczero.com/`.
40. SCIPR Lab. `libiop`: a C++ library for IOP-based zkSNARKs. `https://github.com/scipr-lab/libiop`, 2021.
41. Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Advances in Cryptology – CRYPTO 2020*, pages 704–737, 2020.
42. Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023.
43. Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023.
44. Starkware. Starknet. `https://www.starknet.io/en`.
45. StarkWare Team. `ethSTARK`. `https://github.com/starkware-libs/ethSTARK`, 2021.
46. Scroll tech. Scroll. `https://scroll.io/`.
47. Justin Thaler. Time-Optimal Interactive Proofs for Circuit Evaluation. In *Advances in Cryptology – CRYPTO 2013*, pages 71–89, 2013.
48. Justin Thaler. Proofs, arguments, and zero-knowledge, 2020.
49. Justin Thaler. Proofs, Arguments, and Zero-Knowledge. December 2022.
50. Valida. Valida, a stark-based virtual machine, 2023. `https://github.com/valida-xyz/valida`.
51. Wikipedia contributors. Basic block — Wikipedia, the free encyclopedia, 2023.
52. 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.
53. 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.
54. 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`.

55. 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.
56. 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.
57. 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.
58. 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 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 $0\le k<c'$.*

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

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

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 $f^{(0)}_{1,\mathsf{eval}},\dots,f^{(c+c'-1)}_{1,\mathsf{eval}}$ to the verifier.*
4. *Set $\sigma=F_{1,\mathsf{eval}}$, and $F_2(\mathbf{T})=f_2(\mathbf{T})g_2(\mathbf{T})$, where*

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

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.*

– $\mathsf{GKR}^{\mathsf{Phase1}}_{\mathsf{layered}}.\mathsf{Verify}^{(L_i)}_{n}\left(\left\{(\widetilde{V}_i(\mathbf{y}_k\|\mathbf{t}_k), (\mathbf{y}_k\|\mathbf{t}_k))\right\}, \left\{(\widetilde{V}_{i\to\ell_{k-c}}(\mathbf{y}_k\|\mathbf{t}_k), (\mathbf{y}_k\|\mathbf{t}_k))\right\}\right)$*:*
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 $f^{(0)}_{1,\mathsf{eval}}, \ldots, f^{(c+c'-1)}_{1,\mathsf{eval}}$ from the prover.*
4. *Compute $g^{(k)}_{1,\mathsf{eval}}(\mathbf{y})$ as follows:*
$$g^{(k)}_{1,\mathsf{eval}} = \alpha^k \widetilde{eq}(\mathbf{y}_j, \mathbf{y}),\ \textit{for } 0 \le k < c\ ,$$
$$g^{(k)}_{1,\mathsf{eval}} = \alpha^k \widetilde{\mathsf{copy\_to}}_{i\to\ell_{k-c}}(\mathbf{y}_j, \mathbf{y}),\ \textit{for } c \le k < c + c',$$
5. *Check whether $F_{1,\mathsf{eval}} = \sum_{k=0}^{c+c'} f^{(k)}_{1,\mathsf{eval}} g^{(k)}_{1,\mathsf{eval}}$.*
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*
$$g^{(k)}_{2,\mathsf{eval}} = \alpha^k \widetilde{eq}(\mathbf{y}_j, \mathbf{y})\widetilde{eq}(\mathbf{t}_k, \mathbf{t})\ \textit{for } 0 \le k < c\ ,$$
$$g^{(k)}_{2,\mathsf{eval}} = \alpha^k \widetilde{\mathsf{copy\_to}}_{i\to\ell_{k-c}}(\mathbf{y}_j, \mathbf{y})\widetilde{eq}(\mathbf{t}_k, \mathbf{t}),\ \textit{for } c \le k < c + c',$$
10. *Check whether $F_{2,\mathsf{eval}} = \sum_{k=0}^{c+c'} g^{(k)}_{2,\mathsf{eval}} \cdot f_{2,\mathsf{eval}}$.*

Protocol 7 (GKR Layer Prover Protocol, Phase 2, Based on Equation 8)
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.

– $\mathsf{GKR}^{\mathsf{Phase2}}_{\mathsf{layered}}.\mathsf{Prove}^{(L_i)}_{n'}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t})), \widetilde{V}_{i+1}(\mathbf{X}, \mathbf{S}), \left\{\widetilde{V}_{\ell'_j \to i}(\mathbf{X}, \mathbf{S})\right\}\right)$*: 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*

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

  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 8)
This is the verifier protocol corresponding to Protocol 7.

– $\mathsf{GKR}^{\mathsf{Phase2}}_{\mathsf{layered}}.\mathsf{Verify}^{(L_i)}_{n'}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t}))\right)$: *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*
$$g_{0,j}(\mathbf{x}^{(0)}\|\mathbf{s}^{(0)}) = \tilde{eq}(\mathbf{t}, \mathbf{s}^{(0)})\widetilde{\mathsf{paste\_from}}_{\ell'_j \to i}(\mathbf{y}, \mathbf{x}^{(0)}) \text{ , for } 0 \le j < c''$$
  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* $\sigma = g_{w,\mathsf{eval}}$.

Protocol 9 (GKR Layer Prover Protocol, Phase 2, Based on Equation 9)
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.

– $\mathsf{Prove}_{n'}^{(L_i)}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t})), \widetilde{V}_{i+1}(\mathbf{X}, \mathbf{S}), \left\{\widetilde{V}_{\ell'_j \to i}(\mathbf{X}, \mathbf{S})\right\}\right)$*: 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*

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

  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*

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

  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*

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

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

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

– $\mathsf{Verify}_{n'}^{(L_i)}\left((\widetilde{V}_i(\mathbf{y}\|\mathbf{t}), (\mathbf{y}\|\mathbf{t}))\right)$: *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* $0 \le j < c''$.
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*
$$g^{(j)}_{0,\mathsf{eval}} = \widetilde{\mathsf{paste\_from}}_{\ell'_j \to i}(\mathbf{z}, \mathbf{x}^{(0)}).$$
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* $\sigma = g_{w,\mathsf{eval}} \cdot (G^{(w)}(\mathbf{z}\|\mathbf{x}^{(w)}))^{-1}$.