Tight ZK CPU[*]

Batched ZK Branching with Cost Proportional to Evaluated Instruction

Yibin Yang
Georgia Institute of Technology,
Atlanta, USA
yyang811@gatech.edu

David Heath
University of Illinois
Urbana-Champaign, Urbana, USA
daheath@illinois.edu

Carmit Hazay
Bar-Ilan University, Ramat Gan, Israel
Ligero Inc., Rochester, USA
Carmit.Hazay@biu.ac.il

Vladimir Kolesnikov
Georgia Institute of Technology,
Atlanta, USA
kolesnikov@gatech.edu

Muthuramakrishnan
Venkitasubramaniam
Ligero Inc., Rochester, USA
muthu@ligero-inc.com

Abstract

We explore Zero-Knowledge Proofs (ZKPs) of statements expressed as programs written in high-level languages, e.g., C or assembly. At the core of executing such programs in ZK is the repeated evaluation of a CPU step, achieved by branching over the CPU's instruction set. This approach is general and covers traversal-execution of a program's control flow graph (CFG): here CPU instructions are straight-line program fragments (of various sizes) associated with the CFG nodes. This highlights the usefulness of ZK CPUs with a *large* number of instructions of *varying sizes.*

We formalize and design an efficient *tight* ZK CPU, where the cost (both computation and communication, for each party) of each step depends only on the instruction taken. This qualitatively improves over state of the art, where cost scales with the size of the *largest* CPU instruction (largest CFG node).

Our technique is formalized in the standard commit-and-prove paradigm, so our results are compatible with a variety of (interactive and non-interactive) general-purpose ZK.

We implemented an interactive tight arithmetic (over $\mathbb{F}_{2^{61}-1}$) ZK CPU based on *Vector Oblivious Linear Evaluation* (VOLE) and compared it to the state-of-the-art non-tight VOLE-based ZK CPU Batchman (Yang et al. CCS'23). In our experiments, under the same hardware configuration, we achieve comparable performance when instructions are of the same size and a 5-18× improvement when instructions are of varied size. Our VOLE-based tight ZK CPU (over $\mathbb{F}_{2^{61}-1}$) can execute 100K (resp. 450K) multiplication gates per second in a WAN-like (resp. LAN-like) setting. It requires $\leq$ 102 Bytes per multiplication gate. Our basic building block, ZK *Unbalanced Read-Only Memory*, may be of independent interest.

CCS Concepts

• **Security and privacy** → **Cryptography**; • **Theory of computation** → **Cryptographic protocols**.

[*]The full version [44] is available online: https://eprint.iacr.org/2024/456.

Keywords

Zero-Knowledge; Disjunctive Statements; CPU Emulation

ACM Reference Format:
Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. 2024. Tight ZK CPU: Batched ZK Branching with Cost Proportional to Evaluated Instruction. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690289

1 Introduction

Zero-Knowledge (ZK) Proofs (ZKPs) [24] allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a given statement is true without revealing anything beyond this fact. With recent advances in efficiency, ZKP has become one of the most active areas in cryptographic research. Example applications include private blockchain [3], private programming analysis [18, 35], private bug-bounty [26, 45], privacy-preserving machine learning [34, 40], and many more.

Most generic ZK schemes prove statements represented as circuits or constraint systems. While these formats support arbitrary statements, they do not align with how computational tasks are often described or developed in practice – using a high-level language, such as C/C++/assembly/etc.

A promising path towards efficient ZKP for general programs is to mimic what plaintext computers do. An assembly (or C/C++ or other high-level) program can be broken into straight-line blocks; the resulting program *control-flow graph* (CFG) describes how program control can transfer between the blocks.

Casting this to ZKP (and for efficiency, omitting the plaintext-world step of compiling to a hardware CPU fixed instruction set), instead of agreeing on a single public circuit, $\mathcal{P}$ and $\mathcal{V}$ agree on B circuits, each corresponding to (i.e., implementing a straight-line program of) a CFG block. Viewed this way, the objective of ZKP is to execute the program from a public initial state to a public final state via a circuit constructed by *privately* "soldering" these (potentially repeated) basic CFG blocks (see Figure 1). This approach can be viewed as executing steps of a *Zero-Knowledge Central Processing Unit* (ZK CPU) whose instruction set is defined in terms of the target program's complex CFG blocks. An MPC version of this approach is explored by recent VISA MPC [46].

Of course, a ZK CPU must be able to access a *random-access memory* (RAM); this technical task is external to our focus. We show

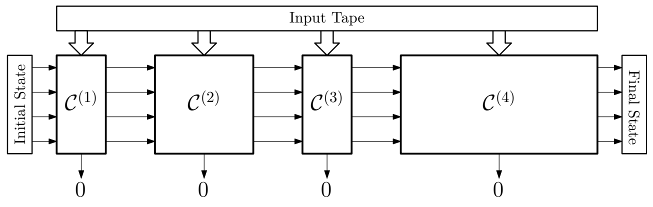


Figure 1: Example ZK CPU execution. $\mathcal{P}$ and $\mathcal{V}$ agree on B public (sub)circuits $I = \{C_1, \ldots, C_B\}$. $\mathcal{P}$ demonstrates to $\mathcal{V}$ that an initial state evaluates to a final state via a *private* circuit $C \triangleq C^{(4)} \circ \cdots \circ C^{(1)}$, where each $C^{(i \in [4])} \in I$. $\mathcal{V}$ learns the size of C but does *not* learn the number or identity of specific subcircuits used. Each subcircuit's output is fed as input to the subsequent subcircuit. We refer to the wires that pass from subcircuit to subcircuit as *registers*. Each subcircuit can read private input from $\mathcal{P}$, and each subcircuit outputs a "checking output", which evaluates to 0 when $\mathcal{P}$ is honest. The checking output can be used to, e.g., force $\mathcal{P}$ to use C_1 when the first register is 1. See Section 3 for formal details.

that the state-of-the-art ZK RAM [42] can be efficiently integrated with our ZK CPU (see Section 6.2).

ZK disjunctions. The sequence of executed CFG blocks (instructions) must remain hidden from $\mathcal{V}$. This can be trivially achieved by $\mathcal{P}$ and $\mathcal{V}$ executing *each* instruction in each step – the circuit for computing such a step would be a disjunction of all instructions (in the instruction set), and the top-level proof statement would simply be a sufficient number of repetitions of the disjunction.

This approach incurs a glaring overhead: parties execute – and pay for – a large number of inactive (i.e., not taken in plaintext execution) clauses in each disjunction. To make matters worse, many programs have large CFGs, so each disjunction is over a large number of clauses, causing corresponding overhead.

A recent line of work ([2, 21–23, 25, 29, 43]) aims to avoid paying for inactive clauses in a disjunction. [25] described the possibility of reusing the cryptographic *material* of the active branch to evaluate (to garbage and privately discard) inactive branches. This limits communication to the cost of a single (longest) branch but still requires processing all branches. Very recent work [22, 43] shows how to limit both communication and computation to that of the single longest branch for our setting, where the same disjunction (of all instructions in the instruction set) is executed repeatedly.

To summarize, the state of the art *pays for the longest branch.*

1.1 Our Focus: Pay for the Active Branch

We are motivated by scenarios where instructions (or branches) differ significantly in size, possibly by orders of magnitude. In such cases, it is unacceptable to incur the cost of the longest branch. While instructions in hardware CPUs are roughly the same size by design, this is *not the case* in CFGs, where blocks correspond to straight-line program segments.

Tight ZK CPU emulation. We mostly adhere to the ZK CPU notation and vocabulary. We choose this over other equivalent vocabularies, such as CFG and blocks, discussed above. This is for simplicity, clarity, and consistency, since prior ZK work already uses the CPU and CPU-emulation terminology and definitions (e.g., [4, 20, 26, 43]).

Extending the existing ZK CPU vocabulary, in this work, we introduce and focus on *tight ZK CPU emulation* (or just tight ZK CPU) – one whose cost of executing each instruction is proportional to the size of *that* instruction. This is in contrast to all prior work on efficient ZK CPU emulation, where the cost of executing a CPU step is proportional to the *total cost of all* instructions in the CPU or, more recently, to the largest instruction in the CPU.

It is challenging to achieve tight ZK CPU concretely efficiently because instruction boundaries must be hidden from $\mathcal{V}$, and corresponding *expensive* instruction set-up and conclusions (which, e.g., handle registers, instruction loads, proof checks, etc.) must be executed at each possible basic step of the ZK proof.

Spliting large instructions. It is, of course, possible to equalize instruction sizes by splitting a large instruction $\mathscr{C}$ into a sequence of small instructions. This incurs the expense of passing *more* registers between instructions more frequently: the current internal state of the larger instruction $\mathscr{C}$ now must be passed between its consecutive sub-instructions $\mathscr{C}_i$ and $\mathscr{C}_{i+1}$. This internal state corresponds to the width of the circuit implementing $\mathscr{C}$ and may be large. Crucially, now *all* instructions must accept this many registers as input to preserve ZK, incurring corresponding overhead.

Our work allows cheaply handling arbitrarily large (and arbitrarily wide!) instructions without incurring the overhead of handling additional registers.

Privacy guarantees. The privacy guarantees provided by prior CPU-emulation definitions and constructions are somewhat different from that of our tight ZK CPU. In prior work, $\mathcal{V}$ learns the number of executed CPU steps; in our work, $\mathcal{V}$ learns the total number of multiplication gates on the program execution path. Both metrics correspond to (slightly different) notions of program runtime. We stress that revealing the runtime is inevitable when demanding tight prover efficiency, and standard padding techniques can provide finer privacy guarantees.

Depending on instruction sizes, the total number of evaluated gates in executing our tight CPU can indicate to $\mathcal{V}$ with high confidence which instructions were executed. A similar concern applies to prior ZK CPU work, where a precise runtime (number of instructions) might tell $\mathcal{V}$ the execution path. Such issues are arguably more relevant in our model since runtime is more granular. As in prior work, this can be addressed by runtime padding via inserting dummy multiplications.

1.2 Our Contribution

We motivate and formalize the notion of a *tight* ZK CPU, where the cost (both computation and communication for each party) of each step depends only on the instruction taken, even when the instructions are of varying sizes. We define an ideal functionality $\mathcal{F}_{\mathsf{ZKCPU}}$ (see Figure 5 and discussion in Section 3) to capture this notion by only sending the length of the entire execution to $\mathcal{V}$.

Our protocol realizes $\mathcal{F}_{\mathsf{ZKCPU}}$ in the commit-and-prove hybrid (defined as $\mathcal{F}_{\mathsf{CPZK}}$ in Figure 2) model with *information-theoretic* *security*. Our protocol is *public-coin* and *constant-round* in $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model, so it natively supports the Fiat-Shamir transformation [19, 38]. Crucially, our abstraction allows realizing the $\mathcal{F}_{\mathsf{ZKCPU}}$ via a variety of commit-and-proof ZK protocols, including interactive and non-interactive ones (e.g., [1, 2, 8, 11, 14, 17, 28, 36, 41]).

We implement[1] a tight ZK CPU protocol by instantiating the commit-and-proof ZK with VOLE-based ZK [17, 41] and report the performance in Section 7. The cost of our VOLE-based tight ZK CPU scales only *linearly* with the number of multiplication gates along the program execution path. Concretely, this protocol outperforms the state of the art Batchman [43] (a VOLE-based non-tight ZK CPU) in *both* computation and communication commensurately with branch size variation (see Section 7). Our VOLE-based ZK CPU achieves a cost of only a constant factor (6–7×) higher than the *non-private* protocol, where the execution path is revealed to $\mathcal{V}$.

1.3 Intuition of Our Construction

We present high-level intuition here; Section 4 presents a detailed technical overview of our approach.

Consider a ZK proof expressed as a high-level program composed of basic "control-flow" blocks, which we call *instructions*. $\mathcal{P}$'s witness is an input to the program that evaluates to an accepting state. The proof convinces $\mathcal{V}$ the existence of a sequence of instructions – an execution path – leading to an accepting state. While the execution path, known to $\mathcal{P}$, can depend on $\mathcal{P}$'s secret witness, a ZK proof must hide the path from $\mathcal{V}$.

The recent Batchman protocol [43] demonstrates that it is possible to efficiently encode each program instruction as a randomized vector of field elements. At a high level, each such vector is the product of $\mathcal{V}$'s random challenge vector and a matrix that encodes the linear constraints imposed by the instruction; see Section 2.5. Thus, an execution path can be encoded as a vector constructed by concatenating subvectors corresponding to each instruction. Batchman uses this encoding to hide the identity of each instruction from $\mathcal{V}$. In particular, this vector encoding the execution path is included in the proof as part of $\mathcal{P}$'s (extended) witness.

If $\mathcal{P}$ is honest, this vector encodes a valid execution path. $\mathcal{P}$ proves her witness satisfies linear constraints imposed by the vector.

Of course, $\mathcal{V}$ must check in ZK that $\mathcal{P}$'s execution path vector is valid – that each subvector (or, rather, each subvector's hash) is in the set of valid instructions (hashes) of the source program. Batchman's ZK hash check is efficient: each subvector hash is a random linear combination of the subvector's elements based on a fresh challenge from $\mathcal{V}$– a single uniform field element sent by $\mathcal{V}$, expanded by taking its powers. A crucial detail here is that $\mathcal{V}$ knows the boundaries of the subvectors, as Batchman's instructions are each padded to the same publicly agreed-upon number of gates.

In our approach, we allow instructions of different sizes. Thus, while our prover also inputs an execution path vector, the subvector (i.e., instruction) boundaries and the lengths of each subvector must be kept private. With this change, the subvector validity check and passing of program state between instructions become a challenge, the resolution of which is core to our contribution. Here, we give high-level intuition underlying our validity check.

[1]Our implementation is available at https://github.com/gconeice/tight-vole-zk-cpu.

To validate the execution path vector, $\mathcal{P}$ inputs an additional 0-1 vector of the same length, which defines the boundaries of the instruction subvectors. Namely, $\mathcal{P}$ sets this *boundary string* to 0 and places 1 *only* at positions corresponding to the ends of subvectors. Similar to Batchman, our hash check is performed via a random linear combination with a $\mathcal{V}$-chosen challenge, but we carefully arrange how parties use the boundary string to construct and verify hash checksums of unknown length to $\mathcal{V}$. We capture this with a novel primitive of independent interest – an *unbalanced ZK read-only memory (ROM)* – a ZK ROM capable of storing vectors of different lengths, but where we do not pay the price of the largest vector for each memory element (by exploiting the boundary string). Based on the above intuition, our unbalanced ZK ROM manages (loads, concatenates and checks) vectors of different lengths.

1.4 Related Work

Efficient handling of disjunctive statements is central to the handling of ZK proofs expressed as high-level programs. High-level-program-based ZK is an intuitive direction that was first concretely explored by [4] and subsequently studied by [5, 6, 20, 22, 26, 45].

Early ZK work [13] gave special-purpose techniques allowing proofs of disjunctions. With relatively recent and dramatic improvement to proofs of general-purpose statements, special-purpose disjunction handling was (temporarily) subsumed by general-purpose techniques. Indeed, disjunctions are easily encoded and proved as part of a circuit that processes each branch and then multiplexes the results. While this works, it is expensive. [25] – building on the MPC result of [29] – demonstrated feasibility of paying (in ZK proof size) for only one branch. The [25] technique "reuses cryptographic material" of the active branch to evaluate (to garbage and privately discard) inactive branches. This sparked a rich line of work [2, 21–23, 25, 29, 43] that continues to reduce the costs of ZK disjunctions.

Very recent work [22, 43] further improved the handling of disjunctions by showing how to improve not just communication but also *computation*. This task is more challenging and cannot be achieved by prior techniques relying on garbage evaluation of inactive clauses. Leveraging the *batched* setting where a single disjunction is executed repeatedly, these works show how $\mathcal{P}$ and $\mathcal{V}$ compute (and hence communicate) proportionally only to the single largest clause of the disjunction. Our work extends and crucially builds on the approach of [43], and our extension enables paying only for the *active* branch. Sections 1.3 and 2.5 summarize [43] and the novel techniques needed for our result. Neither [43] nor [22] address disjunctions of clauses of varying sizes.

Efficient ZK ROM and RAM are essential to CPU-emulation ZK. We integrate recent ZK ROM [42]. We also build on it to design a novel basic primitive *unbalanced ZK ROM*, capable of retrieving variable-size entries in a batch query. We achieve this by extending randomized hashes of [43] to vectors of differing lengths and ultimately use them to execute variable-size instructions.

We note that emulating CPU in SNARK has also been intensively studied recently in, e.g., [12, 16, 27, 30–32, 47]. Some of these elegant works (e.g., [27, 30, 32, 47]) can indeed achieve tight efficiency while offering attractive features such as non-interactivity and succinctness. However, adding ZK to these works may either require large overheads or break tightness (see, e.g., discussions in [22] and [33]). Furthermore, they (at least) reveal the number of instructions to the verifier, while our work reveals only the total number of multiplication gates. See Section 3 for more formal discussions. We suspect that some padding techniques might address the additional leakage in, e.g., [30, 32], and we leave it as valuable future work. Finally, we remark that our protocols can also be instantiated with a succinct and non-interactive commit-and-prove zkSNARK.

2 Preliminaries

2.1 Notation

- λ is the statistical security parameter (e.g., 40 or 60).
- The prover is $\mathcal{P}$. We refer to $\mathcal{P}$ by she, her, hers...
- The verifier is $\mathcal{V}$. We refer to $\mathcal{V}$ by he, him, his...
- $x \triangleq y$ denotes that x is *defined* as y.
- We denote sets by upper-case letters. We denote that x is uniformly drawn from a set S by $x \in_\$ S$.
- We denote $\{1, \ldots, n\}$ by $[n]$.
- We denote a finite field of size p by $\mathbb{F}_p$ where $p \geq 2$ is a prime or a power of a prime. We use $\mathbb{F}$ to represent a sufficiently large field, i.e., $|\mathbb{F}| = \lambda^{\omega(1)}$. $\mathsf{Inverse}(x)$ denotes the multiplicative inverse of $x (\neq 0)$ in $\mathbb{F}$, i.e., $\mathsf{Inverse}(x) \cdot x = 1$.
- For a vector $\boldsymbol{a} \in \mathbb{F}^n$ and an element $x \in \mathbb{F}$, $x\boldsymbol{a} \triangleq (xa_1, \ldots, xa_n)$.
- $\mathsf{last}(\boldsymbol{a})$ denotes the last element of $\boldsymbol{a}$, i.e., a_n if $\boldsymbol{a} \in \mathbb{F}^n$. For some $\boldsymbol{a} \in \mathbb{F}^*$, if $\mathsf{last}(\boldsymbol{a}) \neq 0$, we refer to $\boldsymbol{a}$ as a *non-zero-end* vector.
- We denote row vectors by bold lower-case letters (e.g., $\boldsymbol{a}$), where a_i (or $\boldsymbol{a}[i]$) denotes the i-th component of $\boldsymbol{a}$ (starting from 1) and $\boldsymbol{a}[:i]$ the subvector $(a_1, \ldots, a_i)$.
- We denote matrices by bold upper-case letters (e.g., $\boldsymbol{A}$), where $\boldsymbol{A}(i)$ denotes the i-th row vector of $\boldsymbol{A}$ (starting from 1) and $\boldsymbol{A}[i]$ denotes the i-th column vector of $\boldsymbol{A}$ (starting from 1). $\boldsymbol{A}(i)[j]$ denotes j-th value in i-th row.
- Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be vectors of equal length. $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ denotes the inner product; $\boldsymbol{a} \odot \boldsymbol{b}$ denotes the element-wise product.
- We denote a multiplication (gate) by MULT.

2.2 Security Model

We formalize our protocol via the universally composable (UC) framework [9] and prove its security in the presence of a *malicious, static* adversary. For simplicity, we omit standard UC session (and sub-session) IDs.

2.3 Commit-and-Prove Zero-Knowledge

Our protocol is defined in the *commit-and-prove* hybrid model [10]. This functionality, denoted by $\mathcal{F}_{\mathsf{CPZK}}$ and formally defined in Figure 2, allows $\mathcal{P}$ to commit to field elements (over $\mathbb{F}$) and then prove that evaluating a particular circuit on the committed values yields a vector of 0's. We denote by $\mathsf{com}(\alpha)$ a cryptographic commitment to $\alpha \in \mathbb{F}$, and naturally extend this notation to vectors (e.g., $\mathsf{com}(\boldsymbol{\alpha})$).

There are several ways to instantiate $\mathcal{F}_{\mathsf{CPZK}}$ (e.g., [1, 2, 8, 11, 14, 17, 28, 36, 41]). To concretely evaluate our abstraction, we choose to instantiate our protocol via the VOLE-based ZK (e.g., [2, 17, 41], cf. Lemma 1), a proof paradigm known for its fast end-to-end running times and small (constant) computation/communication rates compared to $|C|$. This paradigm employs information-theoretic MACs (IT-MACs) [7, 37] as linearly homomorphic commitment schemes over $\mathbb{F}$. We describe the computation/communication of VOLE-based ZK (via a formal version of Lemma 1) in [44].

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

$\mathcal{F}_{\mathsf{CPZK}}$, parameterized by a field $\mathbb{F}$, proceeds as follows, running with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and an adversary $\mathcal{S}$:

Commitments. On receiving $(\mathtt{Commit}, cid, x)$ from $\mathcal{P}$ where (a) there is no recorded tuple $(cid, \cdot)$, and (b) $x \in \mathbb{F}$: Record tuple (cid, x) and send $(\mathtt{commit}, cid)$ to $\mathcal{V}$ and $\mathcal{S}$.

Linear Combination. On receiving $(\mathtt{Linear}, cid, cid_1, \ldots, cid_k, c_0, c_1, \ldots, c_k)$ from $\mathcal{P}$ where (a) there is no recorded tuple $(cid, \cdot)$, (b) each $cid_{i \in [k]}$ has a recorded tuple, and (c) $c_0, \ldots, c_k \in \mathbb{F}$:

1. Fetch recorded $(cid_1, x_1), \ldots, (cid_k, x_k)$.
2. Compute $x := c_0 + c_1 x_1 + \cdots + c_k x_k$. Record (cid, x).
3. Send $(\mathtt{linear}, cid, cid_1, \ldots, cid_k, c_0, \ldots, c_k)$ to $\mathcal{V}, \mathcal{S}$.

Open. On receiving $(\mathtt{Open}, cid)$ from $\mathcal{P}$ where cid has a recorded tuple, fetch (cid, x), send $(\mathtt{open}, cid, x)$ to $\mathcal{V}$ and $\mathcal{S}$.

Check. On receiving $(\mathtt{Check}, C, cid_1, \ldots, cid_n)$ from $\mathcal{P}$ where (a) $C : \mathbb{F}^{(n)} \to \mathbb{F}^{(*)}$ is an arithmetic circuit, and (b) each $cid_{i \in [n]}$ has a recorded tuple: Fetch tuples $(cid_1, x_1), \ldots, (cid_n, x_n)$ and compute $\boldsymbol{y} := C(x_1, \ldots, x_n)$. If $\boldsymbol{y} = 0^{(*)}$, send $(\mathtt{check}, C, \boldsymbol{cid}, \mathtt{true})$ to $\mathcal{V}$ and $\mathcal{S}$; else send $(\mathtt{check}, C, \boldsymbol{cid}, \mathtt{false})$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 2: Ideal functionality for commit-and-prove ZK. Each committed element is associated with a unique identifier *cid*. $\mathtt{Linear}$ operation allows $\mathcal{P}$ to generate a new commitment (associated with *cid*) via a *public affine function* over committed elements.

Lemma 1 (VOLE-based ZK, Informal). *There exists a protocol Π_{CPZK} that UC-realizes $\mathcal{F}_{\mathsf{CPZK}}$ in the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model (see [44]) with $O(|C|)$ comp./comm. costs per ZKP over a circuit C.*

Testing vector equality. We apply the *Swchartz-Zippel* lemma as a central tool to prove the equality of two (committed) vectors.

Lemma 2 (Vector Equality). *Consider vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^n$. If $\boldsymbol{a} \neq \boldsymbol{b}$, then for $\chi \in_\$ \mathbb{F}$:*

$$\Pr[\langle (1, \chi, \ldots, \chi^{n-1}), \boldsymbol{a} \rangle = \langle (1, \chi, \ldots, \chi^{n-1}), \boldsymbol{b} \rangle] \leq \frac{n}{|\mathbb{F}|}$$

Specifically, suppose the parties hold committed vectors $\mathsf{com}(\boldsymbol{a})$ and $\mathsf{com}(\boldsymbol{b})$, and $\mathcal{P}$ wishes to convince $\mathcal{V}$ that $\boldsymbol{a}$ is equal to $\boldsymbol{b}$. Lemma 2 states that it suffices for $\mathcal{P}$ to prove that $\langle (1, \chi, \ldots \chi^{n-1}), \boldsymbol{a} \rangle = \langle (1, \chi, \ldots, \chi^{n-1}), \boldsymbol{b} \rangle$, where χ is some uniform challenge sampled by $\mathcal{V}$. Note that zero-end vectors of different lengths (e.g., $\boldsymbol{a} = (1, 1)$ and $\boldsymbol{b} = (1, 1, 0)$) are *not* captured by Lemma 2. On the other hand, it does extend to *non-zero-end* vectors of potentially different lengths (Corollary 1). Looking ahead, we need Corollary 1 because $\mathcal{V}$ does not know the boundaries of instructions/subvectors whose equality is proven by $\mathcal{P}$ in the tight ZK CPU.

Corollary 1. *Consider vectors $\boldsymbol{a} \in \mathbb{F}^{n_a}, \boldsymbol{b} \in \mathbb{F}^{n_b}$ where $a[n_a]$, $b[n_b] \neq 0$. If $\boldsymbol{a} \neq \boldsymbol{b}$, for $\chi \in_\$ \mathbb{F}$:*

$$\Pr[\langle (1, \chi, \ldots, \chi^{n_a - 1}), \boldsymbol{a} \rangle = \langle (1, \chi, \ldots, \chi^{n_b - 1}), \boldsymbol{b} \rangle] \leq \frac{n}{|\mathbb{F}|}$$

where $n \triangleq \max\{n_a, n_b\} - 1$.

Functionality $\mathcal{F}_{\text{CPZK-ROM}}$

$\mathcal{F}_{\text{CPZK-ROM}}$, parameterized by a field $\mathbb{F}$, proceeds as follows, running with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$:

$\boxed{\text{CPZK}}$

The functionality supports all instructions of $\mathcal{F}_{\text{CPZK}}$.

$\boxed{\text{Read-Only Memory}}$

Initialize ROM. On receiving $(\texttt{InitROM}, cid_1, \dots, cid_n)$ from $\mathcal{P}$ where each $cid_{i\in[n]}$ was recorded: Fetch $(cid_1, x_1), \dots, (cid_n, x_n)$, create a key-value store X where

$$X[1] := x_1, \cdots, X[n] := x_n$$

and set $f_{\text{rom}} := \texttt{honest}$. Send $(\texttt{initrom}, \boldsymbol{cid})$ to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent calls to InitROM.

Read ROM. On receiving $(\texttt{ReadROM}, cid_1, \dots, cid_m, y_1, \dots, y_m, cid_1^{(id)}, \dots, cid_m^{(id)})$ from $\mathcal{P}$ where

1. InitROM has been executed; and
2. there is no recorded tuple for each $(cid_{i\in[m]}, \cdot)$; and
3. each $y_{i\in[m]} \in \mathbb{F}$; and
4. each $cid_{i\in[m]}^{(id)}$ was recorded.

Fetch $(cid_1^{(id)}, id_1), \dots, (cid_m^{(id)}, id_m)$. Record $(cid_1, y_1), \dots, (cid_m, y_m)$. If $\mathcal{P}$ is honest, $\forall i \in [m]$, $X[id_i] = y_i$ where $id_i \in [n]$. If $\mathcal{P}$ is corrupted, set $f_{\text{rom}} := \texttt{cheating}$ when

1. there exists an $id_{i\in[m]} \notin [n]$, where n is the size of X; or
2. there exists an $i \in [m]$ such that $X[id_i] \neq y_i$.

Send $(\texttt{readrom}, \boldsymbol{cid}, \boldsymbol{cid}^{(id)})$ to $\mathcal{V}$, $\mathcal{S}$. Ignore subsequent ReadROM calls.

Check ROM. On receiving (CheckROM) from $\mathcal{P}$ where InitROM and ReadROM were executed: If $\mathcal{P}$ is corrupted and $\mathcal{S}$ sends Cheat, set $f_{\text{rom}} := \texttt{cheating}$. Send $(\texttt{checkrom}, f_{\text{rom}})$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 3: Ideal functionality for commit-and-prove zero-knowledge allowing proofs that support a read-only memory. $\mathcal{P}$ specifies the result of the ReadROM operation. However, if $\mathcal{P}^*$ provides an incorrect result, the flag f_{rom} will be set to cheating.

2.4 Zero-Knowledge Read-Only Memory

Our protocol uses an extended version of $\mathcal{F}_{\text{CPZK}}$ where parties can access a ZK ROM (e.g., [15, 20, 42]). Namely, ZK ROM allows $\mathcal{P}$ to specify n commitments to initialize a key-value store data structure (K-V store) indexed by the key $k \in [n]$. Subsequently, given com(i), where $i \in [n]$, $\mathcal{P}$ and $\mathcal{V}$ generate a new commitment com(x) where x is the i-th committed value in the K-V store. Our protocol uses a restricted (batch-read) version of ZK ROM formalized in Figure 3. I.e., $\mathcal{P}$ is allowed a *single* ReadROM call, where $\mathcal{P}$ specifies an arbitrarily long vector of ROM indices, possibly with repetitions. This will allow $\mathcal{P}$ to load a sequence of hashes corresponding to the execution path (note, we later introduce a stronger *novel* primitive, unbalanced ROM, to load the concatenation of variable-length instruction vectors, realized in the $\mathcal{F}_{\text{CPZK-ROM}}$-hybrid model). [42] shows the state-of-the-art realization of $\mathcal{F}_{\text{CPZK-ROM}}$ in the $\mathcal{F}_{\text{CPZK}}$-hybrid model (see Lemma 3, the full version includes formal version).

Lemma 3 (ZK ROM, informal). *Let $n = \text{poly}(\lambda), m = \Omega(n)$. There exists a protocol $\Pi_{\text{CPZK-ROM}}$ that UC-emulates $\mathcal{F}_{\text{CPZK-ROM}}$ (Figure 3) in the $\mathcal{F}_{\text{CPZK}}$-hybrid model (Figure 2) with amortized $O(1)$ comp./comm. costs per element read.*

2.5 ZKP via Topology Matrices

Consider a circuit C with n_{in} inputs and $n_\times$ multiplication gates. Note that a ZKP for C can be separated into two parts: (1) multiplication gates and (2) linear constraints. Suppose that $\mathcal{P}$ commits to its input com$(in_1), \dots,$ com$(in_{n_{in}})$, and also commits to the values on the $3n_\times$ wires associated with C's $n_\times$ multiplication gates. I.e., $\mathcal{P}$ commits to com$(\ell_1), \dots,$ com$(\ell_{n_\times})$, corresponding to the multiplication left input wires, to com$(r_1), \dots,$ com$(r_{n_\times})$, corresponding to the right input wires, and to com$(o_1), \dots,$ com$(o_{n_\times})$, corresponding to the output wires. The full vector of $\mathcal{P}$'s input and the multiplication wires (with a constant 1) is called $\mathcal{P}$'s *extended witness*.

Now, $\mathcal{P}$ first proves to $\mathcal{V}$ that $\boldsymbol{\ell} \odot \boldsymbol{r} = \boldsymbol{o}$, demonstrating that its extended witness satisfies multiplicative constraints. Then, it proves that $\boldsymbol{in}, \boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ indeed respect the linear constraints imposed by circuit C. Note that since all multiplication gates were handled in the first step, $\mathcal{P}$ simply needs to show its extended witness respects a particular linear relation – i.e. a matrix $\boldsymbol{M}$. This public matrix $\boldsymbol{M}$ is induced by the structure of the circuit C, and [43] refers to $\boldsymbol{M}$ as a *topology matrix*. Namely, $\mathcal{P}$ proves the following:

$$\boldsymbol{M} \times (1, \boldsymbol{in}, \boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o})^T = \boldsymbol{0} \tag{1}$$

Since $\boldsymbol{in}, \boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ are committed, this equality check can be handled by $\mathcal{V}$'s sending of a uniform challenge $\chi \in_\$ \mathbb{F}$ where $\mathcal{P}$ uses $\mathcal{F}_{\text{CPZK}}$ to construct a commitment to

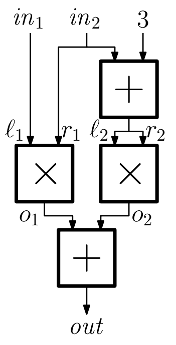
$$\underbrace{(1, \chi, \dots, \chi^{2n_\times}) \times \boldsymbol{M}}_{\text{topology vector}} \times \underbrace{(1, \boldsymbol{in}, \boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o})^T}_{\text{extended witness}} \tag{2}$$

and then proves to $\mathcal{V}$ that this is a commitment to 0. Recall that $\boldsymbol{M}$ is public, so once χ is fixed, both $\mathcal{P}$ and $\mathcal{V}$ know $(1, \dots, \chi^{2n_\times}) \times \boldsymbol{M}$ (called a *topology vector*). Thus, it suffices to check whether the inner product between the topology vector and the extended witness yields 0. Figure 4b shows an example topology matrix.

Proving batched disjunctions: Batchman *[43].* The above paradigm is an overkill if we only perform a ZKP for a single public circuit. This is because it is worse than the state-of-the-art VOLE-based CPZK (e.g. QuickSilver [41]), which only requires committing $\boldsymbol{in}$ and $\boldsymbol{o}$. However, this paradigm becomes useful when considering a batch of disjunctions, as observed by Batchman [43].

In detail, Batchman [43] considers B different circuits $C_1, \dots, C_B$ of the same size. $\mathcal{P}$ wants to repeat the disjunctive proof R times – for each $i \in [R]$, she proves that she knows some witness $\boldsymbol{w}_i$ and some index $id_i \in [B]$ such that $C_{id_i}(\boldsymbol{w}_i) = 0$. To achieve this, for the i-th repetition, $\mathcal{P}$ commits to her extended witness of *only* C_{id_i}. $\mathcal{V}$ then issues a uniform challenge χ to compress B topology matrices to B topology vectors. The *crucial step* is that, for the i-th repetition, $\mathcal{P}$ can commit to the id_i-th topology vector. An extra mechanism is needed to prevent $\mathcal{P}$ from committing to an arbitrary vector that is not a topology vector, which can be built based on a ZK ROM (storing and then loading vectors' hashes). Finally, it suffices to show that the inner product between the extended witness and the topology vector is 0 for each repetition. Batchman can be viewed as a *non-tight* ZK CPU (with extra constraints to support registers).

Note, topology matrices (combined with multiplication constraints) support efficient branching, and thus is a more convenient program representation than, e.g., R1CS [4], for our setting.



(a) The circuit

$$\begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ in_1 \\ in_2 \\ \ell_1 \\ \ell_2 \\ r_1 \\ r_2 \\ o_1 \\ o_2 \end{pmatrix}$$
$$= \left(in_1 - \ell_1, 3 + in_2 - \ell_2, in_2 - r_1, 3 + in_2 - r_2, o_1 + o_2\right)^T$$
$$= \mathbf{0}$$

(b) Original topology matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} in_1 \\ in_2 \\ 1 \\ o_1 \\ o_2 \end{pmatrix}$$
$$= \left(in_1, in_2, 3 + in_2, 3 + in_2, o_1 + o_2\right)^T$$
$$= \left(\ell_1, r_1, \ell_2, r_2, 0\right)^T$$

(c) Refined topology matrix

Figure 4: (a) An arithmetic circuit computing $(in_1 \cdot in_2) + (in_2 + 3)^2$ and its (b) original and (c) refined topology matrix.

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

$\mathcal{F}_{\mathsf{ZKCPU}}$ runs with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$, and is parameterized by a field $\mathbb{F}$, an non-negative integer m, a positive integer B and B m-instructions (Definition 1) $C_1, \ldots, C_B$, an initial state $\boldsymbol{st}^{(0)} \in \mathbb{F}^m$ and a final state $\boldsymbol{st}^{(final)} \in \mathbb{F}^m$. For each $i \in [B]$, let m-instruction C_i have $n_{in}^{(i)}$ inputs and $n_{\times}^{(i)}$ multiplication gates. Note that $n_{in}^{(i \in [B])} \geq m$. W.l.o.g., for each $i \in [B]$, assume $n_{in}^{(i)} - m = n_{\times}^{(i)} + m + 2$ and denote this value as $n^{(i)}$. $\mathcal{F}_{\mathsf{ZKCPU}}$ proceeds as follows:

On receiving $(\mathsf{Prove}, \tau, i_1, \ldots, i_\tau, \boldsymbol{in}_1, \ldots, \boldsymbol{in}_\tau)$ from $\mathcal{P}$ where (1) τ is a positive integer (i.e., the *private* steps), (2) $i_{j \in [\tau]} \in [B]$, and (3) each $\boldsymbol{in}_{j \in [\tau]} \in \mathbb{F}^{n_{in}^{(i_j)} - m}$ (i.e., the inputs except registers), proceed as follows:

(1) Set $\boldsymbol{st} := \boldsymbol{st}^{(0)}$ and $f := \mathsf{true}$. For each $j \in [\tau]$ in order:
   (a) Let $\boldsymbol{st}' \| f' := C_{i_j}(\boldsymbol{st} \| \boldsymbol{in}_j)$ where $\boldsymbol{st}' \in \mathbb{F}^m, f' \in \mathbb{F}$. I.e., $\boldsymbol{st}'$ is the updated registers and f' is the checking output.
   (b) Set $\boldsymbol{st} := \boldsymbol{st}'$. If $f' \neq 0$ (i.e., invalid checking), set $f := \mathsf{false}$.
(2) If $\boldsymbol{st} \neq \boldsymbol{st}^{(final)}$ (i.e., incorrect final state), set $f := \mathsf{false}$.
(3) Let $n \triangleq n^{(i_1)} + \cdots + n^{(i_\tau)}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ can send (Cheat, n') where $n' \in \mathbb{Z}^+$: Set $f := \mathsf{false}$, $n := n'$.
(4) Send (prove, f, n) to $\mathcal{V}$ and $\mathcal{S}$.

Figure 5: Ideal functionality for a tight ZK CPU.

3 Our Target Functionality: $\mathcal{F}_{\mathsf{ZKCPU}}$

We define the functionality of a tight ZK CPU realized by our protocol. To define a ZK CPU over $\mathbb{F}$, we need to specify: (1) $B \in \mathbb{Z}^+$ denotes the number of instructions; (2) $m \in \mathbb{Z}^+$ denotes the number of registers; and (3) each instruction (see Definition 1) is defined as a circuit (over $\mathbb{F}$) mapping $\geq m$ values to $m + 1$ values.

Definition 1 (Instruction). *An instruction is a circuit $C : \mathbb{F}^{n_{in}} \longrightarrow \mathbb{F}^{m+1}$ where $n_{in} \geq m$. In particular, we consider standard fan-in 2 circuits over $\mathbb{F}$ with addition and multiplication gates. We call an instruction $C : \mathbb{F}^{n_{in}} \longrightarrow \mathbb{F}^{m+1}$ a m-instruction, where the first m output wires of C's capture the updated CPU registers, and the last wire is a* checking *output (0 in a valid execution).*

In a tight ZK CPU execution, $\mathcal{P}$ and $\mathcal{V}$ agree on the initial/final state of the m registers (called the initial/final state), where $\mathcal{P}$ demonstrates her ability to execute the initial state to the final state by a sequence of (potentially repeatedly) instructions. We formalize this functionality in Figure 5 with the following remarks:

(1) For each instruction $C^{(i)}$ with $n_{\times}^{(i)}$ multiplications, $n_{in}^{(i)}$ inputs, and m registers, the size of this instruction is defined as $n^{(i)} = n_{in}^{(i)} - m = n_{\times}^{(i)} + m + 2$. Essentially, $n^{(i)}$ reflects the number of the multiplication gates in $C^{(i)}$. We note that our protocol introduces $m + 2$ extra multiplication gates, which are used to constrain m input registers, the constant 1 input, and the checking output. The equality can be enforced by simply padding the instruction with dummy inputs or multiplications. Looking ahead, this equality ensures that the total execution path length hides the executed instructions.
(2) $\mathcal{F}_{\mathsf{ZKCPU}}$ reveals n – the total runtime – to $\mathcal{V}$. Prior non-tight ZK CPUs achieve a similar functionality where $\mathcal{V}$ learns the number of executed instructions τ. We remark that this implies that $\mathcal{V}$ cannot learn τ directly in the tight ZK CPU.
(3) In Figure 5, $\mathcal{P}$ arbitrarily selects which instructions to execute. In some use cases (e.g., when emulating real-world CPUs), $\mathcal{P}$'s chosen instructions should be constrained by the current register state. For example, a program counter register might dictate which instruction runs next. Such constraints can be captured by each instruction's checking output wire, which *must* be 0 in a valid proof (see Sub-step 1b).
(4) $\mathcal{F}_{\mathsf{ZKCPU}}$ only supports limited state (i.e., up to m registers) to be passed between instructions. Perhaps surprisingly, we show that by introducing 5 special registers and 2 extra rounds, our protocol can natively support a large (poly-size in λ) read-write random access memory (see Section 6.2).

4 Technical Overview

In this section, we provide a technical overview of our tight ZK CPU protocol. We refer the reader to Section 1.3 for a high-level intuition. The main steps to achieve our target ideal functionality $\mathcal{F}_{\mathsf{ZKCPU}}$ (Figure 5) are outlined as follows.

$$\mathcal{F}_{\mathsf{CPZK\text{-}ROM}} \overset{\text{Sections 4.4 and 5}}{\Longrightarrow} \mathcal{F}_{\mathsf{CPZK\text{-}UROM}} \overset{\text{Sections 4.3 and 5}}{\Longrightarrow} \mathcal{F}_{\mathsf{ZKCPU}}$$

4.1 Boundary Strings and Helper Notation

Recall our discussion from Section 1.3 regarding a 0-1 vector of field elements used by our protocol, denoted as a *boundary string.*

This section formally defines the boundary strings and introduces useful notations for demonstrating how these strings will be used.

For a vector $\boldsymbol{p} \in \mathbb{F}^n$ where $n \in \mathbb{Z}^+$, we say that $\boldsymbol{p}$ is a *boundary string* if and only if $\boldsymbol{p} \in \{0,1\}^{n-1}\|1$. We note that it is efficient to check whether $\mathsf{com}(\boldsymbol{p})$ commits to a valid boundary string. Namely, given $\mathsf{com}(\boldsymbol{p})$, $\mathcal{P}$ opens p_n to prove it is 1, and $\mathcal{P}$ proves $\boldsymbol{p} \odot (\boldsymbol{1}-\boldsymbol{p}) = \boldsymbol{0}$ (i.e., each $p_{i\in[n]}$ is either 0 or 1).

We use $\mathsf{HW}(\boldsymbol{p})$ to denote the *Hamming weight* of a boundary string. I.e., the number of ones in $\boldsymbol{p}$. We now introduce two functions Partition and Filter that we use as analysis tools. We emphasize that we *never* run these functions inside ZK.

Partition. Consider a length-n boundary string $\boldsymbol{p}$. $\boldsymbol{p}$ specifies a *partition* of a length-n vector $\boldsymbol{v}$ into $\mathsf{HW}(\boldsymbol{p})$ subvectors. We define a function Partition:

$$\boldsymbol{p} = (\overbrace{0,\dots,0,1}^{n_1},\overbrace{0,\dots,0,1}^{n_2},\overbrace{0,\dots,0,1}^{n_3},\dots), \boldsymbol{v} \in \mathbb{F}^n$$
$$\Rightarrow \mathsf{Partition}(\boldsymbol{p},\boldsymbol{v}) = \left(\boldsymbol{v}^{(1)},\dots,\boldsymbol{v}^{(\mathsf{HW}(\boldsymbol{p}))}\right) \text{ such that}$$
$$\boldsymbol{v}^{(1)} = (v_1,\dots,v_{n_1}), \boldsymbol{v}^{(2)} = (v_{n_1+1},\dots,v_{n_1+n_2}),\cdots$$

Filter. A length-n boundary string $\boldsymbol{p}$ also specifies a way to *filter* a length-n vector $\boldsymbol{v}$ into a length-$\mathsf{HW}(\boldsymbol{p})$ vector. We define a function Filter:

$$\boldsymbol{p} = (\overbrace{0,\dots,0,1}^{n_1},\overbrace{0,\dots,0,1}^{n_2},\overbrace{0,\dots,0,1}^{n_3},\dots), \boldsymbol{v} \in \mathbb{F}^n$$
$$\Rightarrow \mathsf{Filter}(\boldsymbol{p},\boldsymbol{v}) = (v_{n_1}, v_{n_1+n_2}, v_{n_1+n_2+n_3},\dots,v_n)$$

Expanding random challenges. In our protocol, $\mathcal{V}$ will issue random challenges, which will be composed with $\mathcal{P}$'s chosen boundary string. We consider two ways to compose these:

(1) For some public challenge $\chi \in \mathbb{F}$, let $s_1 \triangleq 1$, and for each $i \in [n-1]$ in order, let $s_{i+1} := s_i(1-p_i) + \chi^i p_i$. That is,

$$\boldsymbol{p} = (\overbrace{0,\dots,0,1}^{n_1},\overbrace{0,\dots,0,1}^{n_2},\overbrace{0,\dots,0,1}^{n_3},\dots)$$
$$\Rightarrow \boldsymbol{s} = (\underbrace{1,\dots,1}_{n_1},\underbrace{\chi^{n_1},\dots,\chi^{n_1}}_{n_2},\underbrace{\chi^{n_1+n_2},\dots,\chi^{n_1+n_2}}_{n_3},\dots)$$

We denote this procedure by $\boldsymbol{s} \triangleq \mathsf{Expand}_1(\boldsymbol{p},\chi)$.

(2) For some public challenge $\gamma \in \mathbb{F}$, let $s_1 \triangleq 1$, for each $i \in [n-1]$ in order, let $s_{i+1} := \gamma s_i(1-p_i) + p_i$. That is,

$$\boldsymbol{p} = (\overbrace{0,\dots,0,1}^{n_1},\overbrace{0,\dots,0,1}^{n_2},\overbrace{0,\dots,0,1}^{n_3},\dots)$$
$$\Rightarrow \boldsymbol{s} = (1,\gamma,\dots,\gamma^{n_1-1},1,\gamma,\dots,\gamma^{n_2-1},1,\gamma,\dots,\gamma^{n_3-1},\dots)$$

We denote this procedure by $\boldsymbol{s} \triangleq \mathsf{Expand}_2(\boldsymbol{p},\gamma)$.

Starting from $\mathsf{com}(\boldsymbol{p})$, we can compute commitments to the above compositions (i.e., $\mathsf{com}(\boldsymbol{s})$) each via a circuit with $n-1$ MULTs.

4.2 More Powerful Topology Matrices

This section includes how we adjust and optimize the definitions of the topology matrices (discussed in Section 2.5) for our setting.

We first introduce a $\approx 2\times$ optimization to the topology matrix/vector of [43] (see Figure 4c). Note that the order of the multiplication inputs in the [43] topology matrix is fixed (e.g., in Figure 4b, this order is $\ell_1, \ell_2, r_1, r_2, 0$). Based on this observation, there is no need to include the constraints of these fixed order wires internally in the topology matrix (see Figure 4c, refined topology), reducing its size in roughly two and achieving corresponding improvement.

However, neither the topology matrix format of [43] nor the above improvement are suited to our setting because their verifier knows the instruction boundaries, and hence, explicit routing of registers and other wires into instruction entry points is allowed. We must hide this topology from $\mathcal{V}$. To facilitate this, we further rearrange the topology matrices of instructions of our ZK CPU (Figure 5). In particular, constants 0 and 1 and instruction (register or non-register) inputs are not processed in a distinguished manner but rather treated like outputs of regular multiplication gates. (We unify constant wires, input, and multiplication gates into a universal gate.) Formally, we use the following topology matrix equation:

$$\boldsymbol{M} \times (in_1, o_1, \dots, in_n, o_n)^T = (\ell_1, r_1, \dots, \ell_n, r_n)^T \tag{3}$$

Here, n reflects the size of a m-instruction as a circuit C and we define $n = n_{in} - m = n_\times + m + 2$ (see Section 3 especially remark 1). Looking ahead, $\mathcal{P}$ will *privately* order committed $\boldsymbol{\ell} \odot \boldsymbol{r} = \boldsymbol{o}$, starting from $1 \cdot 1 = 1$ (to capture 1 in the extended witness), followed by m registers, then $n_\times$ multiplication tuples in C, and ending with $1 \cdot 0 = 0$ (to capture the checking output).

Notice that in Equation (3), $\mathcal{P}$'s extended witness (or, rather, its topology meta information) is now *compositional* in the sense that if we were to simply concatenate (committed) vectors from two different instructions, we would obtain new vectors of the same form. As we will see next (Section 4.3), a similar form of composition applies to topology matrices (and hence topology vectors), and this enables us to hide from $\mathcal{V}$ the boundaries between instructions.

4.3 Reducing a Tight ZK CPU to a ZK UROM

This section overviews how a tight ZK CPU can be reduced to a so-called ZK UROM functionality. We consider a tight ZK CPU with B instructions $C_1, \dots, C_B$, each of (potentially) different size, where $\mathcal{P}$ wishes to execute C_1 followed by C_2 (i.e., $C_2 \circ C_1$), as an example.

4.3.1 Special Case: No Registers. For simplicity, let us start by considering a special case where our CPU has no registers for passing data between instructions (i.e., $m = 0$). Recall that, w.l.o.g, for each $C_{i\in[B]}$, we assume $n^{(i)} = n_{in}^{(i)} = n_\times^{(i)} + 2$ where C_i has $n_{in}^{(i)}$ inputs, $n_\times^{(i)}$ multiplications.

Suppose $\mathcal{P}$ wishes to first execute C_1, then execute C_2. $\mathcal{V}$ should only learn $n = n^{(1)} + n^{(2)}$, and $\mathcal{V}$ learns neither how many instructions, nor which instructions are executed (unless such information is implied by n). Now, imagine a larger circuit C that expresses the composition $C_2 \circ C_1$. In particular, C can be described by simply concatenating the gate-by-gate description of C_1 and C_2 and appropriately shifting the names (indexes) of C_2's gates and wires by $n^{(1)}$. A key observation is that the topology matrix for C can be constructed by combining the topology matrices for C_1 and C_2:

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{M}^{(1)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{M}^{(2)} \end{pmatrix} \in \mathbb{F}^{2n\times 2n}, n \triangleq n^{(1)} + n^{(2)} \tag{4}$$

where $\boldsymbol{M}^{(1)}$ (resp. $\boldsymbol{M}^{(2)}$) is the topology matrix induced by C_1 (resp. C_2). Our approach hides C (and $\boldsymbol{M}$) from $\mathcal{V}$, even though each $\boldsymbol{M}^{(i\in[B])}$ and n is *public*. For this simple case, our proof would proceed as follows:

(1) $\mathcal{P}$ commits to n inputs $\boldsymbol{in}$ and n MULT tuples $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ in the order described by Equation (3) (and proves $\boldsymbol{\ell} \odot \boldsymbol{r} = \boldsymbol{o}$).
(2) $\mathcal{P}$ proves that the first MULT output of both subcircuits is 1 and that both circuits check to 0:

$$o_1 = o_{n^{(1)}+1} = 1 \text{ and } o_{n^{(1)}} = o_{n^{(2)}} = 0$$

(3) $\mathcal{P}$ proves in ZK that the committed values and $\boldsymbol{M}$ satisfy Equation (3). To achieve this, $\mathcal{V}$ issues a uniform challenge χ and $\mathcal{P}$ proves in ZK that:

$$\begin{aligned}&\overbrace{(1, \chi, \ldots, \chi^{2n-1}) \times \boldsymbol{M}}^{\text{topology vector } \boldsymbol{c}} \times \overbrace{(in_1, o_1, \ldots, in_n, o_n)^T}^{\text{committed}} \\ &= \underbrace{(1, \chi, \ldots, \chi^{2n-1})}_{\text{public}} \times \underbrace{(\ell_1, r_1, \ldots, \ell_n, r_n)^T}_{\text{committed}}\end{aligned}$$

To achieve the above steps while hiding C (and $\boldsymbol{M}$), $\mathcal{P}$ commits to two additional vectors. The first is an appropriate boundary string (see Section 4.1) $\boldsymbol{p}$:

$$\boldsymbol{p} \triangleq \overbrace{0, \ldots, 0}^{n^{(1)}-1}, 1, \overbrace{0, \ldots, 0}^{n^{(2)}-1}, 1$$

The second vector $\boldsymbol{id}$ places the index of each branch at that branch's boundary, and elsewhere $\mathcal{P}$ fills the vector with any values in $[B]$:

$$\boldsymbol{id} \triangleq \overbrace{\text{any values in } [B]}^{n^{(1)}-1}, 1, \overbrace{\text{any values in } [B]}^{n^{(2)}-1}, 2$$

Looking ahead, these branch IDs will be used as indices to load instruction hashes from a ZK ROM (entries not on boundaries are dummy indices). The definition of $\boldsymbol{id}$ implies that $\mathsf{Filter}(\boldsymbol{p}, \boldsymbol{id})$ outputs a vector of branch IDs (see Section 4.1 for Filter's definition). Informally, $\boldsymbol{p}$ and $\boldsymbol{id}$ jointly form a commitment to a particular execution path.

At a high level, our protocol leverages $\boldsymbol{p}$ and $\boldsymbol{id}$ to cheaply express Steps 2 and 3 as ZK constraints. In detail:

(1) Step 1 only depends on n and is *independent* of $\boldsymbol{M}$. $\mathcal{P}$ commits to her inputs and to well-formed MULT tuples.
(2) Step 2 can be performed by checking the constraints:
  (a) $\boldsymbol{p} \in \{0,1\}^{n-1} \| 1$. I.e., $\boldsymbol{p}$ is a boundary string.
  (b) If $p_{i\in[n]} = 1$, o_i must be 0.
  (c) $o_1 = 1$, and if $p_{i\in[n-1]} = 1$, o_{i+1} must be 1.
  The above constraints can be checked very efficiently.
(3) To perform Step 3, $\mathcal{V}$ cannot construct the topology vector $\boldsymbol{c}$, as $\boldsymbol{M}$ is private. Instead, our protocol requires that $\mathcal{P}$ *commits* to $\boldsymbol{c}$. Of course, $\mathcal{P}$ might attempt to cheat, so we need extra checks that ensure $\mathsf{com}(\boldsymbol{c})$ is properly constructed and is consistent with $\boldsymbol{p}$ and $\boldsymbol{id}$. We will soon show how this can be achieved via a so-called ZK unbalanced ROM (Section 4.4). For now, simply assume that $\mathcal{P}$ commits to the vector:

$$\boldsymbol{c} = (1, \chi, \ldots, \chi^{2n-1}) \times \boldsymbol{M}$$

Crucially, *private* $\boldsymbol{M}$ has *a special structure* – it has square matrices on the diagonal and 0s elsewhere. In particular, these square matrices are determined and ordered by the private execution path. I.e., it (in order) includes $\boldsymbol{M}^{(j)}$ for each $j \in \mathsf{Filter}(\boldsymbol{p}, \boldsymbol{id})$ in order. Note that each $\boldsymbol{M}^{(i\in[B])}$ is public. Finally, once we have $\mathsf{com}(\boldsymbol{c})$, it suffices to show that:

$$\langle \boldsymbol{c}, (in_1, o_1, \ldots, in_n, o_n) \rangle = \langle (1, \ldots, \chi^{2n-1}), (\ell_1, r_1, \ldots, \ell_n, r_n) \rangle$$

4.3.2 Handling Constant 1. Recall that the first MULT gate in each instruction should output 1 defined as $1 \cdot 1 = 1$, enabling that instruction to manipulate the constant 1. As a remark, it is surprisingly difficult to incorporate constants in our approach, because our constraint systems are merely *linear* (and not affine) over $\mathbb{F}$. Sub-step 2c forces that the output of the first MULT gate is 1. Here, we show an optimized way to ensure that the output of this MULT is 1 *for free* by directly constraining its *inputs*. Our idea is to pass the constant 1 from one instruction to the next and, looking forward, this same handling will be used to enable the passing of m registers.

A naïve (failing) attempt to pass a 1 into an instruction would be to have a fixed wire of C carrying 1, to which each instruction can refer. However, we are working with a fixed instruction set (and we check hashes of executed instructions against the corresponding set of hashes). Informally, we *could* make an instruction reference a fixed wire in C, outside of itself. However, due to our use of topology matrices, under the hood (i.e., in the supporting matrix algebra) such an instruction will access this wire via an offset to its own position on the execution path, resulting in a *unique* instruction (topology matrix) hash. Such an instruction cannot be checked against the fixed instruction set (IS).

Thus, our instructions cannot refer to wires by their absolute position, but they *can* refer to wires via a fixed offset relative to their own position on the execution path. Indeed, our solution, at the high level, is for each instruction to "push forward" a 1 wire to the next instruction. This is possible because each instruction knows its own length, and can set up the corresponding constraint for the next instruction. Each instruction $C_{i\in[B]}$ has a *fixed* offset to access (enforce) input constraints (via left/right wires of MULTs) of the next instruction. Thus, $C_{i\in[B]}$'s topology matrix (and hence hash) will be the same anywhere on the execution path. The very first instruction can pick up the 1 from a designated wire of C.

This cleanly translates into our matrix representation. Let us go through our concrete example of $\mathcal{P}$ proving a circuit C consisting of C_1 followed by C_2. Formally, the entire proof will be based on a (slightly) updated equation:

$$\begin{array}{c|c}\begin{array}{c}\boldsymbol{M} \times (1, in_1, o_1, \ldots, in_n, o_n)^T \\ = (\ell_1, r_1, \ldots, \ell_n, r_n, 1, 1)^T\end{array} & \boldsymbol{M} \triangleq \begin{pmatrix} 1 & \boldsymbol{0} & \boldsymbol{0} \\ 1 & \boldsymbol{0} & \boldsymbol{0} \\ 0 & \boldsymbol{M}_*^{(1)} & \boldsymbol{0} \\ 0 & \boldsymbol{0} & \boldsymbol{M}_*^{(2)} \end{pmatrix}\end{array}$$

where each $\boldsymbol{M}_*^{(i\in[B])} = \begin{pmatrix} \boldsymbol{M}^{(i)}(3) \\ \cdots \\ \boldsymbol{M}^{(i)}(2n^{(i)}) \\ 0\ 1\ 0\ \cdots \\ 0\ 1\ 0\ \cdots \end{pmatrix}$ is public. (Here $\boldsymbol{M}_*^{(i\in[B])}$ omits the first two constraints of $\boldsymbol{M}^{(i\in[B])}$, which define left/right wires of a MULT generating 1. As a complement, the last two rows of $\boldsymbol{M}_*^{(i\in[B])}$ constrain the next instruction's left/right wires of the MULT generating 1.[2]) The IS will consist of B instructions $\boldsymbol{M}_*^{(i\in[B])}$.

[2]The first MULT $\ell_1 \cdot r_1 = o_1$ must be $1 \cdot 1 = 1$ as $\ell_1 = r_1 = \langle (1, in_1, \ldots), (1, 0, \ldots) \rangle = 1$.

Crucially, while $\boldsymbol{M}$ is private, the first two rows of $\boldsymbol{M}$ are fixed and public. We need to construct the vector commitment of $(1, \chi, \ldots, \chi^{2n+1}) \times \boldsymbol{M} = (1 + \chi)\|(\chi^2(1, \ldots, \chi^{2n-1}) \times \boldsymbol{M}_*)$, where $\boldsymbol{M}_* = \begin{pmatrix} \boldsymbol{M}_*^{(1)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{M}_*^{(2)} \end{pmatrix}$. Hence, it suffices to construct the commitments of $(1, \ldots, \chi^{2n-1}) \times \boldsymbol{M}_*$, the problem discussed in Step 3 of Section 4.3.1 and postponed to Section 4.3.4.

Similarly to our importing a $1 = 1 \cdot 1$ into an instruction, we will import registers via reg $= 1 \cdot$ reg:

4.3.3 Supporting Registers. Extending our idea of passing 1, we support register passing between two adjacently executed instructions. We view each register as a MULT, where the previous instruction defines MULT's left/right wires. The translation of this into the matrix representation is similar to our handling of $1 \cdot 1 = 1$. Consider the case with a single register as a simple example (the order of gates follows Section 4.3.1). We can (re)define the public matrix

$$\boldsymbol{M}^{(i)} \triangleq \begin{pmatrix} \text{define } \ell_3 \\ \text{define } r_3 \\ \cdots \\ \text{define } \ell_{n_\times^{(i)}+2} \\ \text{define } r_{n_\times^{(i)}+2} \\ 0\ 1\ 0\ \cdots\ \text{(define 1)} \\ \text{define checking output} \\ 0\ 1\ 0\ \cdots\ \text{(define 1)} \\ 0\ 1\ 0\ \cdots\ \text{(define 1)} \\ 0\ 1\ 0\ \cdots\ \text{(define 1)} \\ \text{define first register} \end{pmatrix} \in \mathbb{F}^{2n^{(i)} \times 2n^{(i)}} \tag{5}$$

for each $i \in [B]$. Here, the last two rows of $\boldsymbol{M}^{(i)}$ set the first register (as inputs to a MULT of the next instruction). The prior two rows similarly set a 1 for the next instruction.

Now, suppose $\mathcal{P}$ wants to prove the execution of C_1 followed by C_2, where the register is initialized to x as C's input and stores y as C's output (x, y are public). $\mathcal{P}$ can commit $n = n^{(1)} + n^{(2)}$ inputs and MULT tuples and show:

$$\begin{array}{c|c} \begin{gathered} \boldsymbol{M} \times (1, x, in_1, o_1, \ldots, in_n, o_n)^T \\ = (\ell_1, r_1, \ldots, \ell_n, r_n, 1, 1, 1, y)^T \end{gathered} & \boldsymbol{M} \triangleq \begin{pmatrix} 1 & 0 & \boldsymbol{0} & \boldsymbol{0} \\ 1 & 0 & \boldsymbol{0} & \boldsymbol{0} \\ 1 & 0 & \boldsymbol{0} & \boldsymbol{0} \\ 0 & 1 & \boldsymbol{0} & \boldsymbol{0} \\ 0 & 0 & \boldsymbol{M}^{(1)} & \boldsymbol{0} \\ 0 & 0 & \boldsymbol{0} & \boldsymbol{M}^{(2)} \end{pmatrix} \end{array}$$

$\boldsymbol{M}$ is private but $\mathcal{P}$ and $\mathcal{V}$ can obtain the commitment of $(1, \chi, \chi^2, \ldots) \times \boldsymbol{M}$ by constructing the commitment of $(1, \ldots, \chi^{2n-1}) \times \begin{pmatrix} \boldsymbol{M}^{(1)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{M}^{(2)} \end{pmatrix}$ (discussed next).

4.3.4 Committing to the Topology Vector. We now show how $\mathcal{P}$ and $\mathcal{V}$ can construct $\mathsf{com}(\boldsymbol{c})$, a crucial task postponed from Section 4.3.1. The methodology applies to Sections 4.3.2 and 4.3.3. We explain it on the special case of two instructions $C_2 \circ C_1$; our discussion applies generally. We exploit the following equality:

$$\begin{aligned} \boldsymbol{c} &= (1, \chi, \ldots, \chi^{2n-1}) \times \boldsymbol{M} \\ &= (1, \ldots, \chi^{2n^{(1)}-1}) \times \boldsymbol{M}^{(1)} \| (\chi^{2n^{(1)}}, \ldots, \chi^{2n^{(1)}+2n^{(2)}-1}) \times \boldsymbol{M}^{(2)} \\ &= (1, \ldots, \chi^{2n^{(1)}-1}) \times \boldsymbol{M}^{(1)} \| \chi^{2n^{(1)}} \cdot (1, \ldots, \chi^{2n^{(2)}-1}) \times \boldsymbol{M}^{(2)} \\ &= \left(\boldsymbol{a} \triangleq (\underbrace{1, \ldots, 1}_{2n^{(1)}}, \overbrace{\chi^{2n^{(1)}}, \ldots, \chi^{2n^{(1)}}}^{2n^{(2)}}) \right) \odot \\ &\quad \left(\boldsymbol{b} \triangleq \underbrace{(1, \chi, \ldots, \chi^{2n^{(1)}-1})}_{2n^{(1)}} \times \boldsymbol{M}^{(1)} \| \underbrace{(1, \chi, \ldots, \chi^{2n^{(2)}-1})}_{2n^{(2)}} \times \boldsymbol{M}^{(2)} \right) \end{aligned}$$

Hence, to construct $\mathsf{com}(\boldsymbol{c})$, it suffices to construct $\mathsf{com}(\boldsymbol{a})$ and $\mathsf{com}(\boldsymbol{b})$. Note, $\boldsymbol{a}$ is a structured vector based on χ and $\boldsymbol{p}$ (see Section 4.1, Expand_1). We only need to construct $\mathsf{com}(\boldsymbol{b})$, and the crucial observation is the following vectors are *public*:

$$\forall i \in [B], \boldsymbol{v}^{(i)} \triangleq (1, \chi, \ldots, \chi^{2n^{(i)}-1}) \times \boldsymbol{M}^{(i)}$$

The functionality we need is to "load" from *unbalanced* ROM then "concatenate" $\boldsymbol{v}^{(1)}$ and $\boldsymbol{v}^{(2)}$. This can be viewed as

(1) $\mathcal{P}$ and $\mathcal{V}$ agree on an unbalanced read-only memory (ROM) storing (public) entries $(1, \boldsymbol{v}^{(1)}), \ldots, (B, \boldsymbol{v}^{(B)})$.
(2) $\mathcal{P}$ and $\mathcal{V}$ load-concatenate $\boldsymbol{v}^{(i \in [B])}$s where the ordered indexes are decided by $\mathsf{Filter}(\boldsymbol{p}, \boldsymbol{id})$.

Note that these vectors saved in ROM are randomized by $\mathcal{V}$'s uniform challenge sent after $\boldsymbol{p}$ and $\boldsymbol{id}$ have been committed. As are instructions, these vectors are of *different* lengths. We capture this as a (more generic and novel) hybrid functionality *ZK Unbalanced ROM* (ZK UROM) and include the overview in Section 4.4. Crucially, the access cost of our ZK UROM is proportional to the length of the data retrieved – this is needed to meet our tight efficiency budget.

4.4 ZK Non-Zero-End Unbalanced ROM

This section overviews how to reduce a ZK UROM to a ZK ROM.

We observe that it suffices to design a ZK UROM supporting only non-zero-end vectors. This simplifies our task, enabling concise soundness checks based on Corollary 1, and can always be achieved, e.g., by padding. (We later show that padding is not needed for us.)

In ZK non-zero-end UROM, $\mathcal{P}$ and $\mathcal{V}$ agree on a set of key-value tuples $(1, \boldsymbol{v}^{(1)}), \ldots, (B, \boldsymbol{v}^{(B)})$, where $\boldsymbol{v}^{(i \in [B])}$ are non-zero-end vectors in $\mathbb{F}$ that can have *different* lengths. The objective is allowing $\mathcal{P}$ to commit to a vector $\boldsymbol{v}$, a concatenation of several $\boldsymbol{v}^{(i \in [B])}$s, e.g., $\boldsymbol{v} \triangleq \boldsymbol{v}^{(1)} \| \boldsymbol{v}^{(2)} \| \boldsymbol{v}^{(1)}$. Crucially, $\mathcal{V}$ should only learn $n \triangleq |\boldsymbol{v}|$ and be convinced that $\boldsymbol{v}$ is a concatenation of vectors from UROM. Prior work (e.g., [42], on which we build) only considers ZK ROM over vectors of equal length (see Section 2.4).

Our ZK UROM protocol works in the commit-and-prove paradigm. I.e., we require $\mathcal{P}$ to directly commit to $\boldsymbol{v}$ and prove in ZK that $\boldsymbol{v}$ is a valid concatenation. To support this proof, $\mathcal{P}$ additionally commits how she wants to partition $\boldsymbol{v}$. That is, $\mathcal{P}$ commits a length-n boundary string $\boldsymbol{p}$ and a length-n vector $\boldsymbol{id} \in [B]^n$ such that for each $x \in \mathsf{Filter}(\boldsymbol{p}, \boldsymbol{id})$ and $\boldsymbol{y} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{v})$ pair (total $\mathsf{HW}(\boldsymbol{p})$ pairs, unknown to $\mathcal{V}$) in sequence, $\boldsymbol{y} = \boldsymbol{v}^{(x)}$.

To begin with, consider a simplified single-read task: $\mathcal{P}$ commits a vector $\boldsymbol{w}$ and a single index t and wants to prove that $\boldsymbol{w} = \boldsymbol{v}^{(t)}$. This can be checked by $\mathcal{V}$ issuing a uniform challenge $\gamma \in \mathbb{F}$ where parties agree on another *balanced* ROM storing K-V tuples:

$(1, mac^{(1)}), \ldots, (B, mac^{(B)})$ where $mac^{(i)} \triangleq (1, \gamma, \gamma^2, \ldots) \times \boldsymbol{v}^{(i)} \in \mathbb{F}$ for each $i \in [B]$. Now, by accessing the ZK ROM (see Section 2.4), parties convert $\mathsf{com}(t)$ into $\mathsf{com}(mac^{(t)})$. Then, it suffices to show:

(1) $\mathsf{last}(\boldsymbol{w}) \neq 0$. This can be proved by requiring $\mathcal{P}$ to commit a value inv and show that $\mathsf{last}(\boldsymbol{w}) \cdot inv = 1$.
(2) $\langle (1, \gamma, \gamma^2, \ldots), \boldsymbol{w} \rangle = mac^{(t)}$. This can be proved by opening $\mathsf{com}(\langle (1, \gamma, \gamma^2, \ldots), \boldsymbol{w} \rangle - mac^{(t)})$ (which should be 0). Note that γ is public and parties hold $\mathsf{com}(\boldsymbol{w}), \mathsf{com}(mac^{(t)})$.

Soundness is reduced to Corollary 1 as $\mathcal{P}$ is prevented by Step 1 from appending the returned vector with zeros.

Our ZK UROM protocol generalizes the above idea to $\boldsymbol{v}$ with the help of committed $\boldsymbol{p}$ and $\boldsymbol{id}$. In particular, since $\boldsymbol{p}$ already marks where each subvector ends, and the corresponding committed $\boldsymbol{id}$ includes the index of each subvector, we can perform the above checks only at the position where $p_i = 1$. That is, $\mathcal{P}$ and $\mathcal{V}$ perform a check for each position, but checks in positions where $p_i = 0$ are dummy. Formalizing the above, we outline our protocol:

(1) $\mathcal{V}$ issues a uniform challenge $\gamma \in \mathbb{F}$ where $\mathcal{P}$ and $\mathcal{V}$ agree on another *balanced* ROM storing K-V tuples $\{(i, mac^{(i)})\}_{i \in [B]}$ where (public) $mac^{(i)} \triangleq (1, \gamma, \gamma^2, \ldots) \times \boldsymbol{v}^{(i)}$ for each $i \in [B]$.
(2) $\mathcal{P}$ and $\mathcal{V}$ generate committed "selected *mac*s" $\mathsf{com}(\boldsymbol{smac})$ by "reading" single-element ZK ROM (see Section 2.4) initialized by $mac^{(1)}, \ldots, mac^{(B)}$ at positions $\boldsymbol{id}$, where each $smac_{i \in [n]} = mac^{(id_i)}$. We remark that $\boldsymbol{id}$ is fixed before γ.
(3) $\mathcal{P}$ and $\mathcal{V}$ generate commitment of the structured vector $\boldsymbol{s}$ based on γ and $\boldsymbol{p}$ via Expand_2 (see Section 4.1):

$$\boldsymbol{p} = (\overbrace{0, \ldots, 0, 1}^{n_1}, \overbrace{0, \ldots, 0, 1}^{n_2}, \overbrace{0, \ldots, 0, 1}^{n_3}, \ldots)$$
$$\Rightarrow \boldsymbol{s} = (1, \gamma, \ldots, \gamma^{n_1 - 1}, 1, \gamma, \ldots, \gamma^{n_2 - 1}, 1, \gamma, \ldots, \gamma^{n_3 - 1}, \ldots)$$

(4) $\mathcal{P}$ proves that for each $p_{i \in [n]} = 1$, it holds $v_i \neq 0$. (I.e., each segment ends non-zero.) This corresponds to the check in Step 1 of the single-read task. This can be performed by requiring $\mathcal{P}$ to commit to another length-n vector $\boldsymbol{inv}$ where

$$inv_i = (v_i)^{-1} \text{ if } p_i = 1; inv_i = 0 \text{ otherwise}$$

$\mathcal{P}$ then shows that $\boldsymbol{inv} \odot \boldsymbol{v} - \boldsymbol{p} = \boldsymbol{0}$.
(5) $\mathcal{P}$ proves that for each $\boldsymbol{a} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{s}), \boldsymbol{b} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{v})$, $\boldsymbol{c} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{smac})$ (in order, total $\mathsf{HW}(\boldsymbol{p})$ tuples), $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \mathsf{last}(\boldsymbol{c})$. This corresponds to the check in Step 2 of the single-read task. This can be performed by proving:

$$\forall i \in [n], p_i \cdot (\langle \boldsymbol{s}[:i], \boldsymbol{v}[:i] \rangle - \langle \boldsymbol{p}[:i], \boldsymbol{smac}[:i] \rangle) = 0$$

Note, the above equality trivially holds for all $p_i = 0$. Moreover, when p_i is equal to 1, both $\langle \boldsymbol{s}[:i], \boldsymbol{v}[:i] \rangle$ and $\langle \boldsymbol{p}[:i], \boldsymbol{smac}[:i] \rangle$ are accumulating the sum of *mac*s used so far. Importantly, $\mathcal{P}$ and $\mathcal{V}$ *do not* compute these sums for each position separately, which incurs quadratic overhead. Rather, they accumulate a running total, which is being checked at each step. Thus, the total complexity of this check is linear.

4.4.1 Using ZK UROM with Topology Vectors. Recall, our protocol for ZK CPU is reduced to a ZK UROM, where the data are the instructions' topology vectors. In the course of this reduction, $\mathcal{P}$ and $\mathcal{V}$ generate commitments to $\boldsymbol{p}$ and $\boldsymbol{id}$ (see Section 4.3). We need these commitments for the operation of UROM as well. The low-level format of these vectors is different from what UROM needs: while the vectors, as described in Section 4.3 manage *gates*, UROM needs to account for two wires for each of these gates. This discrepancy is easily reconciled (by inserting 0 to $\boldsymbol{p}$ and replicating $\boldsymbol{id}$), and we can work with a single copy of $\boldsymbol{p}$ and $\boldsymbol{id}$.

A more subtle issue is that each topology vector ends with 0. This is because the last column of a topology matrix denotes the contribution of the last output of the instruction to each wire. Note that the last output represents the checking output of the instruction, which is not an input of any wire, resulting in the all-0 last column of the topology matrix (ultimately producing the 0-end topology vector). This does *not* fit the *non-zero-end* requirement!

While this can be resolved by appending 1, we resolve it more efficiently as follows. Since the checking output in a valid instruction is 0, we simply add it into the instruction's first (left) wire. This does not change the function of the instruction, and guarantees that the last column now has a single leading 1. This modification will make each topology vector end with 1. Further, in our proof we need to invert the last position of each topology vector; having set it to 1 optimizes this task. Namely, the vector $\boldsymbol{inv}$ committed by $\mathcal{P}$ in Step 4 is precisely the boundary string $\boldsymbol{p}$.

5 Formalization

This section formalizes our approach. See Section 4 for a detailed overview of our approach. Due to space constraints, we defer some formalization to the appendices.

5.1 Ideal ZK Non-Zero-End UROM: $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$

We define the ideal functionality for CPZK with a single read-only memory for *unbalanced, non-zero-end* vectors, denoted $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ and presented in Figure 6. $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ is defined similarly to $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$. The main difference is that $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ allows $\mathcal{P}$ to initialize the UROM with different-length vectors (via `InitUROM`). Furthermore, $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ allows $\mathcal{P}$ to read a length-n vector $\boldsymbol{d}$ from the UROM (via `ReadUROM`). Vector $\boldsymbol{d}$ must partition into subvectors where each subvector is a UROM entry. Before calling `ReadUROM`, $\mathcal{P}$ can choose the content it wishes to read via `SetProg`. This choice is encoded by length-n vectors $\boldsymbol{p}$ and $\boldsymbol{id}$, where $\boldsymbol{p}$ is the boundary string encoding how $\mathcal{P}$ wishes to partition $\boldsymbol{d}$ and $\mathsf{Filter}(\boldsymbol{p}, \boldsymbol{id})$ is the (ordered) set of indices $\mathcal{P}$ wishes to read. The flag f_{urom} is used to catch malicious $\mathcal{P}^*$ misbehaviors.

5.2 Our Protocols: $\Pi_{\mathsf{CPZK\text{-}UROM}}$ and Π_{ZKCPU}

Our tight ZK CPU protocol (Π_{ZKCPU}) is designed in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid model, and our ZK UROM protocol ($\Pi_{\mathsf{CPZK\text{-}UROM}}$) is designed in the $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid model; see Section 4. We defer the complete UC-style protocol definitions to the full version [44].

Here, we state the security theorems regarding these two protocols. We defer the proof sketches (resp. complete proofs) to [44].

Theorem 1. *Let the UROM be initialized with B non-zero-end vectors where each i-th vector is of length-$n^{(i)}$. Let the read-out vector be of length-n. Then, protocol $\Pi_{\mathsf{CPZK\text{-}UROM}}$ (defined in [44]) UC-realizes $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ (Figure 6) in the $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid model (Figure 3) with soundness error $\frac{\max\{n, n^{(1)}, \ldots, n^{(B)}\} - 1}{|\mathbb{F}|}$ and perfect zero-knowledge, in the presence of a static unbounded adversary.*

Functionality $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$

$\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, parameterized by a field $\mathbb{F}$, proceeds as follows, running with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$:

CPZK

The functionality supports all instructions of $\mathcal{F}_{\mathsf{CPZK}}$.

Unbalanced Non-Zero-End Read-Only Memory

Initialize UROM. On receiving $(\mathtt{InitUROM}, \boldsymbol{u}^{(1)}, \dots, \boldsymbol{u}^{(B)})$ from $\mathcal{P}$, where for each $\boldsymbol{u}^{(i\in[B])} = (u_1^{(i)}, \dots, u_{n^{(i)}}^{(i)})$, each $u_{j\in[n^{(i)}]}^{(i\in[B])}$ is recorded as a *cid* (i.e., the unbalanced vectors were committed):

(1) For each $i \in [B]$, fetch $(u_1^{(i)}, x_1^{(i)}), \dots, (u_{n^{(i)}}^{(i)}, x_{n^{(i)}}^{(i)})$ and let $\boldsymbol{x}^{(i)} := (x_1^{(i)}, \dots, x_{n^{(i)}}^{(i)})$. Halt if $\mathsf{last}(\boldsymbol{x}^{(i)}) = 0$. ($\mathsf{last}(\boldsymbol{x}^{(i)})$ must be a non-zero element if $\mathcal{P}$ is honest.)

(2) Create a key-value store X where
$$X[1] := \boldsymbol{x}^{(1)}, \cdots, X[B] := \boldsymbol{x}^{(B)}$$
and set $f_{\mathsf{urom}} := \mathtt{honest}$.

(3) Send $(\mathtt{initurom}, \boldsymbol{u}^{(1)}, \dots, \boldsymbol{u}^{(B)})$ to $\mathcal{V}$ and $\mathcal{S}$.

Ignore the subsequent calls to $\mathtt{InitUROM}$.

Set Program. On receiving $(\mathtt{SetProg}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$ from $\mathcal{P}$ where $|\boldsymbol{cid}^{(p)}| = |\boldsymbol{cid}^{(p)}| = n \in \mathbb{Z}^+$ and each $cid_{i\in[n]}^{(p)}$, $cid_{i\in[n]}^{(id)}$ was recorded: Fetch $(cid_i^{(p)}, p_i)$, $(cid_i^{(id)}, id_i)$ for each $i \in [n]$. Record $\boldsymbol{p}$ and $\boldsymbol{id}$. If $\boldsymbol{p} \in \{0,1\}^{n-1}\|1$, send $(\mathtt{setprog}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$ to $\mathcal{V}$ and $\mathcal{S}$; otherwise, halt the functionality. (If $\mathcal{P}$ is honest, $\boldsymbol{p}$ must be a length-n boundary string, i.e., $\boldsymbol{p} \in \{0,1\}^{n-1}\|1$.) Ignore the subsequent calls to $\mathtt{SetProg}$.

Read UROM. On receiving $(\mathtt{ReadUROM}, \boldsymbol{cid}^{(d)}, \boldsymbol{d})$ from $\mathcal{P}$ where (1) $\mathtt{InitUROM}$ and $\mathtt{SetProg}$ were executed; (2) $|\boldsymbol{cid}^{(d)}| = |\boldsymbol{d}| = |\boldsymbol{p}| = |\boldsymbol{id}| = n$; (3) there is no recorded tuple for each $cid_{i\in[n]}^{(d)}$; and (4) each $d_{i\in[n]} \in \mathbb{F}$: Record tuples $(cid_1^{(d)}, d_1), \dots, (cid_n^{(d)}, d_n)$.

(1) If $\mathcal{P}$ is honest, $\boldsymbol{id} \in [B]^n$.

(2) If $\mathcal{P}$ is corrupted, set $f_{\mathsf{urom}} := \mathtt{cheating}$ when there exists some $i \in [n]$ such that $id_i \notin [B]$.

For each $\boldsymbol{x} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{d})$, $\boldsymbol{y} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{id})$ pair in order (there are $\mathsf{HW}(\boldsymbol{p})$ pairs in total):

(3) If $\mathcal{P}$ is honest, $\mathsf{last}(\boldsymbol{x}) \neq 0$ and $X[\mathsf{last}(\boldsymbol{y})] = \boldsymbol{x}$.

(4) If $\mathcal{P}$ is corrupted, set $f_{\mathsf{urom}} := \mathtt{cheating}$ when:
$$\mathsf{last}(\boldsymbol{x}) = 0 \text{ or } X[\mathsf{last}(\boldsymbol{y})] \neq \boldsymbol{x}$$

Send $(\mathtt{readurom}, \boldsymbol{cid}^{(d)})$ to $\mathcal{V}$, $\mathcal{S}$. Ignore subsequent $\mathtt{ReadUROM}$ calls.

Check UROM. On receiving $(\mathtt{CheckUROM})$ from $\mathcal{P}$ where $\mathtt{ReadUROM}$ was executed: If $\mathcal{P}$ is corrupted and $\mathcal{S}$ sends $\mathtt{Cheat}$, set $f_{\mathsf{urom}} := \mathtt{cheating}$. Send $(\mathtt{checkurom}, f_{\mathsf{urom}})$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 6: Ideal functionality for commit-and-prove zero-knowledge allowing proofs that support a read-only memory for unbalanced non-zero-end vectors.

Theorem 2. *Protocol Π_{ZKCPU} (defined in [44]) UC-realizes $\mathcal{F}_{\mathsf{ZKCPU}}$ (Figure 5) in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid model (Figure 6) with soundness error $\frac{2m+2n+1}{|\mathbb{F}|}$ and perfect zero-knowledge, in the presence of a static unbounded adversary.*

5.3 Optimization and Cost Analysis

The optimization of $\Pi_{\mathsf{CPZK\text{-}UROM}}$ (defined in [44]) includes:

(1) *Public initialization*: If B vectors used to initialize UROM are public, $\mathtt{InitUROM}$ is free. This is because $\boldsymbol{u}^{(i\in[B])}$ is only used to generate commitments of $\boldsymbol{mac}$ (see $\Pi_{\mathsf{CPZK\text{-}UROM}}$ in [44]), which are further used to initialize the underlying (balanced) ROM. Thus, $\boldsymbol{mac}$ is also public (determined after γ is selected by $\mathcal{V}$), so $\mathcal{P}$ and $\mathcal{V}$ can compute $\boldsymbol{mac}$ locally and use calls to $\mathtt{Linear}$ construct the commitment of (constant).

(2) 1-*ended vectors*: If each vector in the UROM ends with 1 (whose inverse is 1), vector $\boldsymbol{inv}$ is redundant (see $\Pi_{\mathsf{CPZK\text{-}UROM}}$ in [44]) since $\boldsymbol{inv}$ is equal to $\boldsymbol{p}$.

(3) *Rounding optimization*: If each UROM-stored vector has length some multiple of $\varepsilon_{\mathsf{urom}}$, for any $\varepsilon_{\mathsf{urom}} \in \mathbb{Z}^+$, we can optimize some operations. E.g., consider $\varepsilon_{\mathsf{urom}} = 2$, i.e., each $n^{(i\in[B])}$ is even. This implies that every odd position of $\boldsymbol{p}$ must be 0, which further implies that the checks in $C_{1/2/3}^{\mathsf{check}}$ only need to be performed at each even position. Thus, $\mathcal{P}$ only needs to commit length-$\frac{n}{2}$ vectors (instead of length-n) $\boldsymbol{p}$, $\boldsymbol{id}$, $\boldsymbol{inv}$, $\boldsymbol{smac}$, $\boldsymbol{s}$ with half-size $C_{1/2}^{\mathsf{check}}$. In particular, it suffices to define $\boldsymbol{s}$ as $\mathsf{Expand}_2(\boldsymbol{p}, \gamma^2)$. More generally, these commitments reduce in size by factor $\varepsilon_{\mathsf{urom}}$.

The protocol Π_{ZKCPU} (see the full version [44]) can deploy *all* optimizations above and will make one call to each instruction (i.e., $\mathtt{InitUROM}$, $\mathtt{SetProg}$, $\mathtt{ReadUROM}$, $\mathtt{CheckUROM}$). In particular, Π_{ZKCPU}, with instructions of size $n^{(1)}, \dots, n^{(B)}$ and the total execution size n, instantiates a hybrid UROM with vectors of size $2n^{(1)}, \dots, 2n^{(B)}$, and reads a length $2n$ vector from the UROM. Our Π_{ZKCPU} instantiates the UROM with *public* vectors ending with 1, and since all vectors are of even length, we can deploy the above rounding optimization. Moreover, a similar rounding optimization can be deployed to Π_{ZKCPU}. I.e., if the size of each instruction is an integer factor of $\varepsilon \in \mathbb{Z}^+$, we can save cost by constructing shorter vectors, e.g., $\boldsymbol{p}$. In other words, cost can be reduced if we pad each instruction circuit to size $k\varepsilon$, where $k \in \mathbb{Z}^+$.

Cost analysis. Consider a ZK CPU with instructions of size $n^{(1)}$, $\dots, n^{(B)}$ and the total execution size n, let $\varepsilon \triangleq \gcd(n^{(1)}, \dots, n^{(B)})$, we **tally the optimized cost** of Π_{ZKCPU} directly in $\mathcal{F}_{\mathsf{CPZK}}$-hybrid (i.e., plugging $\Pi_{\mathsf{CPZK\text{-}UROM}}$, $\Pi_{\mathsf{CPZK\text{-}ROM}}$):

- $\mathcal{P}$ sends n and $\mathcal{V}$ sends χ, γ.
- $\mathcal{P}$ and $\mathcal{V}$ each compute $O\left(\sum_{i\in[B]} n^{(i)}\right)$ field operations to obtain $\boldsymbol{v}^{(i\in[B])}$ and $\boldsymbol{mac}$. Note, this relies on the technique "evaluate circuits backward"; see [43].
- Parties call $\mathtt{Commit}$ $6n + \frac{6n}{\varepsilon} + 2B$ times.
- Parties call $\mathtt{Linear}$ $2B + 1$ times to commit *constants*.
- Parties call $\mathtt{Open}$ once.
- Parties call $\mathtt{Check}$ with each of the following 9 circuits (defined in $\Pi_{\mathsf{CPZK\text{-}UROM}}$, $\Pi_{\mathsf{CPZK\text{-}ROM}}$; see the full version [44]):
  - $C_{1/2/5}^{\mathsf{check}}$ and $\mathsf{Expand}_{1/2}$ ($\frac{n}{\varepsilon}$ multiplications each).
  - C_3^{check} ($2n + \frac{2n}{\varepsilon}$ multiplications).
  - C_4^{check} (n multiplications).
  - C_6^{check} ($4n$ multiplications).
  - The check circuit in $\Pi_{\mathsf{CPZK\text{-}ROM}}$ (see Lemma 3), which has two products of $\frac{n}{\varepsilon} + B - 1$ multiplication.

To conclude, assuming $n = \Omega(B)$ and assuming each instruction is of size $O(n)$, the protocol requires $O(n)$ calls to `Commit`; $O(B)$ calls to `Linear`; $O(1)$ call to `Open`; $O(1)$ call to `Check`.

When we instantiate $\mathcal{F}_{\mathsf{CPZK}}$ using VOLE-based ZK (see Lemma 1), our ZK CPU has the following cost:

- **Computation**: $O\left(n + \sum_{i\in[B]} n^{(i)}\right)$ field operations.
- **Communication**: $6n + \frac{6n}{\epsilon} + B + o(n)$ field elements.
- **Soundness**: $O\left(\frac{m+\max\{n,n^{(1)},\ldots,n^{(B)}\}}{|\mathbb{F}|}\right)$.

The above costs leverage VOLE-based ZK's support for polynomial evaluation (see formal version of Lemma 1 in [44]). Namely, circuits used in `Check` are polynomials of degree[3] 2 or 3. Note, both computation and communication are proportional to n.

[44] includes more fine-grained cost analysis.

6 Support for Advanced Operations

We have shown how to construct instructions that contain arbitrary addition and multiplication gates. Each instruction also supports a checking output, which $\mathcal{P}$ must prove is equal to zero, and in this section, we discuss examples of how this checking output can be leveraged to support more advanced ZK operations. Most importantly, we discuss support for ZK RAM, which enables our CPU to support poly-size memory, rather than just a fixed number of registers. Our formalization must be adjusted slightly to capture such operations; the following discusses how.

6.1 Equality Gates

As our first advanced operation, we show how to implement an *equality* gate, which forces $\mathcal{P}$ to prove that two particular instruction wires are equal; if they are not equal, the proof fails. This gate is generally useful, and it can enable efficient implementation of other operations, such as a division gate, where we can require $\mathcal{P}$ to commit the quotient and then prove that the product of the quotient and the divisor is equal to the dividend.

In standard CPZK, it is well known that a batch of equality gates can be implemented by subtracting each pair of supposedly-equal commitments, then having $\mathcal{V}$ send a uniform challenge vector to $\mathcal{P}$. $\mathcal{P}$ demonstrates that the inner product of this vector and the vector of committed differences is 0. With some care, we can incorporate this trick into our ZK CPU.

Namely, we modify our protocol such that (1) $\mathcal{P}$ first commits to her extended witness, (2) $\mathcal{V}$ sends its uniform challenge vector (this vector is sent in the same round where $\mathcal{V}$ sends χ), and (3) $\mathcal{V}$'s challenge vector is incorporated as a row of the instruction's topology matrix, where this row is used to constrain the instruction's checking output. In particular, this row of the matrix forces $\mathcal{P}$ to prove that the random linear combination of equality gate difference wires are each equal to zero. With this change, each instruction can use an arbitrary number of equality gates.

The crucial observation is: the above trick can be viewed as a row in the topology matrix that needs to be specified by $\mathcal{V}$. In particular, this row does not affect $\mathcal{P}$ to commit the extended witness since the extended witness is independent of $\mathcal{V}$'s uniform vector. We remark that this row *must* be specified after $\mathcal{P}$ commits the extended witness to maintain soundness. Nevertheless, $\mathcal{V}$ can specify it with the step where he sends χ to compress topology matrices to topology vectors. We note that this row can be embedded into the checking output. I.e., the checking output is the uniform linear combination of all wires that must be 0s.

6.2 Support for `LOAD` and `STORE` Gates

So far, our machine's persistent state is stored in only m registers. Of course, it would be desirable to allow instructions to access a large main memory (supporting any poly(λ) number of memory cells). We show how to implement `LOAD` and `STORE` gates that achieve memory access while keeping the number of registers m constant.

In short, to support ZK RAM, it suffices that $\mathcal{P}$ provide outputs from `LOAD` and `STORE` gates as part of her extended witness, then *prove* that these gate outputs are consistent with the semantics of a read-write array. Our insight is that these consistency checks only require that our machine maintain a constant number (five) of registers.

Setting aside our ZK CPU for a moment, recent work [42] shows that ZK RAM can be implemented by (1) maintaining a vector of all values written to RAM (tagged with appropriate timing metadata), (2) maintaining a vector of all values read from RAM (tagged with appropriate timing metadata), (3) requiring that $\mathcal{P}$ prove the above two vectors are permutations of one another, and (4) for each read, proving the accessed timing metadata value is in the past. Step (4) is achieved by a ZK ROM, which similarly can be implemented by proving two vectors are permutations of one another. Thus, the full RAM reduces to two permutation checks. To prove two vectors $\boldsymbol{a}, \boldsymbol{b}$ are related by a permutation, it is standard for $\mathcal{V}$ to issue a uniform challenge β, and then $\mathcal{P}$ shows that $\prod_{i\in[n]}(a_i - \beta) = \prod_{i\in[n]}(b_i - \beta)$.

Returning to our ZK CPU, we observe that for each permutation proof we can use two registers to accumulate the above two products; once all instructions are complete, $\mathcal{P}$ proves these two registers are equal. [42]'s RAM also requires a global clock variable, and we can support this with another register that is initialized to 0 and incremented on each RAM access. Therefore, we can compile each `LOAD`/`STORE` gate into a constant number of `INPUT`/`ADD`/`MULT` gates by maintaining five registers that jointly store the clock and partial products of the permutation checks.

One small caveat is that the ZK RAM's soundness relies on the fact that $\mathcal{P}$ cannot guess β. However, in our presented ZK CPU protocol, $\mathcal{P}$ must commit all inputs $\boldsymbol{i}$ and multiplication tuples $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ at the same time. But per the above discussion, some multiplication gates will depend on β, so $\mathcal{P}$ does not even *know* $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ until after β is chosen. This problem is straightforwardly fixed by introducing two extra protocol rounds.

Namely, (1) $\mathcal{P}$ commits to its input $\boldsymbol{i}$, (2) $\mathcal{V}$ sends β, and then (3) $\mathcal{P}$ computes and commits to $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$. This change is sound because the input $\boldsymbol{i}$ determines the entire instruction's computation, and $\boldsymbol{i}$ must be independent of β. It is possible to omit the extra two rounds by applying Fiat-Shamir [19]. Note that the combination of our tight ZK CPU with ZK RAM interestingly hides from $\mathcal{V}$ the number of RAM accesses.

[3]The circuit in $\Pi_{\mathsf{CPZK\text{-}ROM}}$ is a $O(n)$-degree polynomial, but the cost can be reduced since it computes products. See Lemma 3 and [42].

7 Evaluation

Our implementation. Using VOLE-based ZK, we implemented $\Pi_{\mathsf{CPZK\text{-}UROM}}$ (see the full verison [44]) and Π_{ZKCPU} (see, again, the full verison [44]). In particular, we instantiated $\mathcal{F}_{\mathsf{CPZK}}$ (see Figure 2) and $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ (see Figure 3) via VOLE-based ZK. VOLE-based $\mathcal{F}_{\mathsf{CPZK}}$ (QuickSilver [41]) is implemented as part of the EMP Toolkit [39], and VOLE-based $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ [42] is open-sourced[4]. We used their implementations in an (almost) black-box manner. Following these implementations, we use the prime field $\mathbb{F}_{2^{61}-1}$.

Baseline implementation. We compare our implementation to the prior state-of-the-art non-tight ZK CPU, Batchman [43]. Their implementation is open-sourced[5]. It is also a VOLE-based ZK protocol over $\mathbb{F}_{2^{61}-1}$.

Code availability. Our implementation is open-sourced and available at https://github.com/gconeice/tight-vole-zk-cpu.

Experiment setup. Unless otherwise specified, following our baseline [43], all our experiments were executed over two AWS EC2 `m5.2xlarge` machines[6] that respectively implemented $\mathcal{P}$ and $\mathcal{V}$. Each party ran single-threaded. We configured different network bandwidth settings, varying from a WAN-like 100Mbps connection to a LAN-like 1Gbps connection, via the Linux `tc` command.

Benchmarks. Our experiments used randomly generated circuits as instructions. Given a number of MULT gates, we generated gates uniformly until we reached the specified number of MULT. Our random circuits use the last input as the first register output. For each i-th instruction, the checking output is set as the first input minus i. I.e., our benchmark allows $\mathcal{P}$ to select each instruction. *Our $\mathcal{P}$ chooses each next instruction uniformly at random.* We acknowledge that this benchmark is contrived. It is used to evaluate performance only. Our implementation includes sufficient expressivity to handle a non-contrived instruction set.

We consider the following distributions of sizes of B instructions to instantiate a ZK CPU:

- *Balanced*: Each of the B instructions are of same size. This distribution is more suitable for prior non-tight ZK CPUs. Additionally, the rounding optimization of our tight ZK CPU is effective for this distribution.
- *Unbalanced*: One instruction is much bigger than the others (which are each of the same size).
- *Varied*: All sizes are distributed evenly. E.g., consider an instruction set having sizes $\{10, 20, 30, \cdots\}$.

Metrics. We report the following metrics:

- *Time*: We measured end-to-end proof execution time.
- *Communication*: We tested the overall communication.
- *Hertz Rate*: We calculated the hertz rate of a ZK CPU defined by $\frac{\#step}{time}$. This is mainly used to compare with prior non-tight ZK CPUs, i.e., the Batchman [43].
- *Multiplication Gates Per Second (MGPS)*: We calculated the MGPS defined by $\frac{\#multiplication}{time}$. This metric is only meaningful for a tight ZK CPU since all executed multiplications are useful. In a non-tight ZK CPU, some multiplications are used as padding.
- *Communication Per Multiplication (CPM)*: We calculated the CPM defined by $\frac{communication}{\#multiplication}$.

[4]Available at https://github.com/gconeice/improved-zk-ram.
[5]Available at https://github.com/gconeice/stacking-vole-zk.
[6]Intel Xeon Platinum 8175 CPU @ 3.10GHz, 8 vCPUs, 32GiB Memory, 10Gbps Network

B	m	Distribution	MGPS (#Multi./s)			CPM
			100 Mbps	500 Mbps	1 Gbps	Byte/#Multi.
10	5	Balanced	111 K	330 K	442 K	102
	1		109 K	334 K	438 K	102
50	10	Balanced	107 K	323 K	432 K	102
	20		108 K	342 K	459 K	102
		Balanced	109 K	346 K	458 K	102
100	20	Unbalanced	110 K	337 K	467 K	102
		Varied	109 K	340 K	460 K	102

Figure 7: The multiplication gates per second (MGPS) and communication per multiplication (CPM) of our ZK CPU. Recall that B denotes the number of instructions and m denotes the number of registers.

Protocol	Network Bandwidth			Comm./Step
	100 Mbps	500 Mbps	1 Gbps	
Batchman [43]	1.5 KHz	5.4 KHz	8.0 KHz	7.3 KB
Ours (Balanced)	0.6 KHz	2.7 KHz	3.7 KHz	12.7 KB
	0.56×	**0.51×**	**0.46×**	
Ours (Balanced)	1.7 KHz	5.9 KHz	8.5 KHz	6.3 KB
Rounding Opt.	**1.13×**	**1.11×**	**1.05×**	
Ours (Unbalanced)	10.6 KHz	32.5 KHz	43.8 KHz	1.0 KB
	6.90×	**6.07×**	**5.45×**	

Figure 8: Comparison with Batchman **[43].** We loaded each ZK CPU with 50 instructions and tested a 500K step execution. For the non-tight ZK CPU based on Batchman, each instruction has 125 multiplications. For our tight ZK CPU, we tested (1) balanced instructions where each has 125 multiplications and (2) unbalanced instructions where only one has 125 multiplications and others each has 5 multiplications. We report the hertz rate.

MGPS and CPM of our ZK CPU. We loaded our ZK CPU with different B and m and considered different distributions of the sizes of B instructions. In particular, we considered (1) each instruction with 100 multiplications for the balanced distribution, (2) one instruction with 100 multiplications and others each with 5 multiplications for the unbalanced distribution, and (3) i-th instruction with $10 \cdot i$ multiplications for the varied distribution. We tested our ZK CPU with each configuration by executing it over a large enough number of steps to amortize the cost of generating VOLE correlations. Figure 7 tabulates the results. It shows that our ZK CPU's speed depends mainly on network bandwidth, which aligns with our asymptotic analysis. In particular, it is (almost) *independent* of B, m, and on how instructions are distributed.

Comparison with Batchman *[43].* We compare our tight ZK CPU with prior state-of-the-art non-tight ZK CPU (i.e., Batchman). More precisely, Batchman implements batched ZK disjunctions, which can be viewed as a special ZK CPU with no registers.

The two ZK CPUs were each loaded with 50 instructions. We considered the balanced (with/without our rounding optimization)

Protocol	Network Bandwidth			Comm./Step
	100 Mbps	500 Mbps	1 Gbps	
Batchman [43]	0.2 KHz	0.8 KHz	1.1 KHz	52.1 KB
Ours	4.0 KHz	12.6 KHz	17.1 KHz	2.8 KB
	18.58×	**16.33×**	**14.96×**	

Figure 9: Comparison with Batchman **[43] with more biased unbalanced instructions.** We loaded each ZK CPU with 50 instructions and tested an execution with 500K steps. For the regular ZK CPU based on Batchman, each instruction has 1000 multiplications. We tested our ZK CPU with an unbalanced instruction set, where one instruction has 1000 multiplications and the others each have 5 multiplications. We report the hertz rate. We note that these experiments were performed with two AWS EC2 `m5.8xlarge` machines because of Batchman's larger memory requirement.

Protocol	Network Bandwidth, Total Size			Total Comm.
	100 Mbps 15.4 M	500 Mbps 15.4 M	1 Gbps 15.3 M	
QuickSilver [41]	21.2 s	6.6 s	5.1 s	226 MB
Ours	139.1 s	44.4 s	31.5 s	1484 MB
	6.56×	**6.72×**	**6.22×**	**6.56×**

Figure 10: Comparison with the setting where the execution path is public. We loaded our ZK CPU with 50 instructions and ran it for 50K steps. Each i-th instruction had $10 \cdot i$ multiplications.

and unbalanced distributions. We tested the ZK CPUs by executing 500K steps.

Figure 8 tabulates the results. It shows that our tight ZK CPU is slower than Batchman if we consider a balanced instruction set. This is due to overhead we introduce in our tight ZK CPU to ensure privacy, which is redundant when instructions are of the same size. Nevertheless, our tight ZK CPU is only slower by $\approx 2\times$, mainly coming from the $\approx 2\times$ overhead in communication. By turning on our rounding optimization, our ZK CPU performs comparably to (or even faster than) Batchman. This is because of our refined topology matrices. Note that refined topology matrices can also optimize Batchman. When considering an unbalanced instruction set, our tight ZK CPU improves over Batchman by $\approx$ 5-7×, depending on the network. We remark that even with more bandwidth, our runtime would not converge to Batchman – we additionally save constant-factor computation. The decrease in our relative improvement comes from the streamlining nature. Our ZK CPU communicates only $\approx$ 1KB per step.

Our speedup becomes more significant when considering instructions with larger differences in size; see Figure 9.

Comparison with insecure execution path. We compare our ZK CPU with an "insecure" execution where $\mathcal{P}$ and $\mathcal{V}$ agree on a public execution path. Namely, we constructed a single plaintext circuit encoding an execution path and then ran the QuickSilver protocol (which achieves $\mathcal{F}_{\mathsf{CPZK}}$) on that circuit. Of course, a ZK CPU will use more resources than such a circuit, since a ZK CPU provides a stronger privacy guarantee. These experiments illustrate the performance gap between our ZK CPU and the informal "lower bound". Figure 10 tabulates the results. Our ZK CPU has a $\approx 6\times$ overhead in communication (as a constant). Further optimizing this constant is an interesting direction.

Rounding optimization. Recall that our ZK CPU supports an optimization such that if the size of each instruction is a multiple of ε, several contributing costs are reduced by factor ε. To evaluate the effectiveness of this optimization, we loaded our ZK CPU with 50 balanced instructions. By varying the size of each instruction and letting the ZK CPU execute 6.4M multiplications, we deployed the rounding optimization with different ε. Our experiments show that, when $\varepsilon \geq 16$, the rounding optimization can speed up our ZK CPU by $\approx 2\times$, independent of the network bandwidth. The improvement comes from savings in communication, matching asymptotic.

Microbenchmarks. The full version [44] includes fine-grained (i.e., decomposed) end-to-end time.

Acknowledgments

This work is supported in part by Visa research award, Cisco research award, and NSF awards CNS-2246353, CNS-2246354, and CCF-2217070. This material is also based upon work supported in part by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 2087–2104. https://doi.org/10.1145/3133956.3134104

[2] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In *CRYPTO 2021, Part IV (LNCS, Vol. 12828)*, Tal Malkin and Chris Peikert (Eds.). Springer, Heidelberg, Virtual Event, 92–122. https://doi.org/10.1007/978-3-030-84259-8_4

[3] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 459–474. https://doi.org/10.1109/SP.2014.36

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO 2013, Part II (LNCS, Vol. 8043)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6

[5] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *CRYPTO 2014, Part II (LNCS, Vol. 8617)*, Juan A. Garay and Rosario Gennaro (Eds.). Springer, Heidelberg, 276–294. https://doi.org/10.1007/978-3-662-44381-1_16

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 781–796.

[7] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT 2011 (LNCS, Vol. 6632)*, Kenneth G. Paterson (Ed.). Springer, Heidelberg, 169–188. https://doi.org/10.1007/978-3-642-20465-4_11

[8] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 315–334. https://doi.org/10.1109/SP.2018.00020

[9] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*. IEEE Computer Society Press, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[10] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*. ACM Press, 494–503. https://doi.org/10.1145/509907.509980

[11] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *EUROCRYPT 2020, Part I (LNCS, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26

[12] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. 2023. SublonK: Sublinear Prover PlonK. Cryptology ePrint Archive, Paper 2023/902. https://eprint.iacr.org/2023/902

[13] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. 1994. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *CRYPTO'94 (LNCS, Vol. 839)*, Yvo Desmedt (Ed.). Springer, Heidelberg, 174–187. https://doi.org/10.1007/3-540-48658-5_19

[14] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. 2021. Limbo: Efficient Zero-knowledge MPCitH-based Arguments. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 3022–3036. https://doi.org/10.1145/3460120.3484595

[15] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. 2022. Efficient Proof of RAM Programs from Any Public-Coin Zero-Knowledge System. In *Security and Cryptography for Networks*, Clemente Galdi and Stanislaw Jarecki (Eds.). Springer International Publishing, Cham, 615–638.

[16] Zijing Di, Lucas Xia, Wilson Nguyen, and Nirvan Tyagi. 2023. MUXProofs: Succinct Arguments for Machine Computation from Tuple Lookups. Cryptology ePrint Archive, Paper 2023/974. https://eprint.iacr.org/2023/974

[17] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2021. Line-Point Zero Knowledge and Its Applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 199)*, Stefano Tessaro (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:24. https://doi.org/10.4230/LIPIcs.ITC.2021.5

[18] Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. 2021. Zero Knowledge Static Program Analysis. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 2951–2967. https://doi.org/10.1145/3460120.3484795

[19] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO'86 (LNCS, Vol. 263)*, Andrew M. Odlyzko (Ed.). Springer, Heidelberg, 186–194. https://doi.org/10.1007/3-540-47721-7_12

[20] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. 2021. Constant-Overhead Zero-Knowledge for RAM Programs. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 178–191. https://doi.org/10.1145/3460120.3484800

[21] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. 2022. Stacking Sigmas: A Framework to Compose Σ-Protocols for Disjunctions. In *EUROCRYPT 2022, Part II (LNCS, Vol. 13276)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, Heidelberg, 458–487. https://doi.org/10.1007/978-3-031-07085-3_16

[22] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. 2023. Dora: Processor Expressiveness is (Nearly) Free in Zero-Knowledge for RAM Programs. Cryptology ePrint Archive, Paper 2023/1749. https://eprint.iacr.org/2023/1749

[23] Aarushi Goel, Mathias Hall-Andersen, Gabriel Kaptchuk, and Nicholas Spooner. 2023. Speed-Stacking: Fast Sublinear Zero-Knowledge Proofs for Disjunctions. In *EUROCRYPT 2023, Part II (LNCS, Vol. 14005)*, Carmit Hazay and Martijn Stam (Eds.). Springer, Heidelberg, 347–378. https://doi.org/10.1007/978-3-031-30617-4_12

[24] S Goldwasser, S Micali, and C Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, USA) *(STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/22145.22178

[25] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *EUROCRYPT 2020, Part III (LNCS, Vol. 12107)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 569–598. https://doi.org/10.1007/978-3-030-45727-3_19

[26] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. 2021. Zero Knowledge for Everything and Everyone: Fast ZK Processor with Cached ORAM for ANSI C Programs. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1538–1556. https://doi.org/10.1109/SP40001.2021.00089

[27] Wenqing Hu, Tianyi Liu, Ye Zhang, Yuncong Zhang, and Zhenfei Zhang. 2024. Parallel Zero-knowledge Virtual Machine. Cryptology ePrint Archive, Paper 2024/387. https://eprint.iacr.org/2024/387

[28] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-Knowledge from Secure Multiparty Computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '07)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/1250790.1250794

[29] Vladimir Kolesnikov. 2018. Free IF: How to Omit Inactive Branches and Implement S-Universal Garbled Circuit (Almost) for Free. In *ASIACRYPT 2018, Part III (LNCS, Vol. 11274)*, Thomas Peyrin and Steven Galbraith (Eds.). Springer, Heidelberg, 34–58. https://doi.org/10.1007/978-3-030-03332-3_2

[30] Abhiram Kothapalli and Srinath Setty. 2022. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758. https://eprint.iacr.org/2022/1758.

[31] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *CRYPTO 2022, Part IV (LNCS, Vol. 13510)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, Heidelberg, 359–388. https://doi.org/10.1007/978-3-031-15985-5_13

[32] Abhiram Kothapalli and Srinath T. V. Setty. 2024. HyperNova: Recursive Arguments for Customizable Constraint Systems. In *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part X (Lecture Notes in Computer Science, Vol. 14929)*, Leonid Reyzin and Douglas Stebila (Eds.). Springer, 345–379. https://doi.org/10.1007/978-3-031-68403-6_11

[33] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang. 2024. Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 39–39. https://doi.org/10.1109/SP54263.2024.00035

[34] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 2968–2985. https://doi.org/10.1145/3460120.3485379

[35] Ning Luo, Timos Antonopoulos, William R. Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. 2022. Proving UNSAT in Zero Knowledge. In *ACM CCS 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, 2203–2217. https://doi.org/10.1145/3548606.3559373

[36] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2111–2128. https://doi.org/10.1145/3319535.3339817

[37] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO 2012 (LNCS, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, Heidelberg, 681–700. https://doi.org/10.1007/978-3-642-32009-5_40

[38] David Pointcheval and Jacques Stern. 2000. Security Arguments for Digital Signatures and Blind Signatures. *Journal of Cryptology* 13, 3 (June 2000), 361–396. https://doi.org/10.1007/s001450010003

[39] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[40] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In *USENIX Security 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 501–518.

[41] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 2986–3001. https://doi.org/10.1145/3460120.3484556

[42] Yibin Yang and David Heath. 2024. Two Shuffles Make a RAM: Improved Constant Overhead Zero Knowledge RAM. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1435–1452. https://www.usenix.org/conference/usenixsecurity24/presentation/yang-yibin

[43] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. 2023. Batchman and Robin: Batched and Non-Batched Branching for Interactive ZK. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1452–1466. https://doi.org/10.1145/3576915.3623169

[44] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. 2024. Tight ZK CPU: Batched ZK Branching with Cost Proportional to Evaluated Instruction. Cryptology ePrint Archive, Paper 2024/456. https://eprint.iacr.org/2024/456

[45] Yibin Yang, David Heath, Vladimir Kolesnikov, and David Devecsery. 2022. EZEE: Epoch Parallel Zero Knowledge for ANSI C. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*. IEEE, Genoa, Italy, 109–123. https://doi.org/10.1109/EuroSP53844.2022.00015

[46] Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. 2023. Towards Generic MPC Compilers via Variable Instruction Set Architectures (VISAs). In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (, Copenhagen, Denmark,) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2516–2530. https://doi.org/10.1145/3576915.3616664

[47] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster Verifiable RAM with Program-Independent Preprocessing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 908–925. https://doi.org/10.1109/SP.2018.00013