Tight ZK CPU: Batched ZK Branching with Cost Proportional to Evaluated Instruction

Yibin Yang[*] David Heath[†] Carmit Hazay[‡] Vladimir Kolesnikov[§]
Muthuramakrishnan Venkitasubramaniam[¶]

March 18, 2024

Abstract

We explore Zero-Knowledge proofs (ZKP) 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 ZK CPU 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* (ZK UROM), may be of an independent interest.

[*]Georgia Institute of Technology, yyang811@gatech.edu.
[†]University of Illinois Urbana-Champaign, daheath@illinois.edu.
[‡]Bar-Ilan University and Ligero Inc., Carmit.Hazay@biu.ac.il.
[§]Georgia Institute of Technology, kolesnikov@gatech.edu.
[¶]Ligero Inc., muthu@ligero-inc.com.

Contents

1 Introduction **1**
- 1.1 Our Focus: Pay for the Active Branch . . . 2
- 1.2 Our Contribution . . . 3
- 1.3 Intuition of our Construction . . . 3
- 1.4 Related Work . . . 4

2 Preliminaries **5**
- 2.1 Notation . . . 5
- 2.2 Security Model . . . 5
- 2.3 Commit-and-Prove Zero-Knowledge . . . 6
- 2.4 Zero-Knowledge Read-Only Memory . . . 7
- 2.5 Zero-Knowledge Proofs via Topology Matrices . . . 9

3 Our Target Functionality: $\mathcal{F}_{\mathsf{ZKCPU}}$ **10**

4 Technical Overview **11**
- 4.1 Boundary Strings and Helper Notation . . . 11
- 4.2 More Powerful Topology Matrices . . . 13
- 4.3 Reducing a Tight ZK CPU to a ZK UROM . . . 13
  - 4.3.1 Special Case: No Registers . . . 13
  - 4.3.2 Handling Constant 1 . . . 15
  - 4.3.3 Supporting Registers . . . 16
  - 4.3.4 Committing to the Topology Vector . . . 17
- 4.4 ZK Non-Zero-End Unbalanced ROM . . . 18
  - 4.4.1 Using ZK UROM with Topology Vectors . . . 19

5 Formalization **20**
- 5.1 Ideal ZK Non-Zero-End UROM: $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ . . . 20
- 5.2 Our Protocols: $\Pi_{\mathsf{CPZK\text{-}UROM}}$ and Π_{ZKCPU} . . . 20
- 5.3 Optimization and Cost Analysis . . . 26

6 Support for Advanced Operations **28**
- 6.1 Equality Gates . . . 28
- 6.2 Support for `LOAD` and `STORE` Gates . . . 29

7 Evaluation **29**

A Deferred Complete Proofs **39**
- A.1 Complete Proof of Theorem 1 . . . 39
- A.2 Complete Proof of Theorem 2 . . . 42

B Fine-grained Cost Analysis **43**
- B.1 Cost Analysis for $\Pi_{\mathsf{CPZK\text{-}UROM}}$ in $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid . . . 43
- B.2 Cost Analysis for Π_{ZKCPU} in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid . . . 44

1 Introduction

Zero-Knowledge (ZK) proofs (ZKP) [GMR85] 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 [BCG+14], private programming analysis [FDNZ21, LAH+22], private bug-bounty [HYDK21, YHKD22], privacy-preserving machine learning [LXZ21, WYX+21], 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 in practice – using a high-level language, such as `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 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 [YPHK23].

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 that a state-of-the-art ZK RAM [YH23] 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, 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 ([BMRS21, GGHAK22, GHAK23, GHAKS23, HK20, Kol18, YHH+23]) aims to avoid paying for inactive clauses in a disjunction. [HK20] 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 (largest) branch, but still requires processing all branches. Very recent work [GHAK23, YHH+23] shows how to limit both communication and computation to that of the single largest branch for our setting, where instructions are executed repeatedly.

To summarize, the previous state of the art *pays for the largest branch.* In contrast:

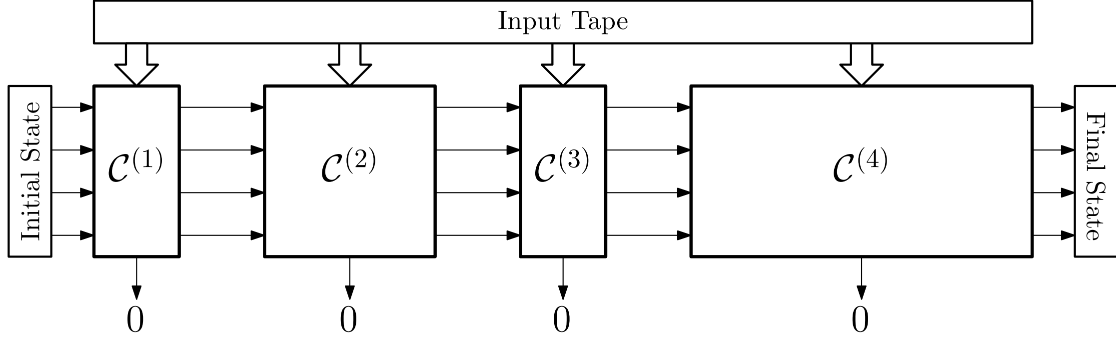


Figure 1: Example ZK CPU execution. $\mathcal{P}$ and $\mathcal{V}$ agree on B public (sub)circuits $I = \{\mathcal{C}_1, \ldots, \mathcal{C}_B\}$. $\mathcal{P}$ demonstrates to $\mathcal{V}$ that an initial state evaluates to a final state via a *private* circuit $\mathcal{C} \triangleq \mathcal{C}^{(4)} \circ \cdots \circ \mathcal{C}^{(1)}$, where each $\mathcal{C}^{(i \in [4])} \in I$. $\mathcal{V}$ learns the size of $\mathcal{C}$, but he does not learn the identity of any particular subcircuit. 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 $\mathcal{C}_1$ when the first register is 1. See Section 3 for formal details.

1.1 Our Focus: Pay for the Active Branch

We are motivated by scenarios where instructions (or branches) are of significantly different sizes, potentially differing in size by orders of magnitude. In such cases, it is unacceptable to incur the cost of the largest 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.

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 to cheaply handle arbitrarily large (and arbitrarily wide!) instructions without incurring the overhead of handling additional registers.

Tight ZK CPU emulation. We mostly adhere to the ZK CPU notation and vocabulary. We chose 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., [BCG+13, FKL+21, HYDK21, YHH+23]).

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.

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 (see, e.g., [HYDK21]) 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.

1.2 Our Contribution

We motivate and formalize the notion of a *tight* ZK CPU, where cost (both computation and communication, for each party) of each step depends only on the instruction taken, even when instructions are of varying sizes. Informally, the proof just follows the (*private*) plaintext evaluation path inside ZK.

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 proof.

We instantiate a concretely efficient tight ZK CPU. Our protocol is abstracted in the general commit-and-prove paradigm. We instantiate our techniques in the interactive ZK setting by building on Vector OLE (VOLE) [BCGI18]. Section 7 reports on the performance of the resulting protocol. Compared with prior ZK CPU emulation, our protocol achieves cost that is only a constant factor (6–7×) higher than if the execution path were entirely revealed to $\mathcal{V}$, as our protocol scales only linearly with the number of multiplication gates along the program execution path.

In concrete terms, our constructions are lean, and outperform state of the art Batchman [YHH+23] (a non-tight ZK CPU) both in computation and communication commensurately with branch size variation (see Section 7).

Because our techniques rely only on commit-and-prove, they are compatible with a variety of ZK proof systems, including non-interactive systems (our techniques are constant-round with public-coin challenges, and are compatible with Fiat-Shamir [FS87], leading to a NIZK CPU).

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 "oblivious 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 [YHH+23] 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 witness.

If $\mathcal{P}$ is honest, this vector encodes a valid execution path. $\mathcal{P}$ proves that 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 primitive gates.

In our approach, we allow instructions of different sizes. Thus, while our $\mathcal{P}$ 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.

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 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. Based on the above intuition, our unbalanced ZK ROM manages (loads 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 [BCG+13] and subsequently studied by [BCTV14a, BCTV14b, FKL+21, GHAK23, HYDK21, YHKD22].

Early ZK work [CDS94] 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, then multiplexes the results. While this works, it is expensive. [HK20] – building on the MPC result of [Kol18] – demonstrated feasibility of paying (in ZK proof size) for only one branch. The [HK20] technique "reuses cryptographic material" of the active branch to evaluate (to garbage and privately discard) inactive branches. This sparked a rich line of work [BMRS21, GGHAK22, GHAK23, GHAKS23, HK20, Kol18, YHH+23] that continues to reduce the costs of ZK disjunctions.

Very recent work [GHAK23, YHH+23] further improved handling of disjunctions by showing how to improve not just communication, but also *computation*. This task is harder 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, too) proportionally only to the single largest clause of the disjunction, Our work extends and crucially builds on the approach of [YHH+23], and our extension enables paying only for the *active* branch. Sections 1.3 and 2.5 summarize [YHH+23] and the novel techniques needed for our result. Neither [YHH+23], nor [GHAK23] address disjunctions of clauses of varying sizes.

Efficient ZK ROM and RAM are essential to CPU-emulation ZK. We integrate recent ZK ROM [YH23]. We also build on it to design a novel basic primitive *unbalanced ZK ROM*, one capable of retrieving variable-size entries in a batch query. We achieve this by extending randomized hashes of [YHH+23] to vectors of differing lengths, and ultimately use to execute variable-size instructions.

Non-tight ZK CPU constructions have been studied in the SNARK setting (e.g., [ZGK+18, KS22, CGG+23, DXNT23]). Recently, Hu et al. [HLZ+24] proposed a tight SNARK CPU construction, where $\mathcal{P}$ leaks the number of executed instances of each CPU instruction. ZK property can be added by padding, but this loses tightness.

2 Preliminaries

2.1 Notation

- λ is the statistical security parameter (e.g., 40).
- 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 $\mathbb{F}$. 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 $\boldsymbol{a}$. 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 $a[i]$) 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 [Can01]. We prove the security of our protocol in the presence of a *malicious, static* adversary. For simplicity, we omit standard UC session IDs.

2.3 Commit-and-Prove Zero-Knowledge

Our protocol is generic, as it builds on the standard *Commit-and-Prove* Zero-Knowledge (CPZK) functionality [CLOS02]; see Figure 3. In CPZK, $\mathcal{P}$ commits to values (in $\mathbb{F}$) and then proves that evaluating a particular circuit on the committed values yields a vector of 0. We use $\mathsf{com}(x)$ to denote a cryptographic commitment to $x \in \mathbb{F}$, and we extend this notation to vectors naturally (e.g., $\mathsf{com}(\boldsymbol{x})$).

There are various ways to instantiate $\mathcal{F}_{\mathsf{CPZK}}$ (e.g., [AHIV17, BMRS21, BBB+18, CHM+20, DOT21, DIO21, IKOS07, MBKM19, YSWW21]). To concretely evaluate our approach, we instantiate our protocol via VOLE-based ZK (e.g., [BMRS21, DIO21, YSWW21]), a ZK paradigm notable for its fast end-to-end proof times. In VOLE-based ZK, commitments are information-theoretic MACs (IT-MACs) [BDOZ11, NNOB12] over $\mathbb{F}$. IT-MAC commitments are linearly homomorphic, and the resulting proof systems have total communication and computation that scale linearly (with low constants) in $|\mathcal{C}|$. We recall VOLE-based ZK's computation/communication:

Lemma 1 (VOLE-based ZK)**.** *There exists a protocol Π_{CPZK} that UC-realizes $\mathcal{F}_{\mathsf{CPZK}}$ in the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model (Figure 2) with the following efficiency metrics:*

- `Commit` *requires* 1 *field element of communication and $\mathcal{O}(1)$ field operations.*
- `Linear` *requires* no *communication and $\mathcal{O}(k)$ field operations.*
- `Open` *requires* 2 *field elements of communication and $\mathcal{O}(1)$ field operations, with soundness error $\mathcal{O}(\frac{1}{|\mathbb{F}|})$.*
- *For circuit $\mathcal{C}$ with m outputs and $n_\times$ multiplication gates,* `Check` *requires $n_\times + m + 3$ field elements of communication and $\mathcal{O}(|\mathcal{C}|)$ field operations, with soundness error $\mathcal{O}(\frac{|\mathcal{C}|}{|\mathbb{F}|})$.*
- *Communication of* `Check` *for certain circuits can be improved at the cost of computation and soundness. In particular, for a circuit $\mathcal{C}$ that outputs m polynomials $f_1, \ldots, f_m$ where $f_{i\in[m]}$ is of degree-d_i,* `Check` *can be performed with $\max_{i\in[m]} d_i + 1$ field elements of communication and $\mathcal{O}\left((\max_{i\in[m]} d_i)^2 \cdot |\mathcal{C}|\right)$ field operations, with soundness error $\mathcal{O}\left(\frac{m+\max_{i\in[m]} d_i}{|\mathbb{F}|}\right)$. This is useful, e.g., when we want to prove that $\boldsymbol{a} \odot \boldsymbol{b} = \boldsymbol{0}$.*

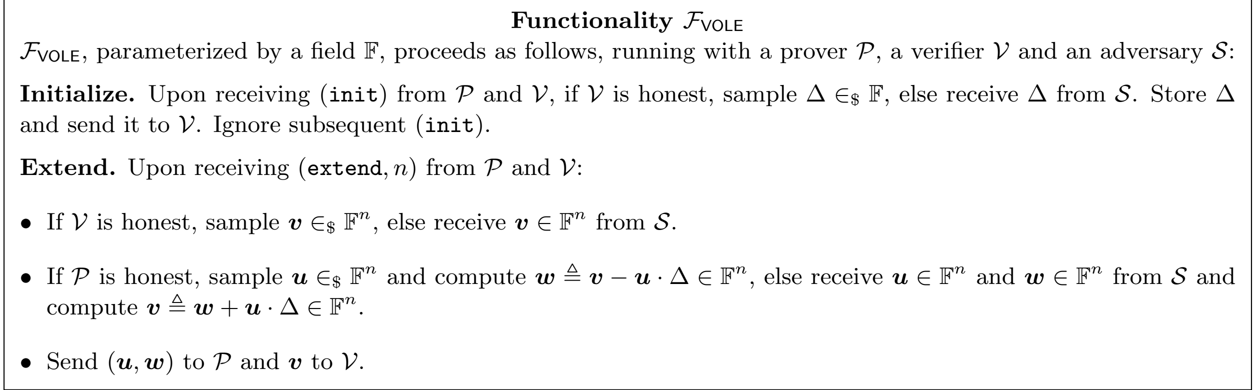
Testing vector equality. Application of *Swchartz-Zippel* lemma allows to prove equality of two committed vectors:

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

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

That is, 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 show 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, zero-end vectors of different lengths (e.g., $\boldsymbol{a} = (1, 1)$ and $\boldsymbol{b} = (1, 1, 0)$) are not handled by Lemma 2.

The equality check of Lemma 2 does extend to *non-zero-end* vectors of potentially different lengths (Corollary 1). We need this because $\mathcal{V}$ does not know the boundaries of instructions/subvectors whose equality is proven by $\mathcal{P}$.



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

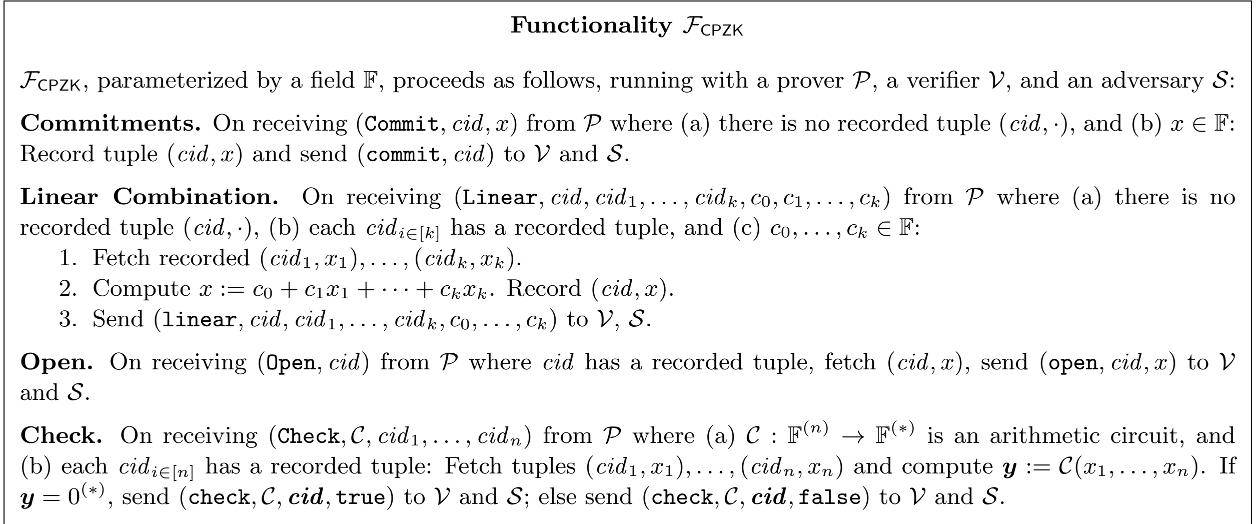
$\mathcal{F}_{\mathsf{VOLE}}$, parameterized by a field $\mathbb{F}$, proceeds as follows, running with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$:

Initialize. Upon receiving $(\texttt{init})$ from $\mathcal{P}$ and $\mathcal{V}$, if $\mathcal{V}$ is honest, sample $\Delta \in_\$ \mathbb{F}$, else receive Δ from $\mathcal{S}$. Store Δ and send it to $\mathcal{V}$. Ignore subsequent $(\texttt{init})$.

Extend. Upon receiving $(\texttt{extend}, n)$ from $\mathcal{P}$ and $\mathcal{V}$:

- If $\mathcal{V}$ is honest, sample $\boldsymbol{v} \in_\$ \mathbb{F}^n$, else receive $\boldsymbol{v} \in \mathbb{F}^n$ from $\mathcal{S}$.
- If $\mathcal{P}$ is honest, sample $\boldsymbol{u} \in_\$ \mathbb{F}^n$ and compute $\boldsymbol{w} \triangleq \boldsymbol{v} - \boldsymbol{u} \cdot \Delta \in \mathbb{F}^n$, else receive $\boldsymbol{u} \in \mathbb{F}^n$ and $\boldsymbol{w} \in \mathbb{F}^n$ from $\mathcal{S}$ and compute $\boldsymbol{v} \triangleq \boldsymbol{w} + \boldsymbol{u} \cdot \Delta \in \mathbb{F}^n$.
- Send $(\boldsymbol{u}, \boldsymbol{w})$ to $\mathcal{P}$ and $\boldsymbol{v}$ to $\mathcal{V}$.

Figure 2: The VOLE correlation functionality.



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

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

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

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

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

Figure 3: Ideal functionality for commit-and-prove ZK.

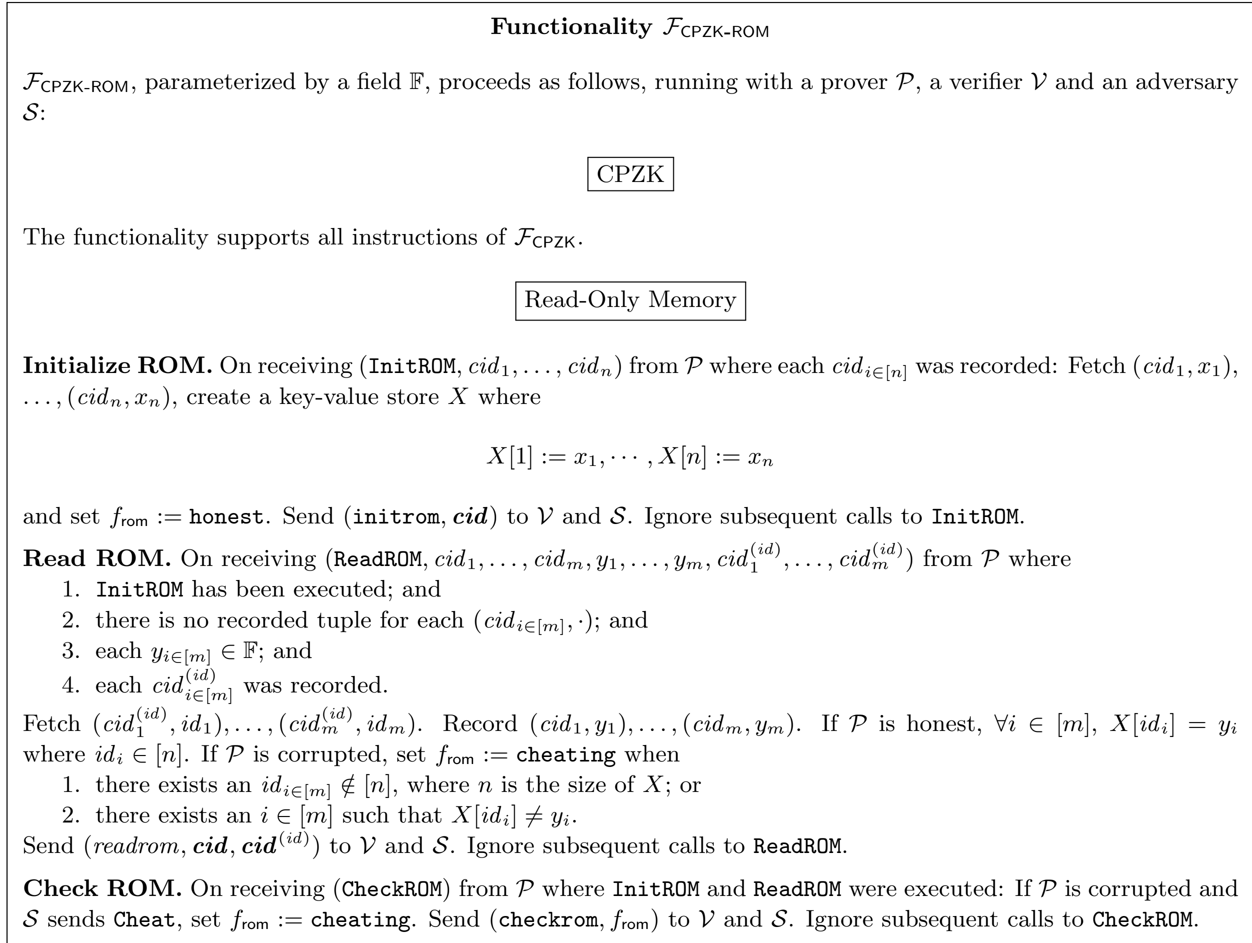
Corollary 1. *Consider vectors $\boldsymbol{a} \in \mathbb{F}^{n_a}, \boldsymbol{b} \in \mathbb{F}^{n_b}$ where $a[n_a], b[n_b] \neq 0$. If $\boldsymbol{a} \neq \boldsymbol{b}$, for $\chi \in_\$ \mathbb{F}$:*

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

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

2.4 Zero-Knowledge Read-Only Memory

We need to access ZK ROM (e.g., [DdSGOTV22, FKL+21, YH23]); in CPZK, a ROM access generates a new committed value based on a committed index. 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 $\mathsf{com}(i)$, where $i \in [n]$, $\mathcal{P}$ and $\mathcal{V}$ generate a new commitment $\mathsf{com}(x)$ where x is the i-th committed value in the K-V store. Our protocol uses a restricted (batch-read)



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

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

CPZK

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

Read-Only Memory

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

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

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

Read ROM. On receiving ($\mathtt{ReadROM}$, $cid_1, \ldots, cid_m, y_1, \ldots, y_m, cid_1^{(id)}, \ldots, cid_m^{(id)}$) from $\mathcal{P}$ where
1. $\mathtt{InitROM}$ 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 $(cid_1^{(id)}, id_1), \ldots, (cid_m^{(id)}, id_m)$. Record $(cid_1, y_1), \ldots, (cid_m, y_m)$. 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 $f_{\mathsf{rom}} := \mathtt{cheating}$ 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 ($readrom$, $\boldsymbol{cid}$, $\boldsymbol{cid}^{(id)}$) to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent calls to $\mathtt{ReadROM}$.

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

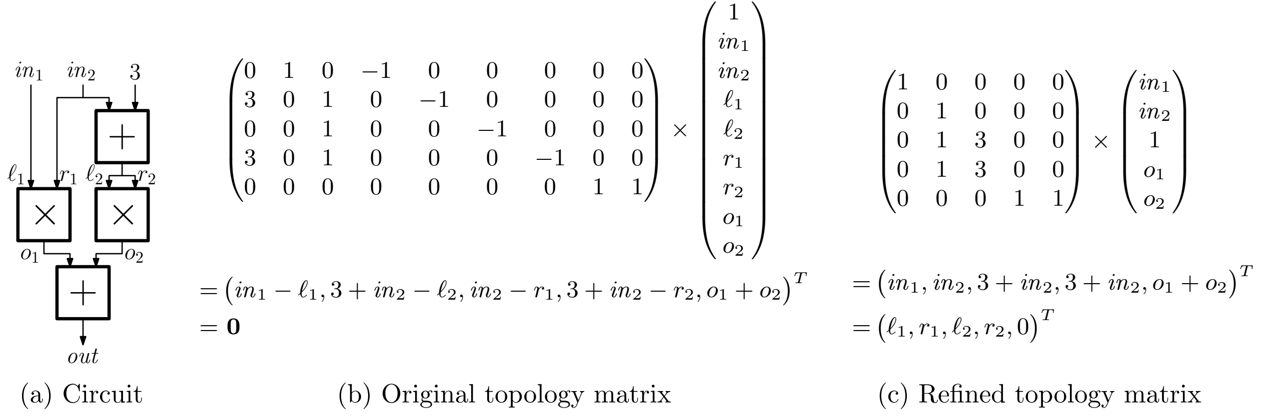
Figure 4: Ideal functionality for commit-and-prove zero-knowledge with a single read-only memory.

version of ZK ROM formalized in Figure 4. I.e., $\mathcal{P}$ is allowed a *single* $\mathtt{ReadROM}$ 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 primitive, unbalanced ROM, to load the variable-length instruction vectors).

[YH23] is a state-of-the-art realization of $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ in the $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model, as stated in Lemma 3.

Lemma 3 (ZK Read-Only Memory)**.** *Let $n, m = \mathsf{poly}(\lambda)$. There exists a protocol $\Pi_{\mathsf{CPZK\text{-}ROM}}$ that UC-emulates $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ (Figure 4) in the $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model (Figure 3) with the following efficiency metrics:*

- $\mathtt{InitROM}$ *requires $\mathcal{P}$ to only send* $\boldsymbol{cid}$ *to $\mathcal{V}$, and $n + 1$* $\mathtt{Linear}$ *hybrid calls to generate* $\mathsf{com}(0), \ldots, \mathsf{com}(n)$.
- $\mathtt{ReadROM}$ *requires $2m$* $\mathtt{Commit}$ *hybrid calls.*
- $\mathtt{CheckSet}$ *requires $2n$* $\mathtt{Commit}$ *hybrid calls, followed by $\mathcal{V}$'s sending 4 uniform elements in $\mathbb{F}$*



(a) Circuit (b) Original topology matrix (c) Refined topology matrix

Figure 5: (a) An arithmetic circuit computing $(in_1 \cdot in_2) + (in_2 + 3)^2$ and its (b) original and (c) refined topology matrix.

and a `Check` *hybrid call, where* $\mathcal{C}$ *is dominated by two products each has* $n+m-1$ *multiplication gates.*

2.5 Zero-Knowledge Proofs via Topology Matrices

Consider a circuit $\mathcal{C}$ with n_{in} inputs and $n_\times$ multiplication gates. Batchman [YHH+23] observed that ZKP for $\mathcal{C}$ can be separated into two parts: (1) multiplication gates and (2) linear constraints. Suppose that $\mathcal{P}$ commits to its input $\mathsf{com}(in_1), \ldots, \mathsf{com}(in_{n_{in}})$, and $\mathcal{P}$ also commits to values on the $3n_\times$ wires associated with $\mathcal{C}$'s $n_\times$ multiplication gates. Namely, $\mathcal{P}$ commits to $\mathsf{com}(\ell_1), \ldots, \mathsf{com}(\ell_{n_\times})$, corresponding to multiplication left input wires, to $\mathsf{com}(r_1), \ldots, \mathsf{com}(r_{n_\times})$, corresponding to right input wires, and to $\mathsf{com}(o_1), \ldots, \mathsf{com}(o_{n_\times})$, corresponding to output wires. The full vector of $\mathcal{P}$'s input and the multiplication wires is called $\mathcal{P}$'s *extended witness*.

Now, $\mathcal{P}$ first proves to $\mathcal{V}$ that $\boldsymbol{l} \odot \boldsymbol{r} = \boldsymbol{o}$, demonstrating that its extended witness satisfies multiplicative constraints. Then, $\mathcal{P}$ proves to $\mathcal{V}$ that $\boldsymbol{in}, \boldsymbol{l}, \boldsymbol{r}, \boldsymbol{o}$ indeed respect the constraints imposed by circuit $\mathcal{C}$. 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 $\mathcal{C}$, and [YHH+23] refers to $\boldsymbol{M}$ as a *topology matrix*. Namely, $\mathcal{P}$ proves the following:

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

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 $\mathcal{F}_{\mathsf{CPZK}}$ to construct a commitment to

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

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 $(1, \ldots, \chi^{2n_\times}) \times \boldsymbol{M}$ (called a *topology vector*). Thus, it suffices to check whether the inner product between the topology vector and the extended witness yields 0. Figure 5b shows an example 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) $\mathcal{C}_1, \ldots, \mathcal{C}_B$, 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 $\mathcal{C}_i$ 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 $(\texttt{Prove}, \tau, i_1, \ldots, i_\tau, \boldsymbol{in}_1, \ldots, \boldsymbol{in}_\tau)$ from $\mathcal{P}$ where (1) τ is a positive integer, (2) $i_{j \in [\tau]} \in [B]$, and (3) for each $j \in [\tau]$, $\boldsymbol{in}_j \in \mathbb{F}^{n_{in}^{(i_j)} - m}$, proceed as follows:

1. Set $\boldsymbol{st} := \boldsymbol{st}^{(0)}$ and $f := \texttt{true}$. For each $j \in [\tau]$ in order:
   (a) Let $\boldsymbol{st}' \| f' := \mathcal{C}_{i_j}(\boldsymbol{st} \| \boldsymbol{in}_j)$ where $\boldsymbol{st}' \in \mathbb{F}^m, f' \in \mathbb{F}$.
   (b) Set $\boldsymbol{st} := \boldsymbol{st}'$. If $f' \neq 0$, set $f := \texttt{false}$.
2. Let $n \triangleq n^{(i_1)} + \cdots + n^{(i_\tau)}$.
3. If $\boldsymbol{st} \neq \boldsymbol{st}^{(final)}$, set $f := \texttt{false}$.
4. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ can send $(\texttt{Cheat}, n')$ where $n' \in \mathbb{Z}^+$: Set $f := \texttt{false}$, $n := n'$.
5. Send $(\texttt{prove}, f, n)$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 6: Ideal functionality for a tight ZK CPU.

Proving batched disjunctions: Batchman [YHH+23]. The above paradigm seems useless if we only consider ZKP for a single public circuit. Indeed, it yields (constant) overhead since state-of-the-art CPZK (e.g. QuickSilver [YSWW21]) only requires committing $\boldsymbol{in}$ and $\boldsymbol{o}$. However, this paradigm becomes useful when considering a batch of disjunctions.

Batchman [YHH+23] considers B circuits $\mathcal{C}_1, \ldots, \mathcal{C}_B$ of the same size. $\mathcal{P}$ wants to repeatedly prove to $\mathcal{V}$ R times that she knows some witness $\boldsymbol{w}_{i \in [R]}$ and some index id_i such that $\mathcal{C}_{id_i}(\boldsymbol{w}_i) = 0$. Batchman can be viewed as a *non-tight* ZK CPU (with no registers). The intuition behind Batchman is that $\mathcal{P}$ commits to her extended witness for each i-th repetition of *only* $\mathcal{C}_{id_i}$. $\mathcal{V}$ then issues a uniform challenge χ to compress B topology matrices to B topology vectors. The *crucial step* is that $\mathcal{P}$ then commits to each id_i-th topology vector. Of course, $\mathcal{P}$ can cheat and commit a vector that is not a topology vector, so extra mechanisms are needed to maintain soundness, and this mechanism can be achieved by a ZK ROM (storing then loading vector's hashes). Finally, it suffices to show that the inner product between the extended witness and the topology vector is 0 for each repetition.

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

In this section, we formalize the functionality of a tight ZK CPU achieved by our protocol. A CPU over some field $\mathbb{F}$ can be viewed as an object where:

1. $B \in \mathbb{Z}^+$ denotes the number of instructions.
2. $m \in \mathbb{Z}^+$ denotes the number of registers.
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 $\mathcal{C} : \mathbb{F}^n \to \mathbb{F}^{m+1}$ where $n \geq m$. In particular, we consider standard fan-in 2 circuits over $\mathbb{F}$ with addition and multiplication gates. We call an instruction $\mathcal{C} : \mathbb{F}^n \to \mathbb{F}^{m+1}$ a m-instruction. The first m of $\mathcal{C}$'s output wires capture the updated CPU registers, and the last wire is a* checking *output (0 in a valid execution).*

In a tight CPU execution, $\mathcal{P}$ and $\mathcal{V}$ agree on the initial and final state of the m registers (the initial and final state, respectively). $\mathcal{P}$ proves that she can "execute" from the initial state to the final state by one-by-one applying (potentially repeatedly) instructions. We formalize this functionality in Figure 6 with the following remarks:

1. For each instruction with $n_\times$ multiplications, n_{in} inputs, and m registers, w.l.o.g., we assume $n_{in} - m = n_\times + m + 2$. That is, we assume each instruction has a number of inputs that is similar to its number of multiplications. This similarity simplifies handling, and it can be enforced by simply padding the instruction with dummy input wires/multiplication gates.

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 CPU steps τ.

3. In Figure 6, $\mathcal{P}$ freely selects which instructions to execute. Of course, it is more realistic that $\mathcal{P}$'s chosen instructions should be constrained by the current register state. For example, a program counter variable might dictate which instruction runs next. Such constraints can be captured by each instruction's checking output wire, which must hold 0 for the proof to succeed. As a simple example, an instruction's checking output could be defined by subtracting one from that instruction's first register, which would allow $\mathcal{P}$ to choose this instruction only if the first register were set to 1.

4. This model of computation only allows limited persistent program state (i.e., up to m registers) to be pass from instruction to instruction. Perhaps surprisingly, we show that by introducing 5 special registers and 2 extra rounds, our protocol can support a large (poly-size in λ) read-write random access memory (see Section 6.2).

4 Technical Overview

In this section, we present our technical ideas at a level sufficient to understand our contributions. Full details are postponed to subsequent sections. 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 6) are outlined in Figure 7.

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

Figure 7: The outline of our presentation.

4.1 Boundary Strings and Helper Notation

Recall from Section 1.3 that our approach leverages a 0-1 vector of field elements that we refer to as a *boundary string*. This section clarifies the definition of boundary strings, and it introduces helpful notation for reasoning about such strings.

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

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

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:

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

Expanding random challenges. In our protocol, $\mathcal{V}$ will issue random challenges, and these challenges 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 $i \in [n-1]$, let $s_{i+1} \triangleq s_i(1-p_i) + \chi^i p_i$. That is,

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

   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, $s_{i+1} \triangleq \gamma s_i(1-p_i) + p_i$. That is,

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

   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 $n-1$ multiplication gates.

4.2 More Powerful Topology Matrices

Our protocol uses topology matrices (see Section 2.5).

We first introduce a $\approx 2\times$ optimization to the topology matrix/vector of [YHH+23] (see Figure 5c). Note that part of the [YHH+23] topology matrix is dedicated to explicitly connecting inputs and (MULT) gate output wires to gate input wires *in a fixed order* (e.g., in Figure 5b, this order is $\ell_1, \ell_2, r_1, r_2, 0$). Our observation is that the specification of this constraint can be made external to the topology matrix (see Figure 5c, refined topology), reducing its size in two, and achieving corresponding performance improvement.

However, neither the topology matrix format of [YHH+23] nor the above improvement are suited to our approach, because their $\mathcal{V}$ explicitly knows 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 topology matrices of instructions of our ZK CPU (Figure 6). In particular, constants 0 and 1 and instruction inputs are not processed in a distinguished manner, but rather they are 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:

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

Here, n is the number of linear constraints defining a m-instruction (see Definition 1). We set $n = n_{in} - m = n_\times + m + 2$ (see Section 3). This hides the true number of each gate type in the instruction. Looking ahead, an honest $\mathcal{P}$ will *privately* order the gates, starting from $1 \cdot 1 = 1$ (to capture 1 in the extended witness), followed by m registers, 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 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

In this section, let us consider a tight ZK CPU with B instructions $\mathcal{C}_1, \ldots, \mathcal{C}_B$, each of different size, and suppose that $\mathcal{P}$ wishes to execute $\mathcal{C}_1$ followed by $\mathcal{C}_2$ (i.e., $\mathcal{C}_2 \circ \mathcal{C}_1$).

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 $\mathcal{C}_{i \in [B]}$, we assume $n^{(i)} = n_{in}^{(i)} = n_\times^{(i)} + 2$ where $\mathcal{C}_i$ has $n_{in}^{(i)}$ inputs, $n_\times^{(i)}$ multiplications.

Suppose $\mathcal{P}$ wishes to first execute $\mathcal{C}_1$, then execute $\mathcal{C}_2$. $\mathcal{V}$ 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 $\mathcal{C}$ that expresses the composition $C_2 \circ C_1$. In particular, $\mathcal{C}$ can be described by simply concatenating the gate-by-gate description of $\mathcal{C}_1$ and $\mathcal{C}_2$ and appropriately shifting the names (indexes) of $\mathcal{C}_2$'s gates and wires by $n^{(1)}$. A key observation

is that the topology matrix for $\mathcal{C}$ can be constructed by appropriately combining the topology matrices for $\mathcal{C}_1$ and $\mathcal{C}_2$:

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

where $\boldsymbol{M}^{(1)}$ (resp. $\boldsymbol{M}^{(2)}$) is the topology matrix induced by $\mathcal{C}_1$ (resp. $\mathcal{C}_2$). Our approach hides $\mathcal{C}$ (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 n inputs $\boldsymbol{in}$ and n MULT tuples $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ in the order described by Equation (3).
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:
$$\begin{aligned} &\overbrace{(1, \chi, \ldots, \chi^{2n-1}) \times \boldsymbol{M}}^{\text{topology vector } \boldsymbol{c}} \times \overbrace{(in_1, \ldots, o_n)^T}^{\text{committed}} \\ =&\underbrace{(1, \chi, \ldots, \chi^{2n-1})}_{\text{public}} \times \underbrace{(\ell_1, \ldots, r_n)^T}_{\text{committed}} \end{aligned}$$

To achieve the above steps while hiding $\mathcal{C}$, $\mathcal{P}$ 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 arbitrary 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 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 following vector:

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

where *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 executed 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 $\langle \boldsymbol{c}, (in_1, \ldots, o_n)\rangle = \langle (1, \ldots, \chi^{2n-1}), (\ell_1, \ldots, r_n)\rangle$.

4.3.2 Handling Constant 1

Recall that the first MULT gate in each instruction should output 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, but so far, we have not explained how the *inputs* to this gate can be properly constrained. 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 persistent registers.

A naïve (failing) attempt to pass a 1 into an instruction would be to have a fixed wire of $\mathcal{C}$ 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 $\mathcal{C}$, 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 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 $\mathcal{C}_{i\in[B]}$ has a *fixed* offset to access (enforce) an input constraint of the next instruction. Thus, $\mathcal{C}_{i\in[B]}$'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 $\mathcal{C}$.

This cleanly translates into our matrix representation. Let's go through to our concrete example of $\mathcal{P}$ proving a circuit $\mathcal{C}$ consisting of $\mathcal{C}_1$ followed by $\mathcal{C}_2$. Formally, the entire proof will be based on

a (slightly) updated equation:

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

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.) The IS will consist of B instructions $\boldsymbol{M}_*^{(i\in[B])}$.

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 $\boldsymbol{M}_* = \begin{pmatrix} \boldsymbol{M}_*^{(1)} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{M}_*^{(2)} \end{pmatrix}$. 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.

Jumping ahead, similarly to our importing a $1 = 1 \cdot 1$ into an instruction, we will import registers via $\mathsf{reg} = 1 \cdot \mathsf{reg}$:

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 $\mathcal{C}_1$ followed by $\mathcal{C}_2$, where the register is initialized to x as $\mathcal{C}$'s input and stores y as $\mathcal{C}$'s output (x, y are public). $\mathcal{P}$ can commit $n = n^{(1)} + n^{(2)}$ inputs and MULT tuples and show:

$$\begin{array}{l|l} \boldsymbol{M} \times (1, x, in_1, \dots, o_n)^T & \\ = (\ell_1, \dots, r_n, 1, 1, 1, y)^T & \boldsymbol{M} \triangleq \begin{pmatrix} 1 & 0 & \boldsymbol{0} & \boldsymbol{0} \\ 1 & 0 & \boldsymbol{0} & \boldsymbol{0} \\ 1 & 0 & \boldsymbol{0} & \boldsymbol{0} \\ 0 & 1 & \boldsymbol{0} & \boldsymbol{0} \\ 0 & 0 & \boldsymbol{M}^{(1)} & \boldsymbol{0} \\ 0 & 0 & \boldsymbol{0} & \boldsymbol{M}^{(2)} \end{pmatrix} \end{array}$$

$\boldsymbol{M}$ is private but $\mathcal{P}$ and $\mathcal{V}$ can obtain the commitment of $(1, \chi, \chi^2, \dots) \times \boldsymbol{M}$ by constructing the commitment of $(1, \dots, \chi^{2n-1}) \times \begin{pmatrix} \boldsymbol{M}^{(1)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{M}^{(2)} \end{pmatrix}$ (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 $\mathcal{C}_2 \circ \mathcal{C}_1$; our discussion applies generally. We exploit the following equality:

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

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). We only need to construct $\mathsf{com}(\boldsymbol{b})$, and the crucial observation is the following vectors are *public*:

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

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

1. $\mathcal{P}$ and $\mathcal{V}$ agree on a read-only memory (ROM) storing (public) entries $(1, \boldsymbol{v}^{(1)}), \dots, (B, \boldsymbol{v}^{(B)})$.
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

We first observe that it is sufficient 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.

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., [YH23], 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 see the intuition behind our protocol, 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 $\mathcal{P}$ and $\mathcal{V}$ 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), $\mathcal{P}$ and $\mathcal{V}$ 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 $\mathsf{com}(\langle (1, \gamma, \gamma^2, \ldots), \boldsymbol{w} \rangle - mac^{(t)})$. 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 $(1, mac^{(1)}), \ldots, (B, mac^{(B)})$ 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 $\mathsf{com}(\boldsymbol{smac})$ by "reading" single-element ZK ROM (see Section 2.4) initialized by $mac^{(1)}, \ldots, mac^{(B)}$ at positions $\boldsymbol{id}$, where each

$$smac_{i \in [n]} = mac^{(id_i)}$$

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):

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

4. $\mathcal{P}$ proves that for each $p_{i \in [n]} = 1$, it holds $v_i \neq 0$. 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 tuple $\boldsymbol{a} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{s}), \boldsymbol{b} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{v})$, $\boldsymbol{c} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{smac})$ (in order, total $\mathsf{HW}(\boldsymbol{p})$ pairs), $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \mathsf{last}(\boldsymbol{c})$. This corresponds to the check in Step 2 of the single-read task. This can be performed by proving that, $\forall i \in [n]$:

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

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, and we can work with a single copy of $\boldsymbol{p}$ and $\boldsymbol{id}$.

A second, 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. 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.

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

We define the ideal functionality for CPZK with a single read-only memory for *unbalanced, non-zero-end* vectors, denoted $\mathcal{F}_{\textsf{CPZK-UROM}}$ and presented in Figure 8. $\mathcal{F}_{\textsf{CPZK-UROM}}$ is defined similarly to $\mathcal{F}_{\textsf{CPZK-ROM}}$. The main difference is that $\mathcal{F}_{\textsf{CPZK-UROM}}$ allows $\mathcal{P}$ to initialize the UROM with different-length vectors (via `InitUROM`). Furthermore, $\mathcal{F}_{\textsf{CPZK-UROM}}$ 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 $\textsf{Filter}(\boldsymbol{p}, \boldsymbol{id})$ is the (ordered) set of indices $\mathcal{P}$ wishes to read.

5.2 Our Protocols: $\Pi_{\textsf{CPZK-UROM}}$ and $\Pi_{\textsf{ZKCPU}}$

Recall that our tight ZK CPU protocol is designed in the $\mathcal{F}_{\textsf{CPZK-UROM}}$-hybrid model, and our ZK UROM protocol is designed in the $\mathcal{F}_{\textsf{CPZK-ROM}}$-hybrid mode; see Section 4. We formalize our protocols as $\Pi_{\textsf{CPZK-UROM}}$ (Figures 9 and 10) and $\Pi_{\textsf{ZKCPU}}$ (Figures 11 and 12).

We state the security theorems regarding these two protocols. In this section, we provide only a proof sketch for each theorem for readability. The complete proofs are deferred to Appendix A.

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 $\Pi_{\textsf{CPZK-UROM}}$ (Figures 9 and 10) UC-realizes $\mathcal{F}_{\textsf{CPZK-UROM}}$ (Figure 8) in the $\mathcal{F}_{\textsf{CPZK-ROM}}$-hybrid model (Figure 4) 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.*

Proof Sketch. The proof is performed by constructing the simulator $\mathcal{S}$. Note that the instructions related to the CPZK part in $\mathcal{F}_{\textsf{CPZK-UROM}}$ are the *same* as $\mathcal{F}_{\textsf{CPZK-ROM}}$ (see the "CPZK" box in $\mathcal{F}_{\textsf{CPZK-UROM}}$ and $\mathcal{F}_{\textsf{CPZK-ROM}}$; note this is not the ZK property). Thus, the simulation for these instructions is straightforward. Here, we only focus on constructing the simulator for the instructions in the unbalanced non-zero-end read-only memory part.

For these instructions, we need to show completeness (trivial, omitted); soundness (constructing $\mathcal{S}$ for $\mathcal{P}^*$); and Zero-Knowledge (constructing $\mathcal{S}$ for $\mathcal{V}^*$).

Zero-Knowledge, $\mathcal{S}$ for $\mathcal{V}^*$: Note that the simulator for a malicious $\mathcal{V}^*$ is trivial. This is because (1) $\mathcal{P}$ has no output, and (2) $\mathcal{V}^*$ in the real-world execution only receives some commitment IDs (i.e., *cid*s). In particular, these *cid*s are revealed by the $\mathcal{F}_{\textsf{CPZK-UROM}}$ to the simulator. (Indeed, $\mathcal{V}^*$ also receives some output from the `Check` call, but the result is always `true`.) Thus, the simulation is perfect.

Soundness, $\mathcal{S}$ for $\mathcal{P}^*$: For a malicious $\mathcal{P}^*$, the simulator $\mathcal{S}$ interacts with $\mathcal{F}_{\textsf{CPZK-UROM}}$, runs $\mathcal{P}^*$ as a subroutine, and emulates the hybrid $\mathcal{F}_{\textsf{CPZK-ROM}}$ for her. In particular, the simulator will emulate a real-world honest $\mathcal{V}$ interacting with $\mathcal{P}^*$. As $\mathcal{V}$ has *no* input, $\mathcal{S}$ can trivially emulate him. Crucially, $\mathcal{S}$ can trivially extract $\mathcal{P}^*$ inputs (i.e., the witness) to each instruction of $\mathcal{F}_{\textsf{CPZK-ROM}}$. If the emulated $\mathcal{V}$ outputs `cheating`, $\mathcal{S}$ sets the flag in $\mathcal{F}_{\textsf{CPZK-UROM}}$ to output `cheating` to the ideal $\mathcal{V}$; otherwise, $\mathcal{S}$ simply sends the extracted inputs to $\mathcal{F}_{\textsf{CPZK-UROM}}$.

Functionality $\mathcal{F}_{\textsf{CPZK-UROM}}$

$\mathcal{F}_{\textsf{CPZK-UROM}}$, 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 $\mathcal{F}_{\textsf{CPZK}}$.

Unbalanced Non-Zero-End Read-Only Memory

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

1. For each $i \in [B]$, fetch $(u_1^{(i)}, x_1^{(i)}), \ldots, (u_{n^{(i)}}^{(i)}, x_{n^{(i)}}^{(i)})$ and let $\boldsymbol{x}^{(i)} := (x_1^{(i)}, \ldots, x_{n^{(i)}}^{(i)})$. 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 $f_{\mathsf{urom}} := \texttt{honest}$.
3. Send $(\texttt{initurom}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ to $\mathcal{V}$ and $\mathcal{S}$.

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

Set Program. On receiving $(\mathit{SetProg}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$ 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 $(\mathit{setprog}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$ 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 $\texttt{SetProg}$.

Read UROM. On receiving $(\texttt{ReadUROM}, \boldsymbol{cid}^{(d)}, \boldsymbol{d})$ from $\mathcal{P}$ where (1) $\texttt{InitUROM}$ and $\texttt{SetProg}$ 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 $(cid_1^{(d)}, d_1), \ldots, (cid_n^{(d)}, d_n)$.

1. If $\mathcal{P}$ is honest, $\boldsymbol{id} \in [B]^n$.
2. If $\mathcal{P}$ is corrupted, set $f_{\mathsf{urom}} := \texttt{cheating}$ 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 $f_{\mathsf{urom}} := \texttt{cheating}$ when:
$$\mathsf{last}(\boldsymbol{x}) = 0 \text{ or } X[\mathsf{last}(\boldsymbol{y})] \neq \boldsymbol{x}$$

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

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

Figure 8: Ideal functionality for commit-and-prove zero-knowledge with a single read-only memory for unbalanced non-zero-end vectors.

Protocol $\Pi_{\mathsf{CPZK\text{-}UROM}}$

The protocol is parameterized by finite field $\mathbb{F}$. All instructions of the CPZK part are handled by $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ in the natural way.

Intialize UROM. $\mathcal{P}$ selects vectors (commitments) to initialize UROM. In particular, $\mathcal{P}$ proves that each initial vector is non-zero-end.

1. $\mathcal{P}$ generates B fresh[a] *cids* as $\boldsymbol{cid}^{(li)}$. $\mathcal{P}$ sends (a) $(\mathtt{Commit}, cid_i^{(li)}, li_i)$ for each $i \in [B]$ to $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$, where α_i is the element committed by $\mathsf{last}(\boldsymbol{u}^{(i)})$ and $li_i := \mathsf{Inverse}(\alpha_i)$; (b) $(Check, \mathcal{C}_0^{\mathsf{check}}, \mathsf{last}(\boldsymbol{u}^{(i\in[B])}), \boldsymbol{cid}^{(li)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$, where $\mathcal{C}_0^{\mathsf{check}}$ is a circuit with two length-B inputs $\boldsymbol{\alpha}$, $\boldsymbol{li}$ that outputs $\alpha_i \cdot li_i - 1$ for each $i \in [B]$; and (c) $(\mathtt{InitUROM}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ to $\mathcal{V}$.

2. $\mathcal{V}$ on receiving $(\mathtt{InitUROM}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ from $\mathcal{P}$: For each $\boldsymbol{u}^{(i\in[B])} = (u_1^{(i)}, \ldots, u_{n^{(i)}}^{(i)})$, $\mathcal{V}$ checks if each $u_{j\in[n^{(i)}]}^{(i)}$ exists as a *cid* (from previous $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$'s $(\mathtt{commit}, \cdot)$ or $(\mathtt{linear}, \cdot)$ messages). If so, then $\mathcal{V}$ checks if he receives (a) $(\mathtt{commit}, cid_i^{(li)})$ for each $i \in [B]$ from $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$; and (b) $(check, \mathcal{C}_0^{\mathsf{check}}, \mathsf{last}(\boldsymbol{u}^{(i\in[B])}), \boldsymbol{cid}^{(li)}, \mathtt{true})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$[b]. If not, $\mathcal{V}$ halts; otherwise, $\mathcal{V}$ outputs $(\mathtt{initurom}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$, marks $\mathtt{initurom}$ as being executed, and ignores the subsequent $\mathtt{InitUROM}$ messages.

Set Program. $\mathcal{P}$ commits the boundary string $\boldsymbol{p}$, and the vector $\boldsymbol{id}$ encoding the execution path at each position where $p[i] = 1$.

3. $\mathcal{P}$ sends (a) $(\mathtt{Open}, cid_n^{(p)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$; (b) $(\mathtt{Check}, \mathcal{C}_1^{\mathsf{check}}, cid_1^{(p)}, \ldots, cid_{n-1}^{(p)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$, where $\mathcal{C}_1^{\mathsf{check}}$ is a circuit with $n-1$ inputs $p_1, \ldots, p_{n-1}$ that outputs $\boldsymbol{p} \odot (\boldsymbol{1} - \boldsymbol{p})$; and (c) $(SetProg, \boldsymbol{cid}^{(id)})$ to $\mathcal{V}$.

4. $\mathcal{V}$ on receiving $(SetProg, \boldsymbol{cid}^{(id)})$ from $\mathcal{P}$, where $|\boldsymbol{cid}^{(id)}| = n \in \mathbb{Z}^+$ and each $cid_{i\in[n]}^{(id)}$ exists as a *cid*: If $\mathcal{V}$ does *not* receive (a) $(\mathtt{open}, cid_n^{(p)}, 1)$ from $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ or (b) $(\mathtt{check}, \mathcal{C}_1^{\mathsf{check}}, cid_1^{(p)}, \ldots, cid_{n-1}^{(p)}, \mathtt{true})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ [b], $\mathcal{V}$ halts; otherwise, $\mathcal{V}$ outputs and records $(setprog, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$, marks $\mathtt{setprog}$ as being executed, and ignores the subsequent $\mathtt{SetProg}$ messages.

Read UROM. $\mathcal{P}$ directly commits the reading results as $\boldsymbol{d}$ (and later proves that she does not cheat).

5. For each $i \in [n]$, $\mathcal{P}$ sends $(\mathtt{Commit}, cid_i^{(d)}, d_i)$ to $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$; and (b) $(\mathtt{ReadUROM})$ to $\mathcal{V}$.

6. $\mathcal{V}$ on receiving $(\mathtt{ReadUROM})$ from $\mathcal{P}$, $\mathcal{V}$ ignores the messages if $\mathtt{initurom}$ or $\mathtt{setprog}$ has not been executed. Otherwise, let $|\boldsymbol{cid}^{(p)}| = |\boldsymbol{cid}^{(id)}| = n \in \mathbb{Z}^+$. For each $i \in [n]$, $\mathcal{V}$ obtains $(\mathtt{commit}, cid_i^{(d)})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$. $\mathcal{V}$ outputs and records $(\mathtt{readurom}, \boldsymbol{cid}^{(d)})$, marks $\mathtt{readurom}$ as being executed, and ignores the subsequent $\mathtt{ReadUROM}$ messages.

[a] We assume these *cids* will *not* be used in the future as the inputs from the environment $\mathcal{E}$ to avoid trivial distinguisher.

[b] Note that $\mathcal{V}$ also checks that the circuit is constructed correctly with expected commitments as inputs. In particular, $\mathcal{V}$ can notice $\mathcal{P}$'s abort.

Figure 9: The protocol of CPZK with a single ROM for unbalanced non-zero-end vectors in the $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid model.

Protocol $\Pi_{\textsf{CPZK-UROM}}$ (Cont.)

Check UROM. $\mathcal{P}$ sends (`CheckUROM`) to $\mathcal{V}$. $\mathcal{V}$ ignores the message if `readurom` has not been executed. Otherwise, $\mathcal{P}$ and $\mathcal{V}$ retrieve $\boldsymbol{u}^{(i\in[B])}$ and $\boldsymbol{cid}^{(p,id,d)}$, where $|\boldsymbol{u}^{(i\in[B])}| = n^{(i)}$ and $|\boldsymbol{cid}^{(p)}| = |\boldsymbol{cid}^{(id)}| = |\boldsymbol{cid}^{(d)}| = n$. $\mathcal{P}$ also retrieves $\boldsymbol{p}, \boldsymbol{id}, \boldsymbol{d} \in \mathbb{F}^n$. Proceed:

7. **$\mathcal{P}$ convinces $\mathcal{V}$ that for each $\boldsymbol{x} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{d})$, the $\mathsf{last}(\boldsymbol{x}) \neq 0$:**
   (a) $\mathcal{P}$ generates n fresh[a] *cids* as $cid_1^{(inv)}, \ldots, cid_n^{(inv)}$. For each $i \in [n]$, $\mathcal{P}$ sends (`Commit`, $cid_i^{(inv)}, \mathsf{Inverse}(d_i)$) to $\mathcal{F}_{\textsf{CPZK-ROM}}$ if $p_i = 1$; $\mathcal{P}$ sends (`Commit`, $cid_i^{(inv)}, 0$) to $\mathcal{F}_{\textsf{CPZK-ROM}}$ if $p_i = 0$. $\mathcal{P}$ further sends (*Check*, $\mathcal{C}_2^{\textsf{check}}, \boldsymbol{cid}^{(inv)}, \boldsymbol{cid}^{(d)}, \boldsymbol{cid}^{(p)}$) to $\mathcal{F}_{\textsf{CPZK-ROM}}$ where $\mathcal{C}_2^{\textsf{check}}$ is a circuit with three length-n inputs $\boldsymbol{inv}, \boldsymbol{d}, \boldsymbol{p}$ that outputs $inv_i \cdot d_i - p_i$ for each $i \in [n]$.
   (b) If $\mathcal{V}$ does not receive (i) n (`commit`, $\cdot$) messages for each $cid_{i\in[n]}^{(inv)}$; or (ii) (`check`, $\mathcal{C}_2^{\textsf{check}}, \boldsymbol{cid}^{(\texttt{inv})}, \boldsymbol{cid}^{(d)}, \boldsymbol{cid}^{(p)}$, `true`) from $\mathcal{F}_{\textsf{CPZK-ROM}}$ [b], $\mathcal{V}$ outputs (`checkurom`, `cheating`) and aborts.

8. **$\mathcal{P}$ convinces $\mathcal{V}$ that (a) $\boldsymbol{id} \in [B]^n$ and (b) for each $\boldsymbol{x} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{d})$, $\boldsymbol{y} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{id})$ pair in order, $\boldsymbol{x}$ is exactly the vector committed by $\boldsymbol{u}^{(\mathsf{last}(\boldsymbol{y}))}$:** $\mathcal{V}$ samples a uniform challenge $\gamma \in_\$ \mathbb{F}$ and sends it to $\mathcal{P}$. Then proceed as follows:
   (a) **Generate committed single-element MAC of each vector committed by $\boldsymbol{u}^{(i\in[B])}$ by polynomial evaluation at γ:** For each $i \in [B]$, $\mathcal{P}$ generates fresh[a] *cid* as $cid_i^{(mac)}$, then $\mathcal{P}$ sends (`Linear`, $cid_i^{(mac)}, \boldsymbol{u}^{(i)}, 0, 1, \gamma, \ldots, \gamma^{n^{(i)}-1}$) to $\mathcal{F}_{\textsf{CPZK-ROM}}$. For each $i \in [B]$, if $\mathcal{V}$ does not receive (`linear`, $cid_i^{(mac)}, \boldsymbol{u}^{(i)}, 0, 1, \gamma, \ldots, \gamma^{n^{(i)}-1}$) from $\mathcal{F}_{\textsf{CPZK-ROM}}$ (Note that $\mathcal{V}$ already knows $\boldsymbol{u}^{(i)}$), $\mathcal{V}$ outputs (`checkurom`, `cheating`) and aborts. Otherwise, $\mathcal{V}$ records $\boldsymbol{cid}^{(mac)}$.
   (b) **$\mathcal{P}$ uses (hybrid) single-element ZK ROM to construct a length-n vector of selected MACs by committed $\boldsymbol{id}$:**
      i. $\mathcal{P}$ sends (*InitROM*, $\boldsymbol{cid}^{(mac)}$) to $\mathcal{F}_{\textsf{CPZK-ROM}}$. If $\mathcal{V}$ does not receive (*initrom*, $\boldsymbol{cid}^{(mac)}$) from $\mathcal{F}_{\textsf{CPZK-ROM}}$ (Note that $\mathcal{V}$ already knows $\boldsymbol{cid}^{(mac)}$), $\mathcal{V}$ outputs (`checkurom`, `cheating`) and aborts.
      ii. $\mathcal{P}$ generates n fresh[a] *cids* as $cid_1^{(smac)}, \ldots, cid_n^{(smac)}$. For each $i \in [n]$, $\mathcal{P}$ sets $smac_i := mac[id_i]$. $\mathcal{P}$ then sends (*ReadROM*, $\boldsymbol{cid}^{(smac)}, \boldsymbol{smac}, \boldsymbol{cid}^{(id)}$) to $\mathcal{F}_{\textsf{CPZK-ROM}}$. If $\mathcal{V}$ does not receive (*readrom*, $\boldsymbol{cid}^{(smac)}, \boldsymbol{cid}^{(id)}$) from $\mathcal{F}_{\textsf{CPZK-ROM}}$ (Note that $\mathcal{V}$ already knows $\boldsymbol{cid}^{(id)}$), $\mathcal{V}$ outputs (`checkurom`, `cheating`) and aborts. $\mathcal{V}$ records $\boldsymbol{cid}^{(smac)}$.
      iii. $\mathcal{P}$ sends (`CheckROM`) to $\mathcal{F}_{\textsf{CPZK-ROM}}$. If $\mathcal{V}$ does not receives (`checkrom`, `honest`) from $\mathcal{F}_{\textsf{CPZK-ROM}}$, $\mathcal{V}$ outputs (`checkurom`, `cheating`) and aborts.
   (c) **$\mathcal{P}$ convinces $\mathcal{V}$ that the committed $\boldsymbol{smac}$ at $p_{i\in[n]} = 1$ are indeed equal to $(1, \gamma, \gamma^2 \cdots) \times \boldsymbol{x}^T$ for each $\boldsymbol{x} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{d})$:**
      i. $\mathcal{P}$ sends (*Check*, $\mathcal{C}_3^{\textsf{check}}, \boldsymbol{cid}^{(d)}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(smac)}$) to $\mathcal{F}_{\textsf{CPZK-ROM}}$, where $\mathcal{C}_3^{\textsf{check}}$ is a circuit with inputs $\boldsymbol{d}, \boldsymbol{p}, \boldsymbol{smac}$ that outputs: $p_i \cdot (\sum_{j=1}^{i} d_j \cdot s_j - \sum_{j=1}^{i} p_j \cdot smac_j)$ for each $i \in [n]$ where $\boldsymbol{s} \triangleq \mathsf{Expand}_2(\boldsymbol{p}, \gamma)$ (defined in Step 2 of Section 4.1).
      ii. If $\mathcal{V}$ does not receive the message (`check`, $\mathcal{C}_3^{\textsf{check}}, \boldsymbol{cid}^{(d)}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(smac)}$, `true`) from $\mathcal{F}_{\textsf{CPZK-ROM}}$ [b], $\mathcal{V}$ outputs (`checkurom`, `cheating`) and aborts; otherwise, $\mathcal{V}$ outputs (`checkurom`, `honest`).

Figure 10: The (Cont.) protocol of CPZK with a single ROM for unbalanced non-zero-end vectors in the $\mathcal{F}_{\textsf{CPZK-ROM}}$-hybrid model.

Protocol Π_{ZKCPU}

Π_{ZKCPU} runs with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and is parameterized by a field $\mathbb{F}$, an non-negative integer m, a positive integer B and B m-instructions (defined in Definition 1) $\mathcal{C}_1, \ldots, \mathcal{C}_B$, 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 $\mathcal{C}_i$ 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 $n^{(i)}$. Recall that each m-instruction induces a topology matrix $\boldsymbol{M}^{(i\in[B])} \in \mathbb{F}^{2n^{(i)} \times 2n^{(i)}}$ as defined in Equation (5), where we change the right-top corner of each $\boldsymbol{M}^{(i\in[B])}$ from 0 to 1 (see discussion in Section 4.4.1). Π_{ZKCPU} proceeds:

1. **$\mathcal{P}$ claims the size of the execution.** $\mathcal{P}$ on receiving $(\texttt{Prove}, \tau, i_1, \ldots, i_\tau, \boldsymbol{in}_1, \ldots, \boldsymbol{in}_\tau)$, $\mathcal{P}$ calculates and sends $n \triangleq \sum_{j\in[\tau]} n^{(i_j)}$.

2. **$\mathcal{P}$ commits $\boldsymbol{in}, \boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ of the execution.** $\mathcal{P}$ constructs the following 4 length-n vectors: $\boldsymbol{in} \triangleq \boldsymbol{in}_1 \| \cdots \| \boldsymbol{in}_\tau$. (Each $|\boldsymbol{in}_{j\in[\tau]}| = n_{in}^{(i_j)} - m$.) Set $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ be empty. Let the m registers be $\boldsymbol{st}_j$ after executing the first j instructions. For each $j \in [\tau]$ in sequence:

   (a) Let $\boldsymbol{\ell} := \boldsymbol{\ell} \| 1$, $\boldsymbol{r} := \boldsymbol{r} \| 1$, $\boldsymbol{o} := \boldsymbol{o} \| 1$. I.e., this captures the 1 in the extended witness as a $1 \cdot 1 = 1$ multiplication.

   (b) For each $k \in [m]$, let $\boldsymbol{\ell} := \boldsymbol{\ell} \| 1$, $\boldsymbol{r} := \boldsymbol{r} \| \boldsymbol{st}_{j-1}[k]$, $\boldsymbol{o} := \boldsymbol{o} \| \boldsymbol{st}_{j-1}[k]$. I.e., this captures register inputs as multiplications.

   (c) For each multiplication in $\mathcal{C}_{i_j}$ (in the same order related to $\boldsymbol{M}^{(i_j)}$), let the left/right/output wire value of this multiplication be $val^{(\ell,r,o)}$ (i.e., they can be driven from evaluating $\mathcal{C}_{i_j}(\boldsymbol{st}_{j-1} \| \boldsymbol{in}_j)$): Let $\boldsymbol{\ell} := \boldsymbol{\ell} \| val^{(\ell)}$, $\boldsymbol{r} := \boldsymbol{r} \| val^{(r)}$, $\boldsymbol{o} := \boldsymbol{o} \| val^{(o)}$.

   (d) Let $\boldsymbol{\ell} := \boldsymbol{\ell} \| 1$, $\boldsymbol{r} := \boldsymbol{r} \| 0$, $\boldsymbol{o} := \boldsymbol{o} \| 0$. I.e., this captures the 0 checking output as a multiplication.

   $\mathcal{P}$ generates $4n$ fresh $cids$ as $\boldsymbol{cid}^{(in/\ell/r/o)}$. For each $j \in [n]$, $\mathcal{P}$ sends $(\texttt{Commit}, cid_j^{(in/\ell/r/o)}, in_j/\ell_j/r_j/o_j)$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$.

3. **$\mathcal{P}$ commits $\boldsymbol{p}, \boldsymbol{id}$ where (a) $\boldsymbol{p}$ is a length-n boundary string marking 1 at positions $\sum_{j=1}^{k} n^{(i_j)}$ for each $k \in [\tau]$; and (b) $\boldsymbol{id} \in [B]^n$ such that $\mathsf{Filter}(\boldsymbol{p}, \boldsymbol{id}) = \{i_1, \ldots, i_\tau\}$.** $\mathcal{P}$ constructs the following 2 length-n vectors:

$$\boldsymbol{p} = (\overbrace{0, \ldots, 0, 1}^{n^{(i_1)}}, \overbrace{0, \ldots, 0, 1}^{n^{(i_2)}}, \ldots, \overbrace{0, \ldots, 0, 1}^{n^{(i_\tau)}}) \quad \boldsymbol{id} = (\overbrace{i_1, \ldots, i_1}^{n^{(i_1)}}, \overbrace{i_2, \ldots, i_2}^{n^{(i_2)}}, \ldots, \overbrace{i_\tau, \ldots, i_\tau}^{n^{(i_\tau)}})$$

   $\mathcal{P}$ generates $2n$ fresh $cids$ as $\boldsymbol{cid}^{(p/id)}$. For each $j \in [n]$, $\mathcal{P}$ sends $(\texttt{Commit}, cid_j^{(p/id)}, p_j/id_j)$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$.

4. **$\mathcal{V}$ issues uniform challenge $\chi \in_\$ \mathbb{F}$ to compress each topology matrix to topology vector.** $\mathcal{V}$ on receiving n from $\mathcal{P}$, $\mathcal{V}$ waits for $6n$ $cids$ as $(\texttt{commit}, cid_{j\in[n]}^{(in/\ell/r/o/p/id)})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. $\mathcal{V}$ samples and sends $\chi \in_\$ \mathbb{F}$ to $\mathcal{P}$.

5. **$\mathcal{P}$ and $\mathcal{V}$ initialize ZK UROM using B topology vectors.** $\mathcal{P}$ and $\mathcal{V}$ compute $\boldsymbol{v}^{(i)} := (1, \chi, \ldots, \chi^{2n^{(i)}-1}) \times \boldsymbol{M}^{(i)}$ for each $i \in [B]$. Recall that each $\mathsf{last}(\boldsymbol{v}^{(i\in[B])}) = 1$. For each $i \in [B]$, each $j \in 2n^{(i)}$, $\mathcal{P}$ generates a fresh cid as $u_j^{(i)}$, $\mathcal{P}$ sends $(\texttt{Linear}, u_j^{(i)}, v_j^{(i)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$; $\mathcal{V}$ obtains $(\texttt{linear}, u_j^{(i)}, \widetilde{v_j^{(i)}})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$; if $\widetilde{v_j^{(i)}} \neq v_j^{(i)}$, $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts. $\mathcal{P}$ finally sends $(\texttt{InitUROM}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. Note that this is free since $\boldsymbol{u}^{(i\in[B])}$ committing values known by $\mathcal{V}$.

Figure 11: The protocol of a tight ZK CPU in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid model.

Protocol Π_{ZKCPU} (Cont.)

6. $\mathcal{P}$ **uses committed $\boldsymbol{p}, \boldsymbol{id}$ to read ZK UROM.** $\mathcal{P}$ and $\mathcal{V}$ use `Linear` in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ hybrid to generate a 0 committed by cid^{zero}.

   (a) $\mathcal{P}$ sends $(\mathtt{SetProg}, cid^{(zero)}, cid_1^{(p)}, \ldots, cid^{(zero)}, cid_n^{(p)}, cid_1^{(id)}, cid_1^{(id)}, \ldots, cid_n^{(id)}, cid_n^{(id)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. If $\mathcal{V}$ does not receive the **setprog** message from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, $\mathcal{V}$ outputs $(\mathtt{prove}, \mathtt{false}, n)$ and aborts. Otherwise, let $\mathcal{V}$ receive $(\widetilde{setprog}, \widetilde{cid^{(zero)}}, \widetilde{cid_1^{(p)}}, \ldots, \widetilde{cid^{(zero)}}, \widetilde{cid_n^{(p)}}, \widetilde{cid_1^{(id)}}, \widetilde{cid_1^{(id)}}, \ldots, \widetilde{cid_n^{(id)}}, \widetilde{cid_n^{(id)}})$. If $\widetilde{\boldsymbol{cid}^{(p)}} \neq \boldsymbol{cid}^{(p)}$ or $\widetilde{\boldsymbol{cid}^{(id)}} \neq \boldsymbol{cid}^{(id)}$ ($\mathcal{V}$ received in Step 3) or $\widetilde{cid^{zero}} \neq cid^{zero}$, $\mathcal{V}$ outputs $(\mathtt{prove}, \mathtt{false}, n)$ and aborts.

   (b) $\mathcal{P}$ constructs a length-$2n$ vector $\boldsymbol{d} \triangleq \boldsymbol{v}^{(i_1)} \| \ldots \| \boldsymbol{v}^{(i_\tau)}$. $\mathcal{P}$ generates 2n fresh $cids$ as $\boldsymbol{cid}^{(d)}$. $\mathcal{P}$ sends $(\mathtt{ReadUROM}, \boldsymbol{cid}^{(d)}, \boldsymbol{d})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. $\mathcal{V}$ obtains $(\mathtt{ReadUROM}, \boldsymbol{cid}^{(d)})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. $\mathcal{P}$ sends $(\mathtt{CheckUROM})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. If $\mathcal{V}$ does *not* receive $(\mathtt{checkurom}, \mathtt{honest})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, $\mathcal{V}$ outputs $(\mathtt{prove}, \mathtt{false}, n)$ and aborts.

7. $\mathcal{V}$ **checks that the multiplications are formed correctly as well as all linear constraints.**

   (a) $\mathcal{P}$ sends $(\mathtt{Check}, \mathcal{C}_4^{\mathsf{check}}, \boldsymbol{cid}^{(\ell)}, \boldsymbol{cid}^{(r)}, \boldsymbol{cid}^{(o)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, where $\mathcal{C}_4^{\mathsf{check}}$ is a circuit with inputs $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o} \in \mathbb{F}^n$ that outputs: $\ell_j \cdot r_j - o_j$ for each $j \in [n]$. If $\mathcal{V}$ does not receive $(\mathtt{check}, \mathcal{C}_4^{\mathsf{check}}, \boldsymbol{cid}^{(\ell)}, \boldsymbol{cid}^{(r)}, \boldsymbol{cid}^{(o)}, \mathtt{true})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, $\mathcal{V}$ outputs $(\mathtt{prove}, \mathtt{false}, n)$ and aborts. Note that $\mathcal{V}$ already has $\boldsymbol{cid}^{(\ell)}, \boldsymbol{cid}^{(r)}, \boldsymbol{cid}^{(o)}$ and can construct $\mathcal{C}_4^{\mathsf{check}}$ since $\mathcal{V}$ knows n.

   (b) $\mathcal{P}$ sends $(\mathtt{Check}, \mathcal{C}_5^{\mathsf{check}}, \boldsymbol{cid}^{(o)}, \boldsymbol{cid}^{(p)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, where $\mathcal{C}_5^{\mathsf{check}}$ is a circuit with inputs $\boldsymbol{o}, \boldsymbol{p} \in \mathbb{F}^n$ that outputs: $o_j \cdot p_j$ for each $j \in [n]$. If $\mathcal{V}$ does not receive $(\mathtt{check}, \mathcal{C}_5^{\mathsf{check}}, \boldsymbol{cid}^{(o)}, \boldsymbol{cid}^{(p)}, \mathtt{true})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, $\mathcal{V}$ outputs $(\mathtt{prove}, \mathtt{false}, n)$ and aborts. Note that $\mathcal{V}$ already has $\boldsymbol{cid}^{(o)}, \boldsymbol{cid}^{(p)}$ and can construct $\mathcal{C}_5^{\mathsf{check}}$ since $\mathcal{V}$ knows n.

   (c) $\mathcal{P}$ sends $(Check, \mathcal{C}_6^{\mathsf{check}}, \boldsymbol{cid}^{(in)}, \boldsymbol{cid}^{(\ell)}, \boldsymbol{cid}^{(r)}, \boldsymbol{cid}^{(o)}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(d)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, where $\mathcal{C}_6^{\mathsf{check}}$ is a circuit with inputs $\boldsymbol{in}, \boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}, \boldsymbol{p} \in \mathbb{F}^n$, $\boldsymbol{d} \in \mathbb{F}^{2n}$ that outputs: $sum_L - sum_R$ defined as, let $\boldsymbol{s} = \mathsf{Expand}_1(0, p_1, \ldots, 0, p_n, \chi)$ (see Section 4.1),

$$sum_L = \sum_{j=1}^{n} (\chi^{2m+2} \cdot in_j \cdot d_{2j-1} \cdot s_{2j-1}) + \sum_{j=1}^{n} (\chi^{2m+2} \cdot o_j \cdot d_{2j} \cdot s_{2j})$$

$$+ (1+\chi) + \sum_{j=1}^{m} \chi^{2j} + \sum_{j=1}^{m} (\chi^{2j+1} \cdot \boldsymbol{st}^{(0)}[j])$$

$$sum_R = \sum_{j=1}^{n} \chi^{2j-2} \cdot \ell_j + \sum_{j=1}^{n} \chi^{2j-1} \cdot r_j + \sum_{j=1}^{m} (\chi^{2n+j-1} \cdot \boldsymbol{st}^{(final)}[j])$$

   If $\mathcal{V}$ does not receive $(\mathtt{check}, \mathcal{C}_6^{\mathsf{check}}, \boldsymbol{cid}^{(\mathtt{in})}, \boldsymbol{cid}^{(\ell)}, \boldsymbol{cid}^{(r)}, \boldsymbol{cid}^{(o)}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(d)}, \mathtt{true})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, $\mathcal{V}$ outputs $(\mathtt{prove}, \mathtt{false}, n)$ and aborts. Note that $\mathcal{V}$ already has $\boldsymbol{cid}^{(in)}, \boldsymbol{cid}^{(\ell)}, \boldsymbol{cid}^{(r)}, \boldsymbol{cid}^{(o)}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(d)}$ and can construct $\mathcal{C}_6^{\mathsf{check}}$ since $\mathcal{V}$ knows $n, \chi, \boldsymbol{st}^{(0)}, \boldsymbol{st}^{(final)}$. If $\mathcal{V}$ has not abort yet, $\mathcal{V}$ outputs $(\mathtt{prove}, \mathtt{true}, n)$.

Figure 12: The (Cont.) protocol of a tight ZK CPU in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid model.

Now, it suffices to show that if the emulated $\mathcal{V}$ outputs `honset`, the ideal $\mathcal{V}$ will output `cheating` with only negligible probability. Note that if the emulated $\mathcal{V}$ outputs `honest`, $\mathcal{P}^*$ must pass all the checks in $\Pi_{\mathsf{CPZK\text{-}UROM}}$. Therefore, the ideal $\mathcal{V}$ will only output `cheating` when $\mathcal{P}^*$ wants to construct a wrong vector $\boldsymbol{d}$ but has not been caught by the checks. I.e., there is a subvector $\widetilde{\boldsymbol{x}}$ in $\boldsymbol{d}$, marked by the boundary string $\boldsymbol{p}$, that is not equal to the vector $\boldsymbol{x}^{(k)}$ (some $k \in [B]$) saved in the UROM index committed by $\mathcal{P}^*$ in $\boldsymbol{id}$. Furthermore, $\mathcal{P}^*$ pasting all checks implies that $\langle(1, \gamma, \gamma^2, \cdots), \widetilde{\boldsymbol{x}}\rangle = \langle(1, \gamma, \gamma^2, \cdots), \boldsymbol{x}^{(k)}\rangle$, where $\widetilde{\boldsymbol{x}} \neq \boldsymbol{x}^{(k)}$ and $\gamma \in_\$ \mathbb{F}$. This happens negligibly (as Corollary 1). The upper-bound of the soundness error happens when $\mathcal{P}^*$ let $\widetilde{\boldsymbol{x}}$ be the entire $\boldsymbol{d}$ and $\boldsymbol{x}^{(k\in[B])}$ be the longest stored vector. □

Theorem 2. *Protocol Π_{ZKCPU} (Figures 11 and 12) UC-realizes $\mathcal{F}_{\mathsf{ZKCPU}}$ (Figure 6) in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid model (Figure 8) with soundness error $\frac{2n+2m+1}{|\mathbb{F}|}$ and perfect zero-knowledge, in the presence of a static unbounded adversary.*

Proof Sketch. We need to show completeness (trivial, omitted); soundness (constructing $\mathcal{S}$ for $\mathcal{P}^*$); and Zero-Knowledge (constructing $\mathcal{S}$ for $\mathcal{V}^*$).

Zero-Knowledge, $\mathcal{S}$ for $\mathcal{V}^*$: Similar to our proof sketch for Theorem 1, $\mathcal{S}$ for malicious $\mathcal{V}^*$ is trivial since all the messages $\mathcal{V}^*$ received in the execution are just some commitment IDs (revealed by $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$) and `true` for several `Check` calls. Thus, the simulation is perfect.

Soundness, $\mathcal{S}$ for $\mathcal{P}^*$: For a malicious $\mathcal{P}^*$, the simulator $\mathcal{S}$ interacts with $\mathcal{F}_{\mathsf{ZKCPU}}$, runs $\mathcal{P}^*$ as a subroutine, and emulates the hybrid $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ for her. In particular, the simulator will emulate a real-world $\mathcal{V}$ interacting with $\mathcal{P}^*$. As $\mathcal{V}$ has *no* input, $\mathcal{S}$ can trivially emulate him. Crucially, $\mathcal{S}$ can trivially extract $\mathcal{P}^*$ inputs (i.e., the witness) to each instruction of $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. If the emulated $\mathcal{V}$ outputs `false`, $\mathcal{S}$ sets the flag in $\mathcal{F}_{\mathsf{ZKCPU}}$ to output `false` to the ideal $\mathcal{V}$; otherwise, $\mathcal{S}$ simply sends the extracted inputs to $\mathcal{F}_{\mathsf{ZKCPU}}$.

Now, it suffices to show that if the emulated $\mathcal{V}$ outputs `true`, the ideal $\mathcal{V}$ will output `false` with only negligible probability. Note that, this will happen only when the extracted witness is invalid but $\mathcal{P}^*$ is not caught in Π_{ZKCPU}. Since this is not a valid witness, from the definition of our topology matrices, the equality in Section 4.3.3 should not hold. I.e., the left-hand-side vector is not equal to the right-hand-side vector, where both vectors are $(2n + 2m + 2)$-length. However, the emulated $\mathcal{V}$ outputting `true` implies that the inner products between these two vectors and the vector $(1, \chi, \cdots, \chi^{2n+2m+1})$ are two equal elements. This only happens negligibly (as *Swchartz-Zippel* lemma). □

5.3 Optimization and Cost Analysis

The optimization of $\Pi_{\mathsf{CPZK\text{-}UROM}}$ includes:

1. *Public initialization*: If B vectors used to initialize UROM are public, `InitUROM` is free. This is because $\boldsymbol{u}^{(i\in[B])}$ is only used to generate commitments of $\boldsymbol{mac}$ (see Sub-step 8a), 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 `Linear` construct the commitment of (constant).

2. 1-*ended vectors*: If each vector in the UROM ends with 1 (whose inverse is 1), then vector $\boldsymbol{inv}$ is redundant (see Sub-step 7a) 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 $\varepsilon_{\mathsf{urom}} \in \mathbb{Z}^+$, then 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 $\mathcal{C}^{\mathsf{check}}_{1/2/3}$ 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 $\mathcal{C}^{\mathsf{check}}_{1/2}$. 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} can deploy *all* optimizations above and will make one call to each instruction (i.e., `InitUROM`, `SetProg`, `ReadUROM`, and `CheckUROM`). In particular, Π_{ZKCPU}, with instructions of size $n^{(1)}, \ldots, n^{(B)}$ and the total execution size n, instantiates a hybrid UROM with vectors of size $2n^{(1)}, \ldots, 2n^{(B)}$, 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} – 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 $k \in \mathbb{Z}^+$.

Consider a ZK CPU with instructions of size $n^{(1)}, \ldots, n^{(B)}$ and the total execution size n, let $\varepsilon \triangleq \mathsf{gcd}(n^{(1)}, \ldots, n^{(B)})$, 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 $\mathcal{O}\left(\sum_{i \in [B]} n^{(i)}\right)$ field operations to obtain $\boldsymbol{v}^{(i \in [B])}$ and $\boldsymbol{mac}$. Note, this relies on the technique "evaluate circuits backward"; see [YHH+23].
- Parties call `Commit` $6n + \frac{6n}{\varepsilon} + 2B$ times.
- Parties call `Linear` $2B + 1$ times to commit *constants*.
- Parties call `Open` once.
- Parties call `Check` with each of the following 9 circuits:
  - $\mathcal{C}^{\mathsf{check}}_{1/2/5}$ and $\mathsf{Expand}_{1/2}$ ($\frac{n}{\varepsilon}$ multiplications each).
  - $\mathcal{C}^{\mathsf{check}}_{3}$ ($2n + \frac{2n}{\varepsilon}$ multiplications).
  - $\mathcal{C}^{\mathsf{check}}_{4}$ (n multiplications).
  - $\mathcal{C}^{\mathsf{check}}_{6}$ ($4n$ 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 $\mathcal{O}(n)$, the protocol requires $\mathcal{O}(n)$ calls to `Commit`; $\mathcal{O}(B)$ calls to `Linear`; $\mathcal{O}(1)$ call to `Open`; $\mathcal{O}(1)$ call to `Check`.

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

- **Computation**: $\mathcal{O}\left(n + \sum_{i \in [B]} n^{(i)}\right)$ field operations.

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

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

Appendix B includes more fine-grained cost analysis – we analyze the cost of $\Pi_{\mathsf{CPZK\text{-}UROM}}$ in $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid (to realize $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$) and Π_{ZKCPU} in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid (to realize $\mathcal{F}_{\mathsf{ZKCPU}}$).

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.

[1] The circuit in $\Pi_{\mathsf{CPZK\text{-}ROM}}$ is a $\mathcal{O}(n)$-degree polynomial, but cost can be reduced since it computes products. See Lemma 3 and [YH23].

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 $\mathsf{poly}(\lambda)$ 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 [YH23] 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. [YH23]'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 [FS87]. Note that the combination of our tight ZK CPU with ZK RAM interestingly hides from $\mathcal{V}$ the number of RAM accesses.

7 Evaluation

Our implementation. Using VOLE-based ZK, we implemented $\Pi_{\mathsf{CPZK\text{-}UROM}}$ (see Figure 9) and Π_{ZKCPU} (see Figure 11). In particular, we instantiated $\mathcal{F}_{\mathsf{CPZK}}$ (see Figure 3) and $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ (see Figure 4) via VOLE-based ZK. VOLE-based $\mathcal{F}_{\mathsf{CPZK}}$ (QuickSilver [YSWW21]) is implemented as part of the EMP Toolkit [WMK16], and VOLE-based $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ [YH23] is open-sourced[2]. We used their implementations in an (almost) black-box manner. Following these implementations, we use the prime field $\mathbb{F}_{2^{61}-1}$.

[2] Available at https://github.com/gconeice/improved-zk-ram.

Baseline implementation. We compare our implementation to the prior state-of-the-art non-tight ZK CPU, Batchman [YHH+23]. Their implementation is open-sourced[3]. It is also a VOLE-based ZK protocol over $\mathbb{F}_{2^{61}-1}$.

Experiment setup. Unless otherwise specified, following our baseline [YHH+23], our experiments were executed over two AWS EC2 `m5.2xlarge` machines[4] 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.

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 IS.

We consider the following distributions of sizes of B instructions of 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.
- *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 $\frac{communication}{\#multiplication}$.

[3] Available at https://github.com/gconeice/stacking-vole-zk.

[4] 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
50	1	Balanced	109 K	334 K	438 K	102
	10		107 K	323 K	432 K	102
	20		108 K	342 K	459 K	102
100	20	Balanced	109 K	346 K	458 K	102
		Unbalanced	110 K	337 K	467 K	102
		Varied	109 K	340 K	460 K	102

Figure 13: 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 [YHH+23]	1.5 KHz	5.4 KHz	8.0 KHz	7.3 KB
Ours (Balanced)	0.6 KHz **0.56**×	2.7 KHz **0.51**×	3.7 KHz **0.46**×	12.7 KB
Ours (Balanced) Rounding Opt.	1.7 KHz **1.13**×	5.9 KHz **1.11**×	8.5 KHz **1.05**×	6.3 KB
Ours (Unbalanced)	10.6 KHz **6.90**×	32.5 KHz **6.07**×	43.8 KHz **5.45**×	1.0 KB

Figure 14: Comparison with Batchman [YHH+23]. 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 13 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 [YHH+23]. 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

Protocol	Network Bandwidth			Comm./Step
	100 Mbps	500 Mbps	1 Gbps	
Batchman [YHH+23]	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 15: Comparison with Batchman [YHH+23] 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 [YSWW21]	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 16: 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.

our rounding optimization) and unbalanced distributions. We tested the ZK CPUs by executing 500K steps.

Figure 14 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. Our ZK CPU communicates only $\approx$ 1KB per step.

Our speedup becomes more significant when considering instructions with larger differences in size; see Figure 15.

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 16 tabulates the results. Our ZK

Network Bandwidth	Total	Π_{ZKCPU}						
		Step 2 and Sub-step 7a	Step 3	Step 5	Sub-step 6a	Sub-step 6b ($\Pi_{\mathsf{CPZK\text{-}UROM}}$)	Sub-step 7b	Sub-step 7c
1 Gbps	30.1	7.1	3.2	$5e-4$	0.2	15.0	0.1	3.9
500 Mbps	38.3	10.4	4.7	$5e-4$	0.2	17.7	0.1	4.5
100 Mbps	123.7	39.1	19.1	$5e-4$	0.2	53.1	0.1	11.3

Figure 17: Fine-grained analysis of Π_{ZKCPU}. Sub-step 6b can be further decomposed as it includes hybrid calls to $\Pi_{\mathsf{CPZK\text{-}UROM}}$ (see Figure 18). Our ZK CPU was loaded with 50 instructions and 20 registers. It was executed 500K steps. The distribution over the size of instructions is varied, as factors of 10.

Network Bandwidth	Total	$\Pi_{\mathsf{CPZK\text{-}UROM}}$ (Sub-step 6b of Π_{ZKCPU})					
		Steps 5 and 6	Step 7	Sub-steps 8a and 8(b)i	Sub-step 8(b)ii	Sub-step 8(b)iii	Sub-step 8c
1 Gbps	15.0	3.9	0.3	$2e-4$	6.1	2.0	2.5
500 Mbps	17.7	5.5	0.3	$2e-4$	6.8	2.0	3.1
100 Mbps	53.1	19.7	0.3	$2e-4$	20.3	2.4	10.3

Figure 18: Fine-grained analysis of $\Pi_{\mathsf{CPZK\text{-}UROM}}$. Our ZK CPU was loaded with 50 instructions and 20 registers. It was executed 500K steps. The distribution over the size of instructions is varied, as factors of 10.

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 and matches our asymptotic analysis.

Microbenchmarks. We tested fine-grained execution time for our ZK CPU. We decomposed the entire execution time according to the main steps in Π_{ZKCPU}, which can be further decomposed to the main steps in $\Pi_{\mathsf{CPZK\text{-}UROM}}$. Figures 17 and 18 tabulate the results.

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

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, August 2014.

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Heidelberg.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 174–187. Springer, Heidelberg, August 1994.

[CGG+23] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. Sublonk: Sublinear prover plonk. Cryptology ePrint Archive, Paper 2023/902, 2023. https://eprint.iacr.org/2023/902.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

[DdSGOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of ram programs from any public-coin zero-knowledge system. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 615–638, Cham, 2022. Springer International Publishing.

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-Point Zero Knowledge and Its Applications. In Stefano Tessaro, editor, *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, volume 199 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:24, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[DOT21] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, November 2021.

[DXNT23] Zijing Di, Lucas Xia, Wilson Nguyen, and Nirvan Tyagi. Muxproofs: Succinct arguments for machine computation from tuple lookups. Cryptology ePrint Archive, Paper 2023/974, 2023. https://eprint.iacr.org/2023/974.

[FDNZ21] Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. Zero knowledge static program analysis. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2951–2967. ACM Press, November 2021.

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 178–191. ACM Press, November 2021.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[GGHAK22] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. Stacking sigmas: A framework to compose Σ-protocols for disjunctions. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 458–487. Springer, Heidelberg, May / June 2022.

[GHAK23] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. Dora: Processor expressiveness is (nearly) free in zero-knowledge for ram programs. Cryptology ePrint Archive, Paper 2023/1749, 2023. https://eprint.iacr.org/2023/1749.

[GHAKS23] Aarushi Goel, Mathias Hall-Andersen, Gabriel Kaptchuk, and Nicholas Spooner. Speed-stacking: Fast sublinear zero-knowledge proofs for disjunctions. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 347–378. Springer, Heidelberg, April 2023.

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[HK20] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[HLZ+24] Wenqing Hu, Tianyi Liu, Ye Zhang, Yuncong Zhang, and Zhenfei Zhang. Parallel zero-knowledge virtual machine. Cryptology ePrint Archive, Paper 2024/387, 2024. https://eprint.iacr.org/2024/387.

[HYDK21] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy*, pages 1538–1556. IEEE Computer Society Press, May 2021.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[Kol18] Vladimir Kolesnikov. Free IF: How to omit inactive branches and implement S-universal garbled circuit (almost) for free. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 34–58. Springer, Heidelberg, December 2018.

[KS22] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758, 2022. https://eprint.iacr.org/2022/1758.

[LAH+22] Ning Luo, Timos Antonopoulos, William R. Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving UNSAT in zero knowledge. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2203–2217. ACM Press, November 2022.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2968–2985. ACM Press, November 2021.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WYX+21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 501–518. USENIX Association, August 2021.

[YH23] Yibin Yang and David Heath. Two shuffles make a ram: Improved constant overhead zero knowledge ram. Cryptology ePrint Archive, Paper 2023/1115, 2023. https://eprint.iacr.org/2023/1115.

[YHH+23] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Batchman and robin: Batched and non-batched branching for interactive zk. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1452–1466, New York, NY, USA, 2023. Association for Computing Machinery.

[YHKD22] Yibin Yang, David Heath, Vladimir Kolesnikov, and David Devecsery. Ezee: Epoch parallel zero knowledge for ansi c. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 109–123. IEEE, 2022.

[YPHK23] Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. Towards generic mpc compilers via variable instruction set architectures (visas). In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2516–2530, New York, NY, USA, 2023. Association for Computing Machinery.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

[ZGK+18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy*, pages 908–925. IEEE Computer Society Press, May 2018.

SUPPLEMENTARY MATERIAL

A Deferred Complete Proofs

A.1 Complete Proof of Theorem 1

Theorem 1. Let the UROM be initialized by B non-zero-end vectors where the i-th vector is of length-$n^{(i)}$. Let the read-out vector be of length-n. Then, protocol $\Pi_{\mathsf{CPZK\text{-}UROM}}$ (Figures 9 and 10) UC-realizes $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ (Figure 8) in the $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid model (Figure 4) 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.

Proof. By constructing the simulator $\mathcal{S}$ for malicious $\mathcal{V}^*$ and malicious $\mathcal{P}^*$. Note that the Zero-Knowledge part of $\Pi_{\mathsf{CPZK\text{-}UROM}}$ is handled by the hybrid $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ directly. In particular, the instructions of the Zero-Knowledge part in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ is the *same* as $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$. Thus, the simulation for the instructions of the Zero-Knowledge part is straightforward. Here, we only focus on constructing the simulator for the unbalanced non-zero-end read-only memory part.

Malicious $\mathcal{V}^*$: Since $\mathcal{P}$ has *no* output, we only need to sample the transcripts seen by malicious $\mathcal{V}^*$. In particular, the simulator $\mathcal{S}$ interacts with $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, runs $\mathcal{V}^*$ as a subroutine, and emulates $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ for him as follows:

- For the instruction `InitUROM`: Since $\mathcal{P}$ is honest, the messages received by the real-world $\mathcal{V}^*$ are only some *cid*s and `true` for the `check` of $\mathcal{C}_0^{\mathsf{check}}$. Note that these *cid*s are either (1) revealed by $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ to $\mathcal{S}$; or (2) sampled by $\mathcal{P}$, which obviously can be sampled by $\mathcal{S}$. Furthermore, $\mathcal{C}_0^{\mathsf{check}}$ is determined by B only. Hence, $\mathcal{S}$ can trivially generate the *identical* messages for ideal $\mathcal{V}^*$ as interacting with a real-world $\mathcal{P}$.

- For the instruction `SetProg`: Since $\mathcal{P}$ is honest, in the real world, the messages received by $\mathcal{V}^*$ are only some *cid*s from $\mathcal{P}$ as well as an `open` to 1 and a `check` to `true` from $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$. Note that these *cid*s are revealed by $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ to $\mathcal{S}$ and $\mathcal{C}_1^{\mathsf{check}}$ is determined by n only, $\mathcal{S}$ can trivially generate the *identical* messages for $\mathcal{V}^*$ as interacting with a real-world $\mathcal{P}$.

- For the instruction `ReadUROM`: The messages received by $\mathcal{V}^*$ are only some *cid*s as $\mathcal{P}$ is honest. Note that these *cid*s are revealed by $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ to $\mathcal{S}$, $\mathcal{S}$ can trivially generate the *identical* messages for $\mathcal{V}^*$ as interacting with a real-world $\mathcal{P}$.

- For the instruction `CheckUROM`: Since $\mathcal{P}$ is honest, in the real-world, $\mathcal{V}^*$ will only receive `true` for each `check` in $\Pi_{\mathsf{CPZK\text{-}UROM}}$ coming from the hybrid $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ *regardless of $\mathcal{V}^*$'s challenge γ*. Thus, the only messages $\mathcal{S}$ needs to simulate for $\mathcal{V}^*$ are fresh *cid*s, which obviously can be sampled by $\mathcal{S}$. Note that the circuits $\mathcal{C}_2^{\mathsf{check}}, \mathcal{C}_3^{\mathsf{check}}$ are decided by $\mathcal{S}$-known γ, n.

Overall, the distributions between ideal/real are *identical*.

Malicious $\mathcal{P}^*$: The simulator $\mathcal{S}$ interacts with $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, runs $\mathcal{P}^*$ as a subroutine and emulates the hybrid $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ for her. In particular, the simulator will emulate a real-world honest $\mathcal{V}$ interacting with $\mathcal{P}^*$. Note that $\mathcal{V}$ has *no* input, so $\mathcal{S}$ can trivially emulate him. Furthermore, this implies that the crucial part is to simulate ideal $\mathcal{V}$'s output. In detail, $\mathcal{S}$ proceeds as follows:

- For the instruction `InitUROM`: If the emulated $\mathcal{V}$ outputs $(\texttt{initurom}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$, it implies that $\mathcal{P}^*$ uses these committed *cid*s to initialize the ZK UROM. Furthermore, each vector

committed by $\boldsymbol{u}^{(i\in[B])}$ *must* be non-zero-end since $\mathcal{C}_0^{\mathsf{check}}$ *must* output $\mathbf{0}$.[5] $\mathcal{S}$ can simply send $(\texttt{InitUROM}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ on behalf of $\mathcal{P}$. Note that $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ will *always* output the message $(\texttt{initurom}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ to the ideal $\mathcal{V}$, which is *identical* to the real-world $\mathcal{V}$. $\mathcal{S}$ can trivially send `abort` to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, if the emulated $\mathcal{V}$ halts.

- For the instruction `SetProg`:
  - If the emulated $\mathcal{V}$ halts, that means $\mathcal{P}^*$ uses a committed vector that is not a boundary string, then $\mathcal{S}$ can trivially send `abort` to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$.
  - If the emulated $\mathcal{V}$ does not halt, he must output $(\mathit{setprog}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$. Let $n \triangleq |\boldsymbol{cid}^{(p)}| = |\boldsymbol{cid}^{(id)}|$. Note that this means that $\boldsymbol{cid}^{(p)}$ must commit a valid boundary string, i.e., $\boldsymbol{p} \in \{0,1\}^{n-1}\|1$ since $\mathcal{C}_1^{\mathsf{check}}$ *must* output $\mathbf{0}$. Hence, $\mathcal{S}$ can simply send $(\mathit{SetProg}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ on behalf of $\mathcal{P}$ then the ideal-world $\mathcal{V}$ will always output $(\mathit{setprog}, \boldsymbol{cid}^{(p)}, \boldsymbol{cid}^{(id)})$, which is *identical* to the real-world $\mathcal{V}$.
- For the instruction `ReadUROM`: If the emulated $\mathcal{V}$ outputs $(\texttt{readurom}, \boldsymbol{cid}^{(d)})$, it implies that $\mathcal{P}^*$ uses the fresh *cids* in $\boldsymbol{cid}^{(d)}$. Since $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ for P^* and P^* sends $\boldsymbol{d}$ to $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$, $\mathcal{S}$ can trivially extract $\boldsymbol{d}$. $\mathcal{S}$ can then send $(\texttt{ReadUROM}, \boldsymbol{cid}^{(d)}, \boldsymbol{d})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ on behalf of $\mathcal{P}$. Note that $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ will *always* output the message $(\texttt{readurom}, \boldsymbol{cid}^{(d)})$ to the ideal $\mathcal{V}$, which is *identical* to the real-world $\mathcal{V}$. Note that f_{urom} inside $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ will be set to `cheating` according to $\boldsymbol{d}$, $\boldsymbol{p}$, $\boldsymbol{id}$ and logic inside $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$.
- For the instruction `CheckUROM`: $\mathcal{S}$ sends $(\texttt{CheckUROM})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ on behalf of $\mathcal{P}$. The emulated $\mathcal{V}$ can have the following potential outputs:
  - The emulated $\mathcal{V}$ outputs $(\texttt{checkurom}, \texttt{cheating})$: $\mathcal{S}$ sends `Cheat` to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$, which will set f_{urom} inside $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ to `cheating`. Thus, $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ will *always* output $(\texttt{checkurom}, \texttt{cheating})$ to the ideal $\mathcal{V}$, which is *identical* to the real-world $\mathcal{V}$.
  - The emulated $\mathcal{V}$ outputs $(\texttt{checkurom}, \texttt{honest})$: $\mathcal{S}$ does *not* sends `Cheat` to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ will then output $(\texttt{checkurom}, f_{\mathsf{urom}})$ to the ideal $\mathcal{V}$.
    - If $f_{\mathsf{urom}} = \texttt{honest}$, the ideal $\mathcal{V}$ output message is *identical* to the real-world $\mathcal{V}$.
    - If $f_{\mathsf{urom}} = \texttt{cheating}$, the ideal $\mathcal{V}$ output message is *different* from the real-world $\mathcal{V}$. *This is the only case where the ideal/real distribution differs!*

We now focus on analyzing the distributions between ideal/real. Indeed, the difference happens only when ideal-world $\mathcal{V}$ outputting $(\texttt{checkurom}, \texttt{cheating})$ while the real-world $\mathcal{V}$ outputting $(\texttt{checkurom}, \texttt{honest})$. We argue that this event only happens with negligible probability (i.e., the soundness error). Note that this event happens when $\mathcal{S}$ submits a malicious input to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ that sets f_{urom} to `cheating`. We perform case analysis over the reason why f_{urom} has been set to `cheating` (see Steps 2 and 4 in `ReadUROM` of $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$):

- **Case 1: $\boldsymbol{id}$ is not $[B]^n$ (see Step 2 in `ReadUROM` of $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$).** This will *not* happen because $\boldsymbol{id}$ is used as indexes to access a hybrid single-element ZK ROM (i.e., $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$) of size B (see Sub-step 8(b)i in `CheckUROM` of $\Pi_{\mathsf{CPZK\text{-}UROM}}$). Hence, if there exists an index that is *not* in $[B]$, the hybrid ZK ROM will catch it. I.e., $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$ will output $(\texttt{checkrom}, \texttt{cheating})$ to

[5] Of course, certain soundness must be added when instantiating the ZK part in $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$. Here, we work in the $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid model.

the real-world $\mathcal{V}$. Therefore, the real-world $\mathcal{V}$ *must* output ($\texttt{checkurom}, \texttt{cheating}$) (see Sub-step 8(b)iii in $\texttt{CheckUROM}$ of $\Pi_{\textsf{CPZK-UROM}}$).

- **Case 2: there exists $\boldsymbol{x} \in \textsf{Partition}(\boldsymbol{p}, \boldsymbol{d})$ such that $\textsf{last}(\boldsymbol{x})$ is 0 (see Step 4 in $\texttt{ReadUROM}$ of $\mathcal{F}_{\textsf{CPZK-UROM}}$).** This will *not* happen because the $\texttt{check}$ of C_2 in $\Pi_{\textsf{CPZK-UROM}}$ *must* output $\texttt{false}$ (see Step 7 in $\texttt{CheckUROM}$ of $\Pi_{\textsf{CPZK-UROM}}$). Specifically, let α be the index of $\textsf{last}(\boldsymbol{x})$ in $\boldsymbol{d}$, to let check of C_2 output $\texttt{true}$, $\mathcal{P}^*$ has to find some $inv \in \mathbb{F}$ such that $inv \cdot d_\alpha = p_\alpha$. This is impossible since $d_\alpha = 0$ and $p_\alpha = 1$.

- **Case 3: there exists $\boldsymbol{x} \in \textsf{Partition}(\boldsymbol{p}, \boldsymbol{d})$, $\boldsymbol{y} \in \textsf{Partition}(\boldsymbol{p}, \boldsymbol{id})$ pair[6] such that $\boldsymbol{x}$ is not equal to the vector committed by $\boldsymbol{u}^{(\textsf{last}(\boldsymbol{y}))}$ (see Step 4 in $\texttt{ReadUROM}$ of $\mathcal{F}_{\textsf{CPZK-UROM}}$).** Let the kth pair be the first place where this happens and assume $|\boldsymbol{x}^{(k)}| = |\boldsymbol{y}^{(k)}| = m$. Note that this means that any pair $\boldsymbol{x}^{(j)}$, $\boldsymbol{y}^{(j)}$ where $j \in [k-1]$, $\boldsymbol{x}^{(j)}$ is equal to the vector committed by $\boldsymbol{u}^{(\textsf{last}(\boldsymbol{y}^{(j)}))}$. Let $q \triangleq \sum_{j\in[k-1]} |\boldsymbol{x}^{(j)}|$. This implies

$$\sum_{j=1}^{q} d_j \cdot s_j = \sum_{j=1}^{q} p_j \cdot smac_j \tag{6}$$

  where $\boldsymbol{s} = \textsf{Expand}_2(\boldsymbol{p}, \gamma)$ and $smac_j = mac^{(id_j)}$ (see Sub-step 8a in $\texttt{ReadUROM}$ of $\mathcal{F}_{\textsf{CPZK-UROM}}$ for definition of mac). Note, if $smac_j \neq mac^{(id_j)}$, the real-world $\mathcal{V}$ must output ($\texttt{checkurom}$, $\texttt{cheating}$) since he *must* receive ($\texttt{checkrom}, \texttt{cheating}$) from $\mathcal{F}_{\textsf{CPZK-ROM}}$-hybrid (see Sub-steps 8(b)ii and 8(b)iii in $\texttt{ReadUROM}$ of $\mathcal{F}_{\textsf{CPZK-UROM}}$). Now, consider the probability that the real-world $\mathcal{V}$ outputs ($\texttt{checkurom}, \texttt{honest}$), it implies that $\mathcal{V}$ must receive $\texttt{true}$ for the $\texttt{check}$ of $\mathcal{C}_3^{\textsf{check}}$. This further implies that

$$\sum_{j=1}^{q+m} d_j \cdot s_j = \sum_{j=1}^{q+m} p_j \cdot smac_j \tag{7}$$

  Subtracting Equation (7) by Equation (6) we have

$$\begin{aligned} \sum_{j=q+1}^{q+m} d_j \cdot s_j &= \sum_{j=q+1}^{q+m} p_j \cdot smac_j \\ \Leftrightarrow \langle (1, \ldots, \gamma^{m-1}), \boldsymbol{x}^{(k)} \rangle &= mac^{\textsf{last}(\boldsymbol{y}^{(k)})} \\ \Leftrightarrow \langle (1, \ldots, \gamma^{m-1}), \boldsymbol{x}^{(k)} \rangle &= \langle (1, \ldots, \gamma^{n^{(\textsf{last}(\boldsymbol{y}^{(k)}))}-1}), \boldsymbol{v} \rangle \end{aligned}$$

  where $\boldsymbol{v}$ is the vector committed by $\boldsymbol{u}^{(\textsf{last}(\boldsymbol{y}^{(k)}))}$. Since $\boldsymbol{v} \neq \boldsymbol{x}^{(k)}$ and are both non-zero-end, the equality holds with probability up to $\frac{\max\{m, n^{(\textsf{last}(\boldsymbol{y}^{(k)}))}\}-1}{|\mathbb{F}|}$ based on Corollary 1 conditioned over $\gamma \in_\$ \mathbb{F}$.

To sum up, the overall soundness error is up to $\frac{\max\{n, n^{(1)}, \ldots, n^{(B)}\}-1}{|\mathbb{F}|}$. This finishes our proof. □

[6] There are n pairs in total.

A.2 Complete Proof of Theorem 2

Theorem 2. Protocol Π_{ZKCPU} (Figures 11 and 12) UC-realizes $\mathcal{F}_{\mathsf{ZKCPU}}$ (Figure 6) in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid model (Figure 8) with soundness error $\frac{2n+2m+1}{|\mathbb{F}|}$ and perfect zero-knowledge, in the presence of a static unbounded adversary.

Proof. By constructing the simulator $\mathcal{S}$ for malicious $\mathcal{V}^*$ and malicious $\mathcal{P}^*$. Note that the (ZK) simulator for malicious $\mathcal{V}^*$ is trivial as $\mathcal{V}^*$ has no input. In particular, the only messages $\mathcal{V}^*$ received that are not revealed by $\mathcal{F}_{\mathsf{ZKCPU}}$ are just some *cid*s, which can be trivially sampled by $\mathcal{S}$. Indeed, a malicious $\mathcal{V}^*$ can select arbitrary $\chi \in \mathbb{F}$ as the random challenge. However, χ is *independent* of $\mathcal{V}^*$'s transcripts since they always include `true` for each `check` since $\mathcal{P}$ is honest. Note that $\mathcal{S}$ has enough information to build $\mathcal{C}_4^{\mathsf{check}}, \mathcal{C}_5^{\mathsf{check}}, \mathcal{C}_6^{\mathsf{check}}$ including $n, m, \chi, \boldsymbol{st}^{(0)}, \boldsymbol{st}^{(final)}$. Henceforth, we only focus on constructing $\mathcal{S}$ for $\mathcal{P}^*$.

Malicious $\mathcal{P}^*$: The simulator $\mathcal{S}$ interacts with $\mathcal{F}_{\mathsf{ZKCPU}}$, run $\mathcal{P}^*$ as a subroutine and emulates the hybrid $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ for her. In particular, the simulator will emulate a real-world honest $\mathcal{V}$ interacting with $\mathcal{P}^*$. Note that $\mathcal{V}$ has *no* input, so $\mathcal{S}$ can trivially emulate him. Furthermore, this implies that the crucial part is to simulate ideal $\mathcal{V}$'s output. In detail, $\mathcal{S}$ proceeds as follows: if the emulated $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$, the simulator can trivially configure $\mathcal{F}_{\mathsf{ZKCPU}}$ to output $(\texttt{prove}, \texttt{false}, n)$ to the ideal $\mathcal{V}$ (see Step 4 in Figure 6); if the emulated $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{true}, n)$, this means the $\mathcal{P}^*$ past all checks in Π_{ZKCPU}. Now, since $\mathcal{P}^*$ has submitted all commitments via $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ (emulated by $\mathcal{S}$), the $\mathcal{S}$ can trivially extract $\boldsymbol{in}$ (in Step 2) and $\boldsymbol{p}, \boldsymbol{id}$ (in Step 3). Note that the emulated $\mathcal{V}$ outputting `true` implies the following:

1. $\boldsymbol{p}$ must be a length-n boundary string. If not, the `SetProg` instruction in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ hybrid *must* catch it (see Sub-step 6a), and $\mathcal{V}$ should output `false`.

2. $\boldsymbol{id}$ must be $[B]^n$. If not, the `ReadUROM` instruction in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ hybrid *must* catch it (see Sub-step 6b), and $\mathcal{V}$ should output `false`.

3. $\boldsymbol{d}$ must be $\boldsymbol{v}^{(i_1)} \| \cdots \| \boldsymbol{v}^{(i_\tau)}$ where $\boldsymbol{v}^{(j \in [B])}$ is the topology vector of the jth instruction (see Step 5) and $(i_1, \ldots, i_\tau) \triangleq \mathsf{Filter}(\boldsymbol{p}, \boldsymbol{id})$. If not, the `ReadUROM` instruction in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ hybrid *must* catch it (see Sub-step 6b), and $\mathcal{V}$ should output `false`.

Our $\mathcal{S}$ will send $(\texttt{Prove}, \tau \triangleq \mathsf{HW}(\boldsymbol{p}), i_1, \ldots, i_\tau, \boldsymbol{in})$ to $\mathcal{F}_{\mathsf{ZKCPU}}$ (more specifically, $\boldsymbol{in}$ will be broken into τ peaces trivially). Now, it suffices to argue that $\mathcal{F}_{\mathsf{ZKCPU}}$ will send `false` to the ideal $\mathcal{V}$ with *negligible* probability (i.e., the soundness). To see this, note the following additional facts (recall, conditioned over the emulated $\mathcal{V}$ outputting `true`):

4. $\boldsymbol{l} \odot \boldsymbol{r}$ must be equal to $\boldsymbol{o}$. If not, the `Check` instruction in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ hybrid of $\mathcal{C}_4^{\mathsf{check}}$ *must* catch it (see Sub-step 7a), and $\mathcal{V}$ should output `false`.

5. $\mathsf{Filter}(\boldsymbol{p}, \boldsymbol{o})$ must be a length-τ 0-vector. If not, the `Check` instruction in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ hybrid of $\mathcal{C}_5^{\mathsf{check}}$ *must* catch it (see Sub-step 7b), and $\mathcal{V}$ should output `false`.

6. The following two values *must* be equal:

$$\boldsymbol{s} \times \boldsymbol{M} \times \left(1, \boldsymbol{st}^{(0)}, in_1, o_1, \ldots, in_n, o_n\right)^T =$$

$$\boldsymbol{s} \times \left(\ell_1, r_1, \ldots, \ell_n, r_n, 1, 1, 1, \boldsymbol{st}_1^{(final)}, \ldots, 1, \boldsymbol{st}_m^{(final)}\right)^T$$

where $\boldsymbol{s} \triangleq (1, \chi, \ldots, \chi^{2n+2m+1})$. If not, the `Check` instruction in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ hybrid of $\mathcal{C}_6^{\mathsf{check}}$ *must* catch it (see Sub-step 7c), and $\mathcal{V}$ should output `false`.

If the ideal-world $\mathcal{V}$ outputs `false`, it means that extracted witness by $\mathcal{S}$ does not transfer $\boldsymbol{st}^{(0)}$ to $\boldsymbol{st}^{(final)}$. Conditioned over the above facts 1-5, it implies the following *inequality*:

$$\boldsymbol{M} \times \left(1, \boldsymbol{st}^{(0)}, in_1, o_1, \ldots, in_n, o_n\right)^T \neq$$
$$\left(\ell_1, r_1, \ldots, \ell_n, r_n, 1, 1, 1, \boldsymbol{st}_1^{(final)}, \ldots, 1, \boldsymbol{st}_m^{(final)}\right)^T$$

Since two unequal vectors are of length-$(2n + 2m + 2)$, the probability that the fact 6 happens will be bounded by $\frac{2n+2m+1}{|\mathbb{F}|}$, conditioned over $\chi \in_\$ \mathbb{F}$, based on the *Schwartz–Zippel lemma*. This finishes our proof. □

B Fine-grained Cost Analysis

B.1 Cost Analysis for $\Pi_{\mathsf{CPZK\text{-}UROM}}$ in $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid

The cost of $\Pi_{\mathsf{CPZK\text{-}UROM}}$ in $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid includes:

- `InitUROM` requires (1) B `Commit` hybrid calls; (2) 1 `Check` hybrid call with the circuit $\mathcal{C}_0^{\mathsf{check}}$; and (3) $\sum_{i\in[B]} n^{(i)}$ *cid*s from $\mathcal{P}$ to $\mathcal{V}$. Besides, $\mathcal{P}$ needs to compute B multiplicative inverses in $\mathbb{F}$.

- `SetProg` requires (1) 1 `Open` hybrid call; (2) 1 `Check` hybrid call with the circuit $\mathcal{C}_1^{\mathsf{check}}$; and (3) n *cid*s from $\mathcal{P}$ to $\mathcal{V}$.

- `ReadUROM` requires n `Commit` hybrid calls.

- `CheckUROM` requires (0) $\mathcal{V}$ to send γ to $\mathcal{P}$; (1) n `Commit` hybrid calls; (2) 1 `Check` hybrid call with the circuit $\mathcal{C}_2^{\mathsf{check}}$; (3) B `Linear` hybrid calls where the ith call is of length $n^{(i)} + 1$; (4) 1 `InitROM` hybrid call to initialize a size-B *balanced* ROM; (5) 1 `ReadROM` hybrid call to read n elements from the hybrid *balanced* ROM; (6) 1 `CheckROM` hybrid call; and (7) 1 `Check` hybrid call with the circuit $\mathcal{C}_3^{\mathsf{check}}$.

- Note that the check of circuit $\mathcal{C}_3^{\mathsf{check}}$ requires $\mathcal{P}$ and $\mathcal{V}$ to generate the commitments of $\boldsymbol{s} \triangleq \mathsf{Expand}_2(\boldsymbol{p}, \gamma)$. This can be achieved by (1) n `Commit` hybrid calls to commit $\boldsymbol{s}$; and (2) 1 `Check` hybrid call with the circuit to define Expand_2, which has $n - 1$ multiplications.

Hence, we tally the *overall* optimized cost of $\Pi_{\mathsf{CPZK\text{-}UROM}}$ (in $\mathcal{F}_{\mathsf{CPZK\text{-}ROM}}$-hybrid), where 4 instructions are all executed. Let $\varepsilon_{\mathsf{urom}} \triangleq \mathsf{gcd}(n^{(1)}, \ldots, n^{(B)})$, the cost includes:

- $\mathcal{V}$ sends γ to $\mathcal{P}$.

- $n + \frac{n}{\varepsilon_{\mathsf{urom}}}$ `Commit` hybrid calls.

- $\mathcal{O}\left(\sum_{i\in[B]} n^{(i)}\right)$ field operations to compute $\boldsymbol{mac}$.

- B `Linear` hybrid calls to commit *constants*.

- 1 `Open` hybrid call.
- 1 `InitROM` hybrid call to initialize a size-B hybrid (balanced) ROM.
- 1 `ReadROM` hybrid call to read $\frac{n}{\varepsilon_{\mathsf{urom}}}$ elements from the hybrid (balanced) ROM.
- 1 `CheckROM` hybrid call.
- 3 `Check` hybrid calls with circuits $\mathcal{C}_{1/2}^{\mathsf{check}}$ and Expand_2. Each circuit has $\frac{n}{\varepsilon_{\mathsf{urom}}}$ multiplications.
- 1 `Check` hybrid call with $\mathcal{C}_3^{\mathsf{check}}$ (of $n + \frac{2n}{\varepsilon_{\mathsf{urom}}}$ multi.).
- $\frac{n}{\varepsilon_{\mathsf{urom}}}$ *cid*s from $\mathcal{P}$ to $\mathcal{V}$.

B.2 Cost Analysis for Π_{ZKCPU} in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid

The cost of Π_{ZKCPU} in $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid includes:

- Step 1 requires $\mathcal{P}$ to send n to $\mathcal{V}$.
- Step 2 requires $4n$ `Commit` hybrid calls.
- Step 3 requires $2n$ `Commit` hybrid calls.
- Step 4 requires $\mathcal{V}$ to send χ to $\mathcal{P}$.
- Step 5 requires 1 *free* `InitUROM` (and *free* `Linear`) hybrid call since vectors $\boldsymbol{v}^{(i\in[B])}$ are public. To compute each $\boldsymbol{v}^{(i\in[B])}$, $\mathcal{P}$ and $\mathcal{V}$ need to cost $\mathcal{O}\left(\sum_{i\in[B]} n^{(i)}\right)$ field operations. In particular, they can "evaluate the circuit backward" (see [YHH+23] for detail). Note that the length of $\boldsymbol{v}^{(i\in[B])}$ would be $2n^{(i)}$.
- Step 6 requires (1) 1 `SetProg` hybrid call to set up the length-$2n$ vector to read from the UROM[7]; (2) 1 `ReadUROM` hybrid call to read the length-$2n$ vector $\boldsymbol{d}$ from UROM; and (3) 1 `CheckUROM` hybrid call.
- Step 7 requires (1) 1 `Check` hybrid call with circuit $\mathcal{C}_4^{\mathsf{check}}$ of n multiplications; (2) 1 `Check` hybrid call with circuit $\mathcal{C}_5^{\mathsf{check}}$ of n multiplications; and (3) 1 `Check` hybrid call with circuit $\mathcal{C}_6^{\mathsf{check}}$ of $4n$ multiplications. Note that to construct the commitments of $\boldsymbol{s}$ (see Sub-step 7c), $\mathcal{P}$ needs to make (1) $2n$ `Commit` hybrid calls to commit $\boldsymbol{s}$; and (2) 1 `Check` hybrid call with the circuit to define Expand_1, which has $2n$ multiplications.

[7] Note that the commitment indexes of $\boldsymbol{id}$ no longer needs to be transferred from $\mathcal{P}$ to $\mathcal{V}$. I.e., $\mathcal{P}$ and $\mathcal{V}$ already agree them in Step 3.