Constant Overhead Zero Knowledge for RAM Programs

Nicholas Franzese
Northwestern University
Evanston, USA
nicholasfranzese2026@u.northwestern.edu

Jonathan Katz
University of Maryland
College Park, USA
jkatz2@gmail.com

Steve Lu
Stealth Software Technologies, Inc.
Los Angeles, USA
steve@stealthsoftwareinc.com

Rafail Ostrovsky
UCLA
Los Angeles, USA
rafail@cs.ucla.edu

Xiao Wang*
Northwestern University
Evanston, USA
wangxiao@cs.northwestern.edu

Chenkai Weng
Northwestern University
Evanston, USA
ckweng@u.northwestern.edu

ABSTRACT

We show a *constant-overhead* interactive zero-knowledge (ZK) proof system for RAM programs, that is, a ZK proof in which the communication complexity as well as the running times of the prover and verifier scale linearly in the size of the memory N and the running time T of the underlying RAM program. Besides yielding an asymptotic improvement of prior work, our implementation gives concrete performance improvements for RAM-based ZK proofs. In particular, our implementation supports ZK proofs of private read/write accesses to 64 MB of memory (2^{24} 32-bit words) using only 34 bytes of communication per access, a more than 80× improvement compared to the recent BubbleRAM protocol. We also design a lightweight RISC CPU that can efficiently emulate the MIPS-I instruction set, and for which our ZK proof communicates only $\approx$ 320 bytes per cycle, more than 10× less than the BubbleRAM CPU. In a 100 Mbps network, we can perform zero-knowledge executions of our CPU (with 64 MB of main memory and 4 MB of program memory) at a clock rate of 6 6 KHz.

CCS CONCEPTS

• **Theory of computation** → **Cryptographic protocols**.

KEYWORDS

Zero-knowledge proofs

ACM Reference Format:
Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang*, and Chenkai Weng. 2021. Constant-Overhead Zero-Knowledge for RAM Programs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3484800

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484800

1 INTRODUCTION

Zero-knowledge (ZK) proofs enable a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that the prover knows a witness w on which a particular program P evaluates to 1 without revealing anything additional about w. A series of works over the past several years (e.g., [1, 7, 9, 13, 15, 20, 21, 23, 27, 28]) has shown several highly efficient ZK protocols, however for the most part these improved protocols have focused on the case where the program P is represented as a boolean or arithmetic circuit. This makes such proof systems somewhat difficult to apply to the arguably more natural setting where P is a program intended to be run on a general-purpose CPU, that is, when P is represented as a program in the *random-access machine (RAM)* model of computation. Although any such program can be converted to a circuit, doing so can be challenging and time consuming; more importantly, it can lead to sub-optimal performance as a general RAM program running in time T and using memory of size N requires a circuit of size $\Theta(TN)$ to verify its execution.

Some prior work has shown ZK proofs in the RAM model of computation. Hu, Mohassel, and Rosulek [19], and subsequently Mohassel, Rosulek and Scafuro [22], proposed an approach in which the addresses of memory accesses are revealed to the verifier; to account for that, they use oblivious RAM to make the original RAM program oblivious. Their focus was on asymptotic performance, and to the best of our knowledge their protocols have never been implemented. The TinyRAM framework [3, 4, 24] avoids the use of oblivious RAM by using a routing network to ensure consistency of memory accesses. Recent work by Block et al. [5, 6] reduces the memory overhead of the TinyRAM protocol but does not report concrete computational efficiency. Heath and Kolesnikov adapted the routing-network approach and integrated it with garbled-circuit ZK protocols [12, 17, 20] to develop BubbleRAM [16] and BubbleCache [18], which allow for ZK proofs about the execution of a lightweight, general-purpose CPU. All these approaches introduce overhead of at least $\Omega(\log N)$ per memory access performed by the underlying RAM program, either due to the use of oblivious RAM or due to the cost of the routing network. Moreover, the concrete efficiency of the state-of-the-art leaves room for improvement; for example, when memory is of size $N = 2^{24}$ with a 32-bit word size, BubbleRAM requires over 2900 bytes of communication per memory access, and BubbleCache (which introduces a noticeable, data-dependent probability of failure) requires 240 bytes of communication. Bootle et al.[8] can achieve better than $\log N$ overhead,

[*] Corresponding author

but their approach is not concretely more efficient than the schemes discussed above and cannot achieve constant overhead.

1.1 Contributions

In this work, we propose a different approach to RAM-based zero knowledge that offers both asymptotic and concrete efficiency improvements relative to prior work. Specifically, we avoid the log N overhead per memory access present in all prior work, and achieve *constant-overhead* (interactive) ZK proofs, i.e., ZK proofs in which the communication complexity, as well as the running times of the prover and verifier, are $O(N \quad T)$. Our approach yields not only an asymptotic improvement on prior work but, as we will discuss, concrete efficiency improvements as well.

The key novelty of our approach is that it ensures consistency of memory accesses using a *polynomial equality check* rather than a sorting network, while still avoiding the need for oblivious RAM. We illustrate the idea by focusing on ZK proofs for the simplified functionality of read-only array access. This functionality allows the prover to commit to the elements of an array M and then read those elements and prove statements about their values.[1] As a simple example, consider the case where the prover wants to prove in zero knowledge to the verifier that there exists an index i for which $M_i = t$ (where t is a public value). The protocol in this case roughly proceeds as follows:

(1) The prover commits to the list of values $\mathcal{L} = \quad (0, M_0), (1, M_1), \quad ,$ $(N \quad 1, M_{N \quad 1}))$. (This can be done once-and-for-all, and before t is known.)
(2) The prover commits to (i, t), where t is known to the verifier but i is not, and appends (i, t) to $\mathcal{L}$.
(3) The prover then sorts the tuples in $\mathcal{L}$ by their first entry, giving an updated list $\mathcal{L}'$, and commits to the tuples in that list.
(4) The prover then proves two things: (1) that $\mathcal{L}'$ is consistent, namely, that if two tuples in $\mathcal{L}'$ agree in their first entry, then they also agree in their second entry, and (2) that $\mathcal{L}'$ is a permuted version of $\mathcal{L}'$. The first step can be done in the natural way by comparing all adjacent entries in $\mathcal{L}'$, and we omit the details here. The second step relies on the polynomial equality check mentioned earlier. Specifically, let $\mathcal{L} = \quad x_0, \quad)$ denote the tuples in $\mathcal{L}$ (where now we represent each tuple as a single field element), and let $\mathcal{L}' = \quad x'_0, \quad)$ denote the tuples in $\mathcal{L}'$. If we define the polynomials $L(R) = \prod_i (x_i \quad R)$ and $L'(R) = \prod_i (x'_i \quad R)$, then note that $\mathcal{L}$ and $\mathcal{L}'$ are permutations of each other iff $L(R) = L'(R)$. The verifier can efficiently test equality of these polynomials by choosing a uniform field element r and verifying that $L(r) = L(r')$.

We show how to extend the above ideas to additionally handle write access to the array. We also show how to efficiently instantiate the above using recent VOLE-based ZK protocols for boolean/arithmetic circuits [2, 10, 27, 29] as a building block. In doing so we crucially rely on the fact that these VOLE-based ZK protocols support conversions between authenticated Boolean values and authenticated values in an extension field (i.e., $_{2^\kappa}$) "for free." Using polynomials to encode sets and checking polynomial equality in zero knowledge are both ideas that have been explored, in other contexts, in prior work. To the best of our knowledge, we are the first to apply them to RAM-based ZK, as well as to explore the efficiency advantages of implementing them using VOLE-based ZK protocols.

Our implementation enables proofs of private read/write access to an array of 2^{24} 32-bit elements (i.e., 64 MB of memory in total) using only 34 bytes of communication per access. This is more than an 80× improvement in communication compared to the previous state-of-the-art [16]. When running over a medium-bandwidth network, where communication is the bottleneck, this reduction in communication directly translates into a roughly 60× improvement in overall running time.

Our ZK protocol for private read/write array access is powerful enough for some applications that do not require the full expressiveness of RAM model computation. For example, we show a dedicated protocol for proving in zero knowledge that a committed string matches a public regular expression; our protocol runs in time linear in the length of the committed string. As another example, we show how to give a ZK proof of knowledge of a preimage of the scrypt hash function, a memory-hard hash function designed specifically so that its computation requires extensive random accesses to memory.

An efficient zero-knowledge processor. We use the above protocol to build a ZK proof of generic RAM computation. As already noted, this is interesting theoretically as the first *constant-overhead* ZK protocol in this setting. From a practical point of view, however, it is often more useful to consider the specific RAM program corresponding to a CPU that can then execute arbitrary RAM programs. With this in mind, we design a lightweight CPU with a simple (13-instruction) architecture that is nevertheless expressive enough to emulate most instructions in the MIPS-I instruction set within a few cycles. We can give zero-knowledge proofs about the execution of our CPU (with 64 MB of main memory and 4 MB of program memory) using about 320 bytes of communication per cycle. When run in a 100 Mbps network, our zero-knowledge CPU with 64 MB of memory executes at a clock rate of 6.6 KHz, more than an order of magnitude faster than the BubbleCache CPU that executes (for the same memory sizes) at a rate of 0 23 KHz in a 1 Gbps network.

2 PRELIMINARIES

We let κ be the security parameter. For a positive integer k, let $[k] = \{0, \quad , k \quad 1\}$. As convenient, we may express the entries of a length-N array M as $M_0, \quad , M_{N \quad 1}$ or $M[0], \quad , M[N \quad 1]$.

2.1 Information-Theoretic MACs

The ZK protocols we use in this paper are built on top of an information-theoretic MAC (IT-MAC) used by the verifier $\mathcal{V}$ to authenticate values known to the prover $\mathcal{P}$. The verifier $\mathcal{V}$ has a global authentication key $\quad \in \quad _{2^\kappa}$. A bit x known to $\mathcal{P}$ is authenticated by having $\mathcal{P}$ obtain a uniform $M_x \in \quad _{2^\kappa}$ and having $\mathcal{V}$ obtain $K_x \in \quad _{2^\kappa}$ such that

$$K_x = M_x \oplus x$$

We view this as providing a "handle" for the underlying value, and let $[x]$ mean that $\mathcal{P}$ holds (x, M_x) and $\mathcal{V}$ holds K_x. We stress,

[1] While our primary motivation for realizing the read/write version of this functionality is ZK proofs of arbitrary RAM programs, the functionality is also interesting in its own right [11]. We provide further discussion below.

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

Inputs: On receiving (Input, x) from the prover, store x and send $[x]$ to each party.

Constants: On receiving (Const, x) from both parties, store x and send $[x]$ to each party. (If the inputs sent by the two parties do not match, both parties receive cheating.)

Boolean circuit satisfiability: On receiving $(\mathsf{Boolean}, C, [x_0], \ldots, [x_{n-1}])$ from both parties, where $x_i \in \mathbb{F}_2$ and C is a Boolean circuit, compute $b := C(x_0, \ldots, x_{n-1})$ and send b to $\mathcal{V}$.

Arithmetic circuit satisfiability: On receiving $(\mathsf{Arithmetic}, C, [\![x_0]\!], \ldots, [\![x_{n-1}]\!])$ from both parties, where $x_i \in \mathbb{F}_{2^\kappa}$ and C is an arithmetic circuit over $\mathbb{F}_{2^\kappa}$, compute $y := C(x_0, \ldots, x_{n-1})$. If $y = 1$ send 1 to $\mathcal{V}$; else send 0 to $\mathcal{V}$.

Figure 1: Ideal functionality for stateful circuit-based zero-knowledge proofs.

however, that two different handles for the same value are generated independently; thus, in particular, it is not possible for $\mathcal{V}$ to tell whether two handles are for the same value or not. These IT-MACs are XOR-homomorphic in that parties holding $[x]$ and $[y]$ can locally compute $[x \oplus y]$ by XORing values they already hold.

For $x = x_1 \cdots x_n \in \{0,1\}^n$, we overload the above notation by letting $[x] = [x_1], \ldots, [x_n]$. For the special case of $x \in \{0,1\}^\kappa$, we can alternatively view x as an element in the extension field $\mathbb{F}_{2^\kappa}$. In this case we extend the above IT-MAC in the natural way, i.e., by choosing uniform $M_x \in \mathbb{F}_{2^\kappa}$ and setting $K_x = M_x \oplus x \cdot \Delta$, where multiplication is in $\mathbb{F}_{2^\kappa}$. To distinguish this case, we write $[\![x]\!]$ when $x \in \mathbb{F}_{2^\kappa}$. These IT-MACs are similarly homomorphic with respect to addition in $\mathbb{F}_{2^\kappa}$, as well as addition/multiplication by a public constant. Importantly, parties holding authenticated bits $[x_0], \ldots, [x_{\kappa-1}]$ can locally convert these to the authenticated value $[\![x]\!]$. Specifically, if we fix a degree-κ irreducible polynomial $f(X)$ and identify $\mathbb{F}_{2^\kappa}$ with $\mathbb{F}_2[X]/(f(X))$, then we have $x = \sum_{i\in[\kappa]} x_i \cdot \mathsf{X}^i$, where $\mathsf{X} \in \mathbb{F}_{2^\kappa}$ denotes the element corresponding to $X \in \mathbb{F}_2[X]/(f(X))$. The parties can compute $[\![x]\!]$ by having the prover compute $M_x = \sum_{i\in[\kappa]} M_{x_i} \cdot \mathsf{X}^i$ and the verifier compute $K_x = \sum_{i\in[\kappa]} K_{x_i} \cdot \mathsf{X}^i$; we then have

$$\begin{aligned} M_x &= \sum_{i\in[\kappa]} M_{x_i} \cdot \mathsf{X}^i \\ &= \sum_{i\in[\kappa]} (K_{x_i} \oplus x_i \Delta) \cdot \mathsf{X}^i \\ &= \sum_{i\in[\kappa]} K_{x_i} \cdot \mathsf{X}^i \oplus \Big(\sum_{i\in[\kappa]} x_i \cdot \mathsf{X}^i\Big) \cdot \Delta \\ &= K_x \oplus x \cdot \Delta \end{aligned}$$

In short, the IT-MACs are also "homomorphic" with respect to bit packing. We write $[\![x]\!] := \mathsf{Pack}([x_0], \ldots, [x_{\kappa-1}])$ to represent this procedure.

2.2 Stateful) Circuit-Based Zero-Knowledge Proofs

We rely on stateful zero-knowledge proofs for Boolean and arithmetic circuits. By "stateful" we mean that the prover can commit to values and then repeatedly use those values for different zero-knowledge proofs. We provide the relevant functionality in Figure 1. The functionality can be instantiated using several existing protocols, but in our implementation we use the recent VOLE-based ZK protocols [2, 10, 27, 29] that have good concrete efficiency and that enable Boolean/arithmetic conversion (i.e., bit packing) for free. The most-recent such protocols [2, 29] have very low communication: for a Boolean (resp., arithmetic) circuit C with $|C|$ AND gates (resp., multiplication gates), the communication complexity is only $|C|$ bits (resp., field elements). An important optimization introduced by QuickSilver [29] applies when C can be expressed as a polynomial of low (total) degree. In this case, the communication complexity depends on the degree of the polynomial but not the number of multiplications.

2.3 RAM-Based Computation

Our model for RAM-based computation is based on the one of Gordon et al. [14]. A RAM program is defined by a "next-instruction circuit" Π that, given its current state and a value d, outputs the next instruction to execute along with updated state. Thus, an execution of RAM program Π when the memory M is initialized to $M_0, \ldots, M_{N-1} \in \{0,1\}^W$ (we assume for simplicity that W and N are hard-coded in Π) proceeds by setting $\mathsf{st} := \mathsf{start}$ and $d := 0^W$ and then until termination doing:

- Compute $(op, \ell, d', \mathsf{st}') := \Pi(\mathsf{st}, d)$, and set $\mathsf{st} := \mathsf{st}'$. Then:
  - If $op = \mathsf{Stop}$, terminate with output d'.
  - If $op = \mathsf{Read}$, set $d := M_\ell$.
  - If $op = \mathsf{Write}$, set $M_\ell := d'$ and $d := d'$.

We refer to each iteration of the above loop as a *cycle*, and express the entire computation above as $\Pi(M_0, \ldots, M_{N-1})$.

Π may represent a specific algorithm (e.g., binary search) executing in the RAM model. Alternately, it can be a general-purpose CPU that has the ability to execute arbitrary assembly code loaded into a portion of memory. In that case it is convenient to have the state st of Π consist of a *program counter pc* as well as an array of *registers R*. It is also useful to segment the memory into a *program memory M_p* holding the program (i.e., assembly code) to be executed, and a *main memory M_d* for storing data; for efficiency, one may treat M_p as being read-only (though this is not essential). More generally, it may be advantageous to adapt the above model in other ways (e.g., both reading from program memory and accessing main memory once per cycle), as well as to represent Π in other models of computation (e.g., using boolean circuits for some instructions and arithmetic circuits for others); we refer to Section 4 for further discussion in the context of a lightweight RISC CPU we design.

3 ZERO-KNOWLEDGE PROOFS IN THE RAM MODEL

In this section, we present zero-knowledge proofs in the RAM model. We begin by focusing on protocols for private array access. Roughly, these protocols allow a prover to commit to an array of

Functionality $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$

(In addition to the following, the functionality also supports all instructions in $\mathcal{F}_{\mathsf{ZK}}$.)

Initialization: On receiving $(\mathsf{Init}, W, N, [M_0], \ldots, [M_{N-1}])$ from $\mathcal{P}$ and $\mathcal{V}$, where $M_i \in \{0,1\}^W$, store the $\{M_i\}$ and set $f := \mathsf{honest}$.

Read: On receiving $(\mathsf{Read}, [\ell], d)$ from $\mathcal{P}$, and $(\mathsf{Read}, [\ell])$ from $\mathcal{V}$, send $[d]$ to each party. If $d \neq M_\ell$ or $\ell \geq N$ then set $f := \mathsf{cheating}$.

Check: Upon receiving (check) from $\mathcal{V}$ do: If $\mathcal{P}$ sends (cheat) then send $\mathsf{cheating}$ to $\mathcal{V}$. If $\mathcal{P}$ sends $(\mathsf{continue})$ then send f to $\mathcal{V}$.

Figure 2: Ideal functionality for private read-only array access.

values, and then (in the read-only case) read values from the array referenced by address handles, or (in the general case) read/write values from/to the array as dictated by an operation handle $[op]$ and an address handle $[\ell]$. In the read-only case, $\mathcal{V}$ does not learn the address being read; in the read/write case, $\mathcal{V}$ also does not learn the operation being performed or (in case of a write) the value being written. At the end of this section, we show how these protocols can be used to support ZK proofs of general RAM computation.

3.1 Read-Only Array Access

We begin by describing the case of read-only array access. A formal description of the relevant functionality in this case is given in Figure 2. Our protocol realizing this functionality in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model proceeds in three phases:

(1) **Initialization.** In this step, the prover initializes memory with contents $M_0, \ldots, M_{N-1}$ by generating the list of handles $\mathcal{L} = (([0],[M_0]), \ldots, ([N-1],[M_{N-1}]))$.
(2) **Read.** A read operation at the address represented by the handle $[\ell]$ is carried out by simply having the prover generate a handle for M_ℓ. (The prover knows all the values, so it can easily do this.) Both parties append the result (along with $[\ell]$) to $\mathcal{L}$. At this point, nothing prevents a cheating prover from using an inconsistent value $d \neq M_\ell$; any such cheating, however, will be caught in the check phase.
(3) **Check.** To check correctness of a sequence $\mathcal{L}$ of size k consisting of the initial memory contents and the results of a sequence of read accesses, the parties proceed as follows. $\mathcal{P}$ sorts the tuples in $\mathcal{L}$ by their addresses, from least to greatest, and uses $\mathcal{F}_{\mathsf{ZK}}$ to generate handles for a second sorted list $\mathcal{L}'$. Correctness of $\mathcal{L}$ can now be checked by verifying two things: (1) the tuples in $\mathcal{L}'$ are *consistent* in the sense that if $([\ell],[d]), ([\ell],[d'])$ are two tuples in $\mathcal{L}'$ with the same address ℓ, then $d = d'$, and (2) $\mathcal{L}'$ is a permuted version of $\mathcal{L}$. Now:
    (a) The first requirement can be verified using a sequence of $k-1$ verifications performed on adjacent tuples in $\mathcal{L}'$. Namely, for adjacent tuples $([\ell'_i],[d'_i])$ and $([\ell'_{i+1}],[d'_{i+1}])$ the verifier checks that either $\ell'_i < \ell'_{i+1}$ (i.e., the tuples are sorted correctly) or else $\ell'_i = \ell'_{i+1}$ and $d'_i = d'_{i+1}$ (i.e., the tuples are consistent). Note that each of these checks can be performed by a circuit whose size is independent of k.
    (b) The second requirement is verified as follows: first, we use bit-packing to map each tuple in $\mathcal{L}$ and $\mathcal{L}'$ to an element of $\mathbb{Z}_{2^\kappa}$. We thus obtain two lists $\mathcal{L} = (\llbracket x_0 \rrbracket, \ldots, \llbracket x_{k-1} \rrbracket)$ and $\mathcal{L}' = \left(\llbracket x'_0 \rrbracket, \ldots, \llbracket x'_{k-1} \rrbracket\right)$. Consider the formal polynomials $L(R) = \prod_i (x_i - R)$ and $L'(R) = \prod_i (x'_i - R)$; these polynomials are equal iff $\mathcal{L}$ and $\mathcal{L}'$ are permuted versions of each other. The verifier checks equality of $L(R)$ and $L'(R)$ by choosing a uniform value $r \in \mathbb{Z}_{2^\kappa}$ and then checking that $L(r) = L'(r)$.

The protocol is described formally in Figure 3.

Protocol $\Pi_{\mathsf{RO\text{-}ZKarray}}$

Parameters: Let N be the size of the array and let W be the bit-length of array elements, where $W \geq \lceil \log N \rceil \leq \kappa$.

All instructions in $\mathcal{F}_{\mathsf{ZK}}$ are handled in the natural way.

Initialization: $\mathcal{P}$ and $\mathcal{V}$ have a list of handles $\{[M_i]\}_{i\in[N]}$. Each party locally initializes $\mathcal{L}$ as an empty list. Then for $i \in [N]$, the parties do

(1) $\mathcal{P}$ and $\mathcal{V}$ send (Const, i) to $\mathcal{F}_{\mathsf{ZK}}$, which returns $[i]$ to each party. (If either party receives $\mathsf{cheating}$, they abort.)
(2) Each party locally appends $([i],[M_i])$ to $\mathcal{L}$.

Read: The parties hold $[\ell]$. $\mathcal{P}$ sets $d := M_\ell$ and sends (input, d) to $\mathcal{F}_{\mathsf{ZK}}$, which returns $[d]$ to each party. Then the two parties locally append $([\ell],[d])$ to $\mathcal{L}$.

Check: Let T be the number of reads performed. Each party holds a list $\mathcal{L} = (([\ell_0],[M_0]), \ldots)$ containing $k = N + T$ tuples.

(1) $\mathcal{P}$ sorts the tuples in $\mathcal{L}$ by their first entry, from least to greatest, giving a new list $((\ell'_0, M'_0), \ldots)$. For $i \in [k]$, $\mathcal{P}$ sends $(\mathsf{input}, \ell'_i)$ and (input, M'_i) to $\mathcal{F}_{\mathsf{ZK}}$, which returns $[\ell'_i]$ and $[M'_i]$ to each party. Each party forms the list $\mathcal{L}' = (([\ell'_0],[M'_0]), \ldots)$.
(2) For each $i \in [k-1]$, $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Boolean}, C, [\ell'_i], [\ell'_{i+1}], [M'_i], [M'_{i+1}])$ to $\mathcal{F}_{\mathsf{ZK}}$, where $C(\ell'_i, \ell'_{i+1}, M'_i, M'_{i+1})$ is a Boolean circuit that outputs 1 if and only if $\left((\ell'_i = \ell'_{i+1}) \wedge (M'_i = M'_{i+1})\right) \vee \left(\ell'_i < \ell'_{i+1}\right)$. Similarly, $\mathcal{P}$ proves that $[\ell'_{k-1}] < N$. If $\mathcal{F}_{\mathsf{ZK}}$ ever returns 0, then $\mathcal{V}$ outputs $\mathsf{cheating}$ and aborts.
(3) For $i \in [k]$, the parties locally compute $\llbracket x_i \rrbracket := \mathsf{Pack}([\ell_i],[M_i])$ and $\llbracket x'_i \rrbracket := \mathsf{Pack}([\ell'_i],[M'_i])$.
(4) $\mathcal{V}$ samples a uniform $r \in \mathbb{Z}_{2^k}$ and sends it to $\mathcal{P}$.
(5) Let C' be an arithmetic circuit for which $C'(a_0, \ldots, a_{k-1}, b_0, \ldots, b_{k-1}) = 1$ iff $\prod_{i\in[k]} a_i = \prod_{i\in[k]} b_i$. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Arithmetic}, C', \llbracket x_0 - r \rrbracket, \ldots, \llbracket x_{k-1} - r \rrbracket, \llbracket x'_0 - r \rrbracket, \ldots, \llbracket x'_{k-1} - r \rrbracket)$ to $\mathcal{F}_{\mathsf{ZK}}$. If $\mathcal{F}_{\mathsf{ZK}}$ returns 0, then $\mathcal{V}$ outputs $\mathsf{cheating}$. Otherwise, $\mathcal{V}$ outputs honest.

Figure 3: ZK proof for private read-only array access in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model.

Theorem 1. *For T read accesses, the protocol in Figure 3 securely realizes the private read-only array access functionality (cf. Figure 2) in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model with statistical error $(N+T)/2^{\kappa}$.*

Proof. The verifier does not have any input and only receives messages from the $\mathcal{F}_{\mathsf{ZK}}$ functionality. It is therefore straightforward to prove security for a corrupted verifier, and so we focus on the case of a corrupted prover $\mathcal{P}^*$. We define a simulator Sim interacting with the $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$ functionality. The simulator runs $\mathcal{P}^*$ as a subroutine and emulates $\mathcal{F}_{\mathsf{ZK}}$ for $\mathcal{P}^*$. It proceeds as follows:

- **Initialization:** Assuming $\mathcal{P}^*$ does not cheat in this step by sending an incorrect index, Sim sets $\mathcal{L} = \big(([\ell_0],[M_0]),\ldots,([\ell_{N-1}],[M_{N-1}])\big)$, where $[M_0],\ldots,[M_{n-1}]$ are handles defined by previous interactions of $\mathcal{P}^*$ with $\mathcal{F}_{\mathsf{ZK}}$.. The simulator sends (Init, W, N, $[M_0],\ldots,[M_{N-1}]$) to $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$.
- **Read:** On input $[\ell]$, let d be the value that $\mathcal{P}^*$ sends to $\mathcal{F}_{\mathsf{ZK}}$. The simulator then sends (Read, $[\ell]$, d) to $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$, and appends $([\ell],[d])$ to $\mathcal{L}$.
- **Check:** Let T be the number of reads performed, and let $k = N+T$. Then:
  1. The simulator obtains the sequence of values $\{\ell'_i\}_{i\in[k]}$ and $\{M'_i\}_{i\in[k]}$ that $\mathcal{P}^*$ sends to $\mathcal{F}_{\mathsf{ZK}}$. Let $\mathcal{L}' = \big(([\ell'_0],[M'_0]),\ldots,([\ell'_{k-1}],[M'_{k-1}])\big)$.
  2. For $i \in [k-1]$, the simulator obtains the message (Boolean, C, $[\ell'_i],[\ell'_{i+1}],[M'_i],[M'_{i+1}]$) that $\mathcal{P}^*$ sends to $\mathcal{F}_{\mathsf{ZK}}$. If C is not the correct circuit or $C(\ell'_i,\ell'_{i+1},M'_i,M'_{i+1}) = 0$, then the simulator sends (cheat) to $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$, outputs whatever $\mathcal{P}^*$ outputs, and aborts.
  3. Let $[\![x_i]\!] := \mathsf{Pack}([\ell_i],[M_i])$ and $[\![x'_i]\!] := \mathsf{Pack}([\ell'_i],[M'_i])$.
  4. The simulator chooses a uniform value $r \in \mathbb{F}_{2^k}$ and sends it to $\mathcal{P}^*$.
  5. Sim obtains the message (Arithmetic, C', $[\![x_0-r]\!],\ldots,[\![x_{k-1}-r]\!],[\![x'_0-r]\!],\ldots,[\![x'_{k-1}-r]\!]$) that $\mathcal{P}^*$ sends to $\mathcal{F}_{\mathsf{ZK}}$. If C' is not the correct circuit or $C'(x_0-r,\ldots,x_{k-1}-r,x'_0-r,\ldots,x'_{k-1}-r) \neq 1$, the simulator sends (cheat) to $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$. Otherwise, it sends (continue) to $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$. It outputs whatever $\mathcal{P}^*$ outputs and aborts.

We claim that the execution of the simulator in the ideal world is statistically indistinguishable from the execution of the protocol with $\mathcal{P}^*$ in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid world. To see this, note first that the view of $\mathcal{P}^*$ is perfectly simulated, and thus we need only consider the output of the honest $\mathcal{V}$. If $\mathcal{P}^*$ behaves honestly in every read operation (namely, if for input $[\ell]$ it always holds that $d = M_\ell$), then the output of $\mathcal{V}$ is the same in both the ideal world and the hybrid world. If there is at least one read operation in which $\mathcal{P}^*$ behaves dishonestly, then in the ideal-world execution $\mathcal{V}$ always outputs cheating and we need to argue that in the hybrid-world execution $\mathcal{V}$ would output cheating except with probability at most $(N+T)/2^{\kappa}$. There are two possibilities to consider:

- If $\mathcal{L}'$ is a permutation of $\mathcal{L}$, then there is some $i \in [k-1]$ for which $C(\ell'_i,\ell'_{i+1},M'_i,M'_{i+1}) = 0$. Hence in the hybrid-world execution $\mathcal{V}$ would also output cheating.
- If $\mathcal{L}'$ is not a permutation of $\mathcal{L}'$, then the polynomials $L(R) = \prod_i(x_i - R)$ and $L(R') = \prod_i(x'_i - R)$ are distinct degree-$(N+T)$ polynomials, and hence when r is uniform we have $L(r) \neq L'(r)$ except with probability at most $(N+T)/2^{\kappa}$. Thus, in the hybrid-world execution $\mathcal{V}$ would output cheating except with probability at most $(N+T)/2^{\kappa}$.

This completes the proof.

Concrete efficiency and practical considerations. The asymptotic running time of our protocol is $O(N+T)$. Concretely, the cost of the protocol is dominated by (1) authentication of $W+\kappa$ bits per read access (W bits for the initial read—assuming the address is already authenticated—and at most κ bits for the corresponding entry in $\mathcal{L}'$), (2) $O(N+T)$ zero-knowledge proofs for a Boolean circuit with $W+2\log N$ AND gates, and (3) a zero-knowledge proof for an arithmetic circuit with $2\cdot(T+N-1)$ multiplication gates over $\mathbb{F}_{2^{\kappa}}$. Using optimizations from the QuickSilver protocol [29] to instantiate $\mathcal{F}_{\mathsf{ZK}}$, the third part can be done using roughly $2(N+T)\kappa/h + O(h\kappa)$ bits of communication, where h is a parameter that can be adjusted to reduce communication at the cost of increased computation. (We set $h=4$ in our implementation.) See Section 5 for an analysis of the concrete performance of the protocol.

Support for larger data elements. The above protocol assumes $\log N + W \leq 128$, as we pack the index and payload as an element in $\mathbb{F}_{2^{\kappa}}$ and use $\kappa = 128$ in our implementation. To handle larger W, we could instead pack to an extension field $\mathbb{F}_{2^{k'}}$ with $\log N + W \leq k'$. However, the practical efficiency of doing so would be poor since hardware acceleration is only available for $\mathbb{F}_{2^{128}}$. To efficiently support larger W, we can instead use a universal hash function mapping to $\mathbb{F}_{2^{128}}$ whose key is chosen by the verifier after step 1 of the checking protocol. The probability that there exist two unequal elements in lists $\mathcal{L}$ and $\mathcal{L}'$ that hash to the same value is at most $k^2/2^{128}$. We efficiently implement this idea as follows. Say $\log N + W \leq 128c$ for some c. We use a hash function $H_\alpha : \mathbb{F}^{c}_{2^{128}} \to \mathbb{F}_{2^{128}}$, where $H_\alpha(X_1,\ldots,X_c) = \bigoplus \alpha_i \cdot X_i$ for $\alpha = (\alpha_1,\ldots,\alpha_c) \in \mathbb{F}^{c}_{2^{128}}$. Since evaluating this function only requires multiplication by public constants, it is essentially free in communication and cheap in computation.

3.2 Read/Write Array Access

We adapt the protocol from the previous section to also handle write access to the array. The relevant functionality is in Figure 4. To enforce memory consistency we now need to ensure that each value read at a particular memory address corresponds to the last value written to that address. (We require that a write is performed to an address before any read to that address.) We do this by including a timestamp along with every memory access that is appended to the list $\mathcal{L}$. After sorting $\mathcal{L}$ to give a sorted list $\mathcal{L}'$ as before, the parties verify for each pair of consecutive tuples (cf. Equation (1)) that (1) they are sorted correctly; (2) either the addresses in the two tuples are not equal, the value stored there is unchanged, or the second tuple in the pair corresponds to a Write operation; and, (3) the first memory access to an address is a Write operation. The protocol is formally described in Figure 5. The running time of the protocol is $O(T)$ rather than $O(N+T)$ as in the previous section, but this is because we assume the memory is empty at initialization.

Theorem 2. *Protocol* ZKarray *(Figure 5) securely realizes the private read/write array access functionality (cf. Figure 4) in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model with statistical error $T/2^{\kappa}$.*

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

(In addition to the following, the functionality also supports all instructions in $\mathcal{F}_{\mathsf{ZK}}$.)

Initialize memory: On receiving (Init, W, N, T) from both parties, initialize $M_0, \quad , M_{N\ \ 1}$ to $\perp$, and set $f :=$ honest.

Read/write access: At most T such accesses are supported. On receiving $(\mathsf{Access}, [op], [\ell], [d])$ from $\mathcal{P}$ and $\mathcal{V}$, if $\ell < N$ then:

- If $op =$ Read, send $[M_\ell]$ to each party. If $M_\ell = \perp$ then set $f :=$ cheating.
- If $op =$ Write, set $M_\ell := d$ and send (a fresh handle) $[d]$ to each party.

If $\ell \geq N$ then set $f :=$ cheating and send $[d]$ to each party.

Check: Upon receiving (check) from $\mathcal{V}$ do: If $\mathcal{P}$ sends (cheat) then send cheating to $\mathcal{V}$. If $\mathcal{P}$ sends (continue) then send f to $\mathcal{V}$.

Figure 4: Ideal functionality for private read/write array access.

Proof. The proof is substantially the same as that of Theorem 1, and so we focus on the checks implemented in step 2. Assuming $\mathcal{L}'$ is a permuted version of $\mathcal{L}$ (which is enforced by step 5 as in the proof of Theorem 1), we show that if the checks in step 2 all succeed then all the memory accesses in $\mathcal{L}$ are *consistent*, namely: (1) each memory address is written before it is read and (2) each read of a memory address returns the last value written there. To see this, first observe that the checks in step 2 imply that $\mathcal{L}'$ is correctly sorted, since for every pair of adjacent tuples $([\ell'_i], [t'_i], \star, \star)$ and $([\ell'_{i\ \ 1}], [t'_{i\ \ 1}], \star, \star)$ in $\mathcal{L}'$ it holds that $(\ell'_i < \ell'_{i\ \ 1}) \vee \ (\ell'_i = \ell'_{i\ \ 1}) \wedge (t'_i < t'_{i\ \ 1}))$. Moreover, every memory access is in bounds; this is because the parties check that the address of the final tuple in $\mathcal{L}'$ is less than N. Assume toward a contradiction that the consistency requirements are violated. Let $([\ell'_i], [t'_i], [\mathsf{Read}], [d'_i]) \in \mathcal{L}'$ be the tuple with the smallest value of the timestamp t violating one of the consistency requirements. (Note that each tuple in $\mathcal{L}$—and hence $\mathcal{L}'$—has a unique timestamp, so this tuple is uniquely defined.) There are two possibilities:

- If $\ell'_{i\ \ 1} \quad \ell'_i$ or $i = 0$, then the checks would fail since they enforce that the operation for any such tuple is a Write. (If $i = 0$ this is checked explicitly; for $i > 0$ this is because Equation (1) checks that if $\ell'_{i\ \ 1} \quad \ell'_i$ then the operation is a Write.)
- Otherwise, $\ell'_{i\ \ 1} = \ell'_i$. Then the checks would also fail since in this case they enforce that $d'_{i\ \ 1} = d'_i$ but $d'_{i\ \ 1}$ is equal to the previous value written to address ℓ'_i (by our assumption on minimality of t'_i and the fact that $\mathcal{L}'$ is sorted).

This completes the proof.

Concrete efficiency. The optimizations discussed for the read-only setting all apply here. Concretely, the protocol involves: (1) authentication of $1 \quad \log N \quad W \quad \kappa$ bits per access ($1 \quad \log N \quad W$ bits for the inputs of private array access—note that the counter t is known to both parties—and at most κ bits when committing to the corresponding entry in $\mathcal{L}'$), (2) $O(T)$ zero-knowledge proofs for a Boolean circuit with $5 \quad 2\log N \quad W \quad \log T$ AND gates, and (3) a zero-knowledge proof for an arithmetic circuit with $2 \cdot (T \quad 1)$ multiplication gates over $\ _{2^\kappa}$. As in the previous section, we use the technique from QuickSilver [29] to reduce the cost of the third step to roughly $\kappa/2$ bits of communication per access.

Protocol ZKarray

Parameters: Let N be the size of the array, let W be the bit-length of array elements, and let T be an upper bound on the number of memory accesses, where $W \quad \lceil \log N \rceil \quad \lceil \log T \rceil \leq \kappa$.

Initialize memory: Each party initializes $\mathcal{L}$ as an empty list, and initializes a counter $t := 0$ represented using $\lceil \log T \rceil$ bits.

Read/write access: The parties begin holding $[op]$, $[\ell]$, and $[d]$. $\mathcal{P}$ and $\mathcal{V}$ send (Const, t) to $\mathcal{F}_{\mathsf{ZK}}$ to obtain $[t]$, and increment t. (If either party receives cheating from $\mathcal{F}_{\mathsf{ZK}}$, they abort.) The two parties locally append $([\ell], [t], [op], [d])$ to $\mathcal{L}$.

Check: Let k be the number of accesses performed, i.e., $\mathcal{L}$ contains k tuples.

(1) $\mathcal{P}$ sorts the tuples in $\mathcal{L}$ by their first entry, and secondarily by their second entry, from least to greatest, giving a new list $\ (\ell'_0, t'_0, op'_0, d'_0), \quad)$. For $i \in [k]$, $\mathcal{P}$ sends $(\mathsf{input}, \ell'_i)$, (input, t'_i), (input, op'_i), and (input, d'_i) to $\mathcal{F}_{\mathsf{ZK}}$, which returns $[\ell'_i]$, $[t'_i]$, $[op'_i]$, and $[d'_i]$ to each party. Each party forms the list $\mathcal{L}' = \ ([\ell'_0], [t'_0], [op'_0], [d'_0]), \quad)$.

(2) $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Boolean}, C_{\mathsf{initial}}, [op'_0])$ to $\mathcal{F}_{\mathsf{ZK}}$, where $C_{\mathsf{initial}}(op'_0) = 1$ iff $op'_0 =$ Write. For $i \in [k \quad 1]$, $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Boolean}, C, [\ell'_i], [t'_i], [op'_i], [d'_i], [\ell'_{i\ \ 1}], [t'_{i\ \ 1}], [op'_{i\ \ 1}], [d'_{i\ \ 1}])$ to $\mathcal{F}_{\mathsf{ZK}}$, where $C(\ell'_i, t'_i, op'_i, d'_i, \ell'_{i\ \ 1}, t'_{i\ \ 1}, op'_{i\ \ 1}, d'_{i\ \ 1})$ outputs 1 iff

$$\begin{aligned}&\Big((\ell'_i < \ell'_{i\ \ 1}) \vee \ (\ell'_i = \ell'_{i\ \ 1}) \wedge (t'_i < t'_{i\ \ 1}))\Big) \bigwedge \\ &(\ell'_i \quad \ell'_{i\ \ 1}) \vee (d'_i = d'_{i\ \ 1}) \vee (op'_{i\ \ 1} = \mathsf{Write})) \bigwedge \qquad (1) \\ &(\ell'_i = \ell'_{i\ \ 1}) \vee (op'_{i\ \ 1} = \mathsf{Write}))\end{aligned}$$

Finally, $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Boolean}, C_{\mathsf{final}}, [\ell'_{k\ \ 1}])$ to $\mathcal{F}_{\mathsf{ZK}}$, where $C_{\mathsf{final}}(\ell'_{k\ \ 1}) = 1$ iff $\ell'_{k\ \ 1} < N$. If $\mathcal{F}_{\mathsf{ZK}}$ ever returns 0, then $\mathcal{V}$ outputs cheating and aborts.

(3) For $i \in [k]$, the parties each locally compute $[\![x_i]\!] := \mathsf{Pack}([\ell_i], [t_i], [op_i], [d_i])$ and $[\![x'_i]\!] := \mathsf{Pack}([\ell'_i], [t'_i], [op'_i], [d'_i])$.

(4) $\mathcal{V}$ samples a uniform $r \in \ _{2^k}$ and sends it to $\mathcal{P}$.

(5) Let C' be an arithmetic circuit for which $C'(a_0, \quad , a_{k\ \ 1}, b_0, \quad , b_{k\ \ 1}) = 1$ iff $\prod_{i \in [k]} a_i = \prod_{i \in [k]} b_i$. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Arithmetic}, C', [\![x_0 \quad r]\!], \quad , [\![x_{k\ \ 1} \quad r]\!], [\![x'_0 \quad r]\!], \quad , [\![x'_{k\ \ 1} \quad r]\!])$ to $\mathcal{F}_{\mathsf{ZK}}$. If $\mathcal{F}_{\mathsf{ZK}}$ returns 0, then $\mathcal{V}$ outputs cheating. Otherwise, $\mathcal{V}$ outputs honest.

Figure 5: ZK proof for private read/write array access in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model.

Application to key-value storage. It is worth remarking that ZKarray does not require that the size of the memory be equal to the

Functionality $\mathcal{F}_{\mathsf{ZK\text{-}RAM}}$

Inputs: On receiving (Input, x) from the prover, store x and send $[x]$ to each party.

Constants: On receiving (Const, x) from both parties, store x and send $[x]$ to each party.

RAM program satisfiability: Upon receiving $(\Pi, W, N, [M_0], \ldots, [M_{N-1}])$ from the parties, where $M_i \in \{0,1\}^W$ and Π is a RAM program, compute $y := \Pi(M_0, \ldots, M_{N-1})$ and send y to $\mathcal{V}$. If $\mathcal{V}$ is corrupted, also send to $\mathcal{V}$ the number of steps executed during the computation.

Figure 6: Ideal functionality for RAM-based ZK proofs.

number of values stored in memory. In particular, rather than viewing the memory M as an array of N values $M_0, \ldots, M_{N-1}$, we can instead view it as implementing a key-value store $((\mathsf{key}_0, \mathsf{val}_0), \ldots)$ where the keys are $(\log N)$-bit integers. We discuss this further in Section 5.1.

3.3 ZK Proofs of RAM Programs

One immediate application of ZKarray is for ZK proofs of RAM-based computation. We present an appropriate ideal functionality $\mathcal{F}_{\mathsf{ZK\text{-}RAM}}$ in Figure 6. Our ideal functionality is very flexible in terms of how the memory for the RAM program is initialized: some of the values stored in memory may be known only to the prover, while others may be public values (aka constants) known to both the prover and the verifier. As is typical in the setting of RAM-based computation, we leak the running time of $\Pi(M_0, \ldots, M_{N-1})$ to the verifier; if such leakage is not acceptable, then the running time of Π can always be padded to an upper bound on its value.

Figure 7 gives a protocol realizing $\mathcal{F}_{\mathsf{ZK\text{-}RAM}}$ in the $\mathcal{F}_{\mathsf{ZKarray}}$-hybrid model. The protocol is the natural one: the parties rely on $\mathcal{F}_{\mathsf{ZK\text{-}RAM}}$ for all memory accesses, and thus the only additional step is for $\mathcal{P}$ to convince the verifier at every step that it is executing the next-instruction function Π correctly (cf. Section 2.3). Importantly, our protocol makes only $O(N \quad T)$ calls to $\mathcal{F}_{\mathsf{ZKarray}}$.

Theorem 3. *The protocol in Figure 7 perfectly realizes $\mathcal{F}_{\mathsf{ZK\text{-}RAM}}$ in the $\mathcal{F}_{\mathsf{ZKarray}}$-hybrid model.*

Protocol $\Pi_{\mathsf{ZK\text{-}RAM}}$

Inputs and **Constants** are handled using $\mathcal{F}_{\mathsf{ZKarray}}$ in the natural way.

RAM program satisfiability: $\mathcal{P}$ and $\mathcal{V}$ have Π, W, N, $[M_0], \ldots, [M_{N-1}]$, with $M_i \in \{0,1\}^W$. Let T be the number of steps in the execution of $\Pi(M_0, \ldots, M_{N-1})$.

(1) $\mathcal{P}$ and $\mathcal{V}$ send (Init, W, N, T) to $\mathcal{F}_{\mathsf{ZKarray}}$. They also send $(\mathsf{Const}, \mathsf{Write})$, $(\mathsf{Const}, \mathsf{start})$, and $(\mathsf{Const}, 0^W)$ to $\mathcal{F}_{\mathsf{ZKarray}}$, which returns $[\mathsf{Write}]$, $[\mathsf{start}]$, and $[0^W]$ to each party.

(2) For $i \in [N]$, $\mathcal{P}$ and $\mathcal{V}$ do: (1) send (Const, i) to $\mathcal{F}_{\mathsf{ZKarray}}$, which returns $[i]$ to each party; (2) send $(\mathsf{Access}, [\mathsf{Write}], [i], [M_i])$ to $\mathcal{F}_{\mathsf{ZKarray}}$, which returns $[M_i]$ to each party.

(3) $\mathcal{P}$ runs $\Pi(M_0, \ldots, M_{N-1})$. Specifically, it sets $\mathsf{st} := \mathsf{start}$ and $d := 0^W$, and then until termination does:
- Compute $(op, \ell, d', \mathsf{st}') := \Pi(\mathsf{st}, d)$. Also use $\mathcal{F}_{\mathsf{ZKarray}}$ to generate $[op]$, $[\ell]$, $[d']$, and $[\mathsf{st}']$, and prove to $\mathcal{V}$ that $\Pi(\mathsf{st}, d) = (op, \ell, d', \mathsf{st}')$. If $\mathcal{F}_{\mathsf{ZKarray}}$ returns 0 then $\mathcal{V}$ outputs 0 and aborts; otherwise, both parties set $[\mathsf{st}] := [\mathsf{st}']$. Then:
  - If $op = \mathsf{Stop}$ then $\mathcal{P}$ sends Stop to $\mathcal{V}$ and uses $\mathcal{F}_{\mathsf{ZKarray}}$ to prove $(op = \mathsf{Stop}) \wedge (d' = 1)$. $\mathcal{V}$ outputs what it receives from $\mathcal{F}_{\mathsf{ZKarray}}$ and halts.
  - Otherwise (i.e., if $op \in \{\mathsf{Read}, \mathsf{Write}\}$), $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Access}, [op], [\ell], [d'])$ to $\mathcal{F}_{\mathsf{ZKarray}}$, which returns $[M]$ to each party. $\mathcal{P}$ and $\mathcal{V}$ set $[d] := [M]$ and continue.

Figure 7: RAM-based ZK proofs in the $\mathcal{F}_{\mathsf{ZKarray}}$-hybrid model.

4 A RISC CPU SUPPORTING EFFICIENT ZK PROOFS

Rather than design a dedicated RAM program Π for each specific problem one wants to solve, it is more typical in practice to have a dedicated CPU (which is itself a RAM program) and to then express the problems one wants to solve using assembly-language code to be executed by that CPU. Although there exist other CPUs designed for ZK proofs, they are optimized for specific ZK protocols and will not give optimal performance when using our protocol. (E.g., the TinyRAM CPU [3, 4, 24] is optimized for zk-SNARKs and is thus a poor match for ZK proofs operating on RAM programs expressed as boolean circuits.) To achieve good performance, we design our own CPU and associated instruction set architecture (ISA). We design the ISA to be (1) *expressive*, so that it can run computations efficiently, yet (2) *simple* so that it can be represented by a small boolean circuit. Our ISA contains only 13 instructions but can simulate almost all MIPS-I instructions within 1–3 cycles. Below, we discuss the details of our ISA and its semantics. In Section 4.2, we discuss how to optimize our ZK protocol for the CPU we designed.

4.1 Overview of the CPU

Architecture overview. Our architecture follows the traditional MIPS design and contains a program counter *pc*, a set of 32 registers R, main memory M_d, and program memory M_p. We set the word size to 32 bits, i.e., all entries in R, M_d, and M_p are 32-bit values, and instructions are all 32 bits long. The lengths of M_d and M_p may vary, but we assume in our implementation that they have length at most 2^{32}. The following steps are executed in each cycle:

(1) The next instruction I is read from M_p at the location specified by *pc* (i.e., $M_p[pc]$).

(2) The values of the register at certain locations (that depend on I) are read.

(3) A computation (that depends on I) is done using the register values just read, and the values of other registers as well as the *pc* may be updated.

(4) If I is a memory instruction, then certain locations in memory (depending on I) are read/written.

Note that the above is as a special case of the generic RAM execution described in Section 2.3.

Details of our instruction set. All instructions are 32 bits long. The first five bits contain the opcode (specifying the instruction), and the next three sets of five bits specify *operands*, namely, indices of the registers R that will be utilized to carry out the instruction. The rest of the bits in each instruction are used in different ways. Pictorially:

opcode	tar	src0	src1	imm
5 bits	5 bits	5 bits	5 bits	12 bits

Our instruction set is presented in Table 1. Note that unless explicitly specified (i.e., for PC and JMP), all instructions also increment the program counter by 1. Summarizing:

(1) ADD, SUB, MUL, and XOR read input values from the two source registers (src0 and src1), perform an arithmetic operation, and write the result to the target register (tar).
(2) NLG is similar to the above, except that it also uses three bits in imm to allow for other operations. For example, if imm[0 : 2] are all zero then it corresponds to bit-wise AND, while if imm[0 : 2] are all one then it corresponds to bit-wise OR.
(3) MSK generates a bit mask with a number of ones ranging from 1 to 32 (depending on the value of a source register). For example, if the source register R[src1] is equal to 5, the result would be 00000000 00000000 00000000 00011111. All the bits can further be flipped based on the value of imm[0]. Together with the next instruction, this enables many types of shift operations.
(4) CSF represents the *cyclic shift* instruction. It performs a cyclic right shift on the value R[src0] based on the value stored in R[src1]. For example, if

$$R[\texttt{src0}] = 00001111\ 00000111\ 00000011\ 00000001$$

and R[src1] = 8, then the result would be

$$00000001\ 00001111\ 00000111\ 00000011$$

Note that a cyclic right shift automatically supports cyclic left shift (e.g., shifting right by 31 is the same as shifting left by 1). We can also use this to support a logical shift by performing MSK followed by a bit-wise AND. See Appendix A for further discussion.
(5) PUT loads an immediate value to the target register. To maximize the length of the value, we treat src0‖src1‖imm as one long immediate value.
(6) CMV is a *conditional move* instruction. If (imm[0], imm[1]) = (_, 0), it acts as a NO-OP; if (imm[0], imm[1]) = (1, 1), it is an unconditional move; if (imm[0], imm[1]) = (0, 1), it is a conditional move.
(7) PC puts the value of the program counter in the target register and then increments the program counter as usual.
(8) JMP represents a *conditional jump* instruction. It also has multiple modes of execution: if imm[1] = 1, then this is an unconditional jump; if imm[1] = 0, then this is a conditional jump and the value of imm[0] decides if the condition is on equality or inequality. Note that the condition here is the same as in CMV.

opcode	Semantics
	Arithmetic
ADD	$R[\texttt{tar}] \leftarrow R[\texttt{src0}] \quad R[\texttt{src1}]$
SUB	$R[\texttt{tar}] \leftarrow R[\texttt{src0}] \quad R[\texttt{src1}]$
MUL	$R[\texttt{tar}] \leftarrow R[\texttt{src0}] \cdot R[\texttt{src1}]$
XOR	$R[\texttt{tar}] \leftarrow R[\texttt{src0}] \oplus R[\texttt{src1}]$
NLG	$R[\texttt{tar}] \leftarrow (R[\texttt{src0}] \oplus \texttt{imm}[0]) \wedge (R[\texttt{src1}] \oplus \texttt{imm}[1]) \oplus \texttt{imm}[2]$
MSK	$R[\texttt{tar}] \leftarrow ((1 \ll R[\texttt{src1}]) \quad 1) \oplus \texttt{imm}[0]$
CSF	$R[\texttt{tar}] \leftarrow R[\texttt{src0}] \ggg R[\texttt{src1}]$
PUT	$R[\texttt{tar}] \leftarrow \texttt{src0} \Vert \texttt{src1} \Vert \texttt{imm}$
CMV	$R[\texttt{tar}] \leftarrow \begin{cases} R[\texttt{src1}] & \text{if } (R[\texttt{src0}] \quad 0 \vee \texttt{imm}[0]) \wedge \texttt{imm}[1] \\ R[\texttt{tar}] & \text{otherwise} \end{cases}$
	Control
PC	$R[\texttt{tar}] \leftarrow pc; pc \leftarrow pc \quad 1$
JMP	$pc \leftarrow \begin{cases} R[\texttt{src1}] & \text{if } (R[\texttt{src0}] \quad 0 \oplus \texttt{imm}[0]) \vee \texttt{imm}[1] \\ pc \quad 1 & \text{otherwise} \end{cases}$
	Load/Store
LDW	$R[\texttt{tar}] \leftarrow M[R[\texttt{src0}] \quad \texttt{imm}]$
STW	$M[R[\texttt{src0}] \quad \texttt{imm}] \leftarrow R[\texttt{src1}]$

Table 1: Our CPU instruction set. imm is a 12-bit vector and imm[i] refers to the ith bit in imm. We define the XOR of a vector v and a bit b to be the vector $w = v \oplus b^{|v|}$.

(9) LDW and STW are memory-access instructions. LDW loads a word from memory to a register and STW stores a word from a register to memory. The memory address to read or write is determined by the sum of a source register and the immediate value. LDW and STW reference the same location in memory. Thus, they can both be supported using only one memory access per CPU cycle.

In Appendix A, we show that our instruction set can emulate almost all MIPS-I instructions (with the exception of, e.g., interrupt instructions), most using only 1–3 cycles.

ZK proof for our RISC CPU. The circuit corresponding to a cycle of our CPU accesses the program memory once, the register array three times, and the main memory once; it then computes all instructions in parallel and performs a multiplexer on the results. Below, we describe the details of our ZK protocol (in the $(\mathcal{F}_{\text{RO-ZKarray}}, \mathcal{F}_{\text{ZKarray}})$-hybrid model) when applied to our CPU.

- $\mathcal{P}$ locally simulates the computation to obtain the number of cycles T, and sends T to $\mathcal{V}$.
- Both parties use $\mathcal{F}_{\text{RO-ZKarray}}$ to initialize a read-only array containing the program, and use $\mathcal{F}_{\text{ZKarray}}$ to initialize two read/write arrays for the main memory (namely $\mathcal{F}^{M \quad M}_{\text{ZKarray}}$) and for the registers (namely $\mathcal{F}^{R \quad G}_{\text{ZKarray}}$). Both parties set $[pc] = 0$.
- The parties execute the following T times.
  (1) Both parties use $\mathcal{F}_{\text{RO-ZKarray}}$ to obtain $[I]$ such that I = $M_p[pc]$. Both parties parse $[I]$ as ($[opcode]$, [tar], [src0], [src1], [imm]).
  (2) Both parties use $\mathcal{F}^{R \quad G}_{\text{ZKarray}}$ to read [R[src0]] and [R[src1]] from the register array.
  (3) $\mathcal{P}$ locally computes a bit op and an integer $addr$ such that op = Write only if $opcode$ = STW, and $addr$ = R[src0]   imm.

The parties authenticate these values to get $[op]$ and $[addr]$, and $\mathcal{P}$ proves that they were computed correctly.

(4) $\mathcal{P}$ sends (Access, $[op]$, $[addr]$, $[R[\texttt{src1}]]$) to $\mathcal{F}^{M\quad M}_{\text{ZKarray}}$ and $\mathcal{V}$ sends (Access, $[op]$, $[addr]$, $[R[\texttt{src1}]]$) to $\mathcal{F}^{M\quad M}_{\text{ZKarray}}$, which returns $[p]$ to both parties. For LDW, p is the value $M[addr]$ that would be written to a register in the next step.

(5) The parties, using outputs from previous steps, verify a circuit that computes all instructions and then uses a multiplexer (based on $[opcode]$) to obtain the output. They then use $\mathcal{F}^{R\quad G}_{\text{ZKarray}}$ to write this value to $R[\texttt{tar}]$.

(6) $\mathcal{P}$ updates the program counter (incrementing it unless $opcode$ = JMP and the jump condition holds, in which case pc is set equal to $R[\texttt{src0}]$), and proves that this was done correctly.

4.2 Further Optimizations

We optimized our instruction set and our ZK protocol to allow for several optimizations.

Implementing the MSK instruction. The *one-hot encoding* of a $\log n$-bit (non-negative) integer x is the n-bit vector $V \in \{0,1\}^n$ that is zero everywhere except that $V[x] = 1$. Let OneHot_n be the function that maps $x \in \{0,1\}^{\log n}$ to its one-hot encoding. We give a recursive construction of a boolean circuit for OneHot_n that uses $n \quad 1$ AND gates. First, we have $\text{OneHot}_2(x) = x \| (x \oplus 1)$. Furthermore,

$$\text{OneHot}_n(x) = \text{ if } (x[\log n \quad 1]) \text{ then } r \| 0^{n/2} \text{ else } 0^{n/2} \| r,$$

where $r = \text{OneHot}_{n/2}(x[0 : \log n \quad 2])$. Given r, the above can be computed using $n/2$ AND gates, implying that OneHot_n can be computed using n-1 AND gates.

We can use this idea to implement the MSK instruction using $n \quad 1$ AND gates as follows: (1) compute the one-hot encoding of the lowest 5 bits of $R[\texttt{src1}]$ to obtain $e \in \{0,1\}^{32}$, then (2) compute the mask $r \in \{0,1\}^{32}$ by setting $r[31] = e[31]$ and for $i < 31$, $r[i] = r[i \quad 1] \oplus e[i]$.

Efficient ZK for CSF. We are not aware of a boolean circuit implementing an n-bit cyclic shift using fewer than $n \log n$ AND gates. Here, we show an approach suitable for our ZK protocol that does not directly use a boolean circuit. The key observation is that the cyclic shift operation can be verified using n inner products; since inner products are degree-two polynomials, they can be efficiently verified using QuickSilver [29].

Specifically, suppose the parties hold authenticated values $[u]$, $[v]$ where $u \in \{0,1\}^n, v \in \{0,1\}^{\log n}$; their goal is to compute $[u \ggg v]$ without allowing $\mathcal{P}$ to cheat. Our approach works as follows:

(1) The parties compute $[V]$, where $V \in \{0,1\}^n$ is the one-hot encoding of v; as discussed above, this can be computed using $n \quad 1$ AND gates.

(2) $\mathcal{P}$ locally computes $U := u \ggg v$, and generates $[U]$. Then, $\mathcal{P}$ proves that $\langle [u], [V] \qquad i \rangle = [U[i]]$ for all $i \in [0, n \quad 1]$.

In the context of our CPU, the one-hot encoding of $v = R[\texttt{src1}]$ is anyway computed as part of the MSK instruction, so the additional cost of supporting CSF is small.

Instruction multiplexing. After evaluating all the instructions, our CPU circuit runs an 11-way multiplexer to determine the (possibly) updated value of $R[\texttt{tar}]$. (Only 11 of our instructions modify $R[\texttt{tar}]$.) We verify its evaluation as follows. Let $s_0, \quad , s_{10} \in \{0,1\}^{32}$ be the results of the 11 relevant instructions, and let $opcode$ denote the opcode of the actual instruction in this step (the parties already hold authenticated values for all of them); our goal is to obtain an authenticated sharing of s_{opcode}. We first convert $opcode$ into its one-hot encoding v. Letting S be the matrix whose ith column is s_i, we then have $S \cdot v = s_{opcode}$. Verifying this matrix-vector multiplication involves evaluating 32 inner products, and so can be done efficiently.

Delaying expensive instructions. Certain instructions in our instruction set (namely MUL, CSF, LDW, and STW) incur a substantially higher computational cost than others. Inspired by [26], we improve the performance of our CPU by implementing a simple scheme that reduces the overhead of those expensive instructions . The basic idea is that expensive instructions are no longer carried out in every CPU cycle of the CPU, but are instead only evaluated every nth cycle (where n is a public parameter called the *delay*). If an expensive instruction is supposed to be executed "off cycle," its evaluation is delayed by up to $n \quad 1$ cycles.

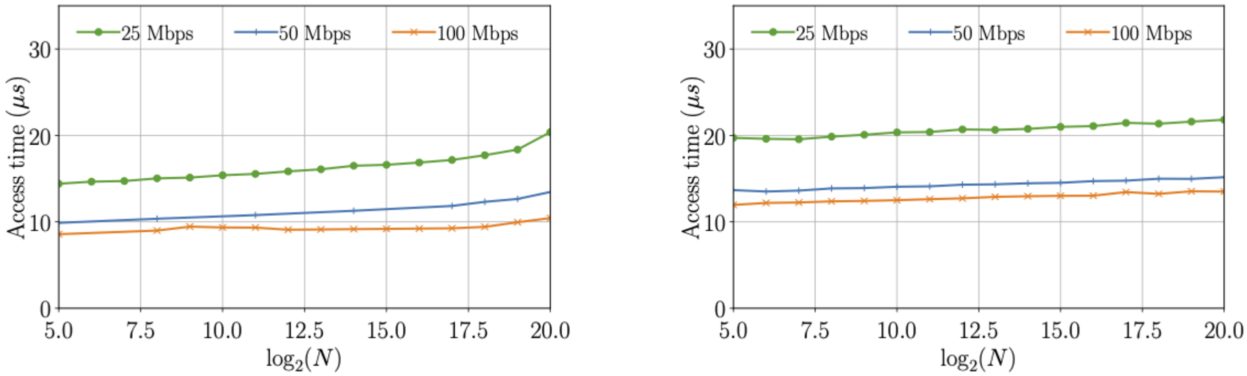
5 EVALUATION

We evaluate the performance of our protocols and compare them with relevant prior work. We incorporated the ZK proof for polynomials [29] in both protocols, as discussed in Section 3.1 and 3.2. All our experiments were performed by running our protocols between two AWS EC2 m5 2xlarge machines. We conduct experiments on RO-ZKarray and ZKarray to demonstrate their computational/communication efficiency. Then, we benchmark the performance of our CPU based on RO-ZKarray and ZKarray. We report microbenchmarks of all protocols to show the performance of the major components. We will make our code available at EMP-toolkit [25].

5.1 Performance of RO-ZKarray and ZKarray

We first explore the performance of our protocols for private array access. Here, all performance numbers are averaged over one million accesses and we fix $W = 32$.

Memory-access time. We measure the performance of our memory-access protocols as a function of the memory size N in different network settings. As shown in Figure 8, for both RO-ZKarray and ZKarray the average time per memory access grows very slowly with N. Even when the network is throttled to 25 Mbps, the average access time of RO-ZKarray increases only slightly from roughly 14 4 to 20 4 microseconds as N increases from 2^5 to 2^{20}. Both RO-ZKarray and ZKarray are much more efficient than a linear scan over the memory. A linear scan would need at least $N \cdot W$ AND gates; giving a ZK proof on the corresponding circuit would be slower than our protocol for all N that we tested. For example, when $N = 2^5, W = 32$, applying the best circuit-based ZK protocol to a linear scan would take 70–100 μs, which is more than 7× slower than our protocol.



a) RO-ZKarray.

b) ZKarray.

Figure 8: Performance of our ZK protocols for private array access.

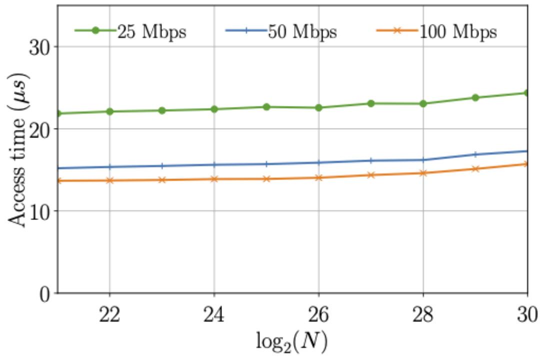


Figure 9: Performance of ZKarray **as a key-value store.**

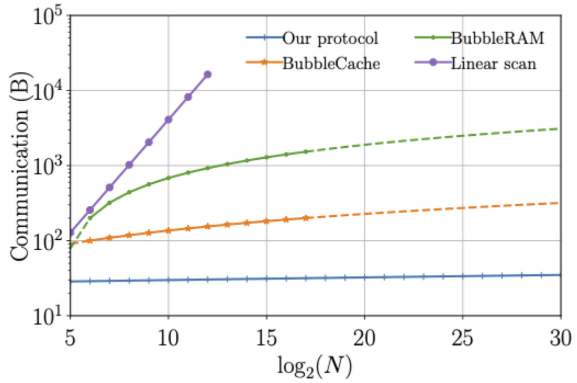


Figure 10: Communication overhead of RAM-based ZK protocols. Numbers for prior work are taken from [18]. Note that the y axis is on a log scale.

As discussed in Section 3.2, ZKarray can also be used to implement a key-value store, where keys take the place of addresses. The complexity of ZKarray depends mainly on N T, the number of items stored, while expanding the key size does not significantly affect the overall performance. We also explore the performance in this case. In Figure 9, we show the performance when the key size ranges from 21 bits to 30 bits while the number of items stored is fixed to one million.

Comparison to BubbleRAM and BubbleCache. In Figure 10, we compare the communication complexity of ZKarray with that of BubbleRAM and BubbleCache [16, 18], which both offer similar functionality. (Note, however, that BubbleCache allows for input-dependent cache misses, and the reported performance of BubbleCache considers a cache miss rate as high as 10%.) We also include the communication cost of linear scan as a baseline. ZKarray not only uses the least communication of any of these protocols, but also scales best with increasing N. As N ranges from 2^{20} to 2^{30}, the per-access communication of BubbleRam (resp., BubbleCache) goes from 2025 (resp., 202) bytes to 4556 (resp., 303) bytes; for ZKarray, the communication goes from 33 to 35 bytes, an improvement of 62–130× (resp., 6–8×).

Microbenchmarks. We benchmark the performance of the main components of our protocols for a memory access in Table 2, for network bandwidths of 25, 50, and 100 Mbps. For RO-ZKarray, we set $N = 2^{15}$. For ZKarray, we set $N = 2^{15}, T = 2^{25}$. "Access" refers to the step where $\mathcal{P}$ authenticates values. "Check consistency" refers to the step where $\mathcal{P}$ verifies an adjacent tuple in the list $\mathcal{L}'$. "Check set equality" refers to verifying equality of $\mathcal{L}$ and $\mathcal{L}'$, and is amortized over the size of these lists.

	RO-ZKarray			ZKarray		
Bandwidth (Mbps)	25	50	100	25	50	100
Access	2.1	1.4	1.2	2.2	1.6	1.6
Check consistency	9.5	6.6	5.8	12.8	9.2	8.6
Check set equality	5.5	3.7	3.0	5.1	3.4	2.8
Total	17.8	12.4	10.8	21.7	15.7	14.4

Table 2: Microbenchmarks for RO-ZKarray **and** ZKarray **with varying bandwidth.** Times are in microseconds.

5.2 ZK Proofs for Our CPU

Here we benchmark the performance of our ZK protocol when applied to our CPU. Unless specified otherwise, the network bandwidth is 100 Mbps.

CPU Performance. We implemented our CPU and evaluated it with varying configurations of the program size and memory capacity. Results for per-cycle execution time are shown in Figure 11. The execution time increases very slowly as the program size grows, and hardly increases at all as a function of the memory capacity. As with execution time, communication per cycle increases only slightly as the program size grows. With a main memory of size 2^{24}, each cycle requires communication of roughly 320 bytes. This is more than a 12 5× improvement compared to BubbleCache, which communicates about 4000 bytes even with a memory of size 2^{17}.

log(program size)	Delay			
	1	2	4	8
10	311	226	183	161
16	315	230	187	166
20	338	253	210	189

Table 3: Effect of delay on communication. Numbers reflect bytes of communication per CPU cycle. All experiments used main memory of size 2^{24}.

We also experimented with different settings for the CPU instruction delay n. Recall this means we only execute certain expensive instructions every nth cycle (see Section 4.2). Results are in Table 3. We see that increasing the delay substantially reduces the average per-cycle execution time, though of course the overall impact of the delay will depend on the program being executed and, in particular, how often that program relies on expensive instructions.

Microbenchmarks. When executing our CPU, we use RO-ZKarray to access the program memory, and use ZKarray to access the registers and the main memory. We report the various access times, under network bandwidths ranging from 25 to 500 Mbps, in Table 4. We observe that computation becomes the bottleneck once the bandwidth is 100 Mbps (since the performance does not improve as the bandwidth increases further). In all cases we use a 32-bit word size. We use ZKarray with $T = 2^{14}$ to emulate access to the registers. (This means that 2^{14} accesses to the registers are supported; once the upper bound is reached, we must refresh the structure). For the main memory, we set $N = 2^{20}$ bits and $T = 2^{25}$.

5.3 Customized RAM-based Protocols

While our CPU can be used for general-purpose RAM computation, in some cases better efficiency is possible using a customized protocol. We explore two such examples here. All experiments described in this section were run over a 100 Mbps network.

ZK proof of regular expression matching. In this application we assume the prover has generated a commitment com to a string $s \in \Sigma^n$, and wants to prove to the verifier that s matches a public regular expression R. That is, the prover wants to prove knowledge of s, decom such that

Memory	Index size (bits)	Bandwidth			
		25 Mbps	50 Mbps	100 Mbps	500 Mbps
Program	15	16.52	11.83	10.05	10.04
Register	5	15.15	9.43	8.26	8.24
Main	20	21.61	14.88	13.38	13.36

Table 4: Memory-access time for different memory components. The numbers represent the average access time, in microseconds.

$$\mathsf{Open}(\mathsf{com}, s, \mathsf{decom}) = 1 \wedge \mathsf{RegexMatch}(s, R) = \mathsf{Accept}$$

To do this, we first convert the regular expression to a DFA with S states, and then represent the transitions of the DFA using a matrix of size $|\Sigma| \times S$, which we flatten to an array M. See Figure 12.

Taking $\Sigma = \{0, \quad , |\Sigma|\}$ and assuming $\mathcal{V}$ knows n, our protocol then proceeds as follows:

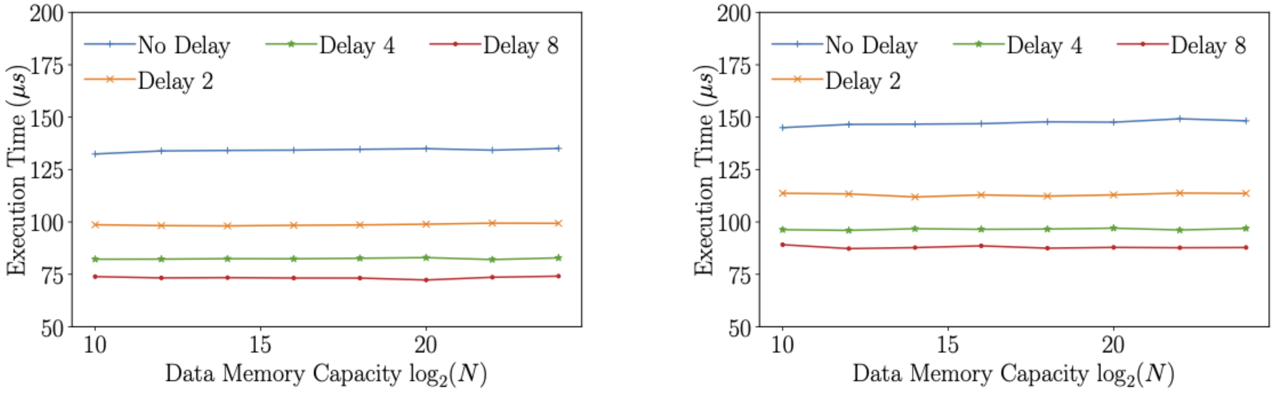
(1) The parties initialize $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$ to contain M.
(2) $\mathcal{P}$ generates $\{[s_i]\}_{i \in [n]}$ and [decom] and proves that Open(com, $[s_0]$, $\quad$, $[s_{n \quad 1}]$, [decom]) = 1
(3) The parties set $cur := 0$ and generate $[cur]$. Then for $i \in [n]$ they do:
    (a) Compute $[\ell] := [cur] \cdot |\Sigma| \quad [s_i]$.
    (b) Use $\mathcal{F}_{\mathsf{RO\text{-}ZKarray}}$ to read M_ℓ and set $[cur] := [M_\ell]$.
(4) Prove that $[cur] = |S| \quad 1$.

The cost of the protocol is dominated by n read-only memory accesses. We implement this protocol using SHA-256 for the commitment. With $n = 128$, $|\Sigma| = 4$, and $S = 20$, the total time to execute the protocol was 2.7 ms, and increasing S to 2,000 only increased the running time to 3.1 ms. In both cases, the time to verify the commitment was the same (1.6 ms), and only the time required to run the DFA changed.

Proving the preimage of a memory-hard hash function. In our second application, we have $\mathcal{P}$ prove knowledge of a preimage of scrypt, a memory-hard hash function that was designed specifically so that its evaluation requires many random accesses to memory. The core of scrypt is ROMix, which involves $O(N)$ evaluations of the 8-round Salsa20 hash function (where N is a hardness parameter) and a set of read-only accesses on values that are computed using Salsa20. Here we focus on the performance of computing ZK proofs for ROMix; see Table 5. We use the default parameters ($r = 8, p = 1$) and let the hardness N (which equals the length of the memory array) range from 4 to 10. ROMix uses a 1024-bit word size. Since our current implementation only supports a maximum word size of 64 bits, we replace a single memory access using a 1024-bit word size with 16 memory accesses using a 64-bit word size. By natively supporting a larger word size it should be possible to further improve performance.

ACKNOWLEDGEMENTS

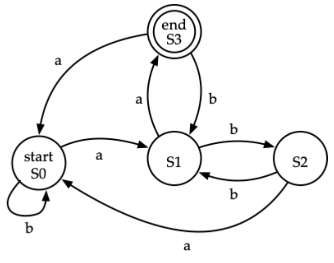
This material is based upon work supported in part by DARPA under Contract Nos. HR001120C0087 and HR00112020025. The



a) Program size 2^{10}

b) Program size 2^{20}

Figure 11: ZK proofs for execution of our CPU. The numbers reported are the execution time per CPU cycle, averaged over 2^{20} cycles, with no delay.



a) DFA for the regular expression `a ab|bb)*a`**.**

State	a	b
0	1	0
1	3	2
2	0	1
3	0	1

b) Matrix corresponding to the DFA.

Figure 12: Representing a DFA as an array. In this example, the final array representing the DFA is $M = \{1, 0, 3, 2, 0, 1, 0, 1\}$.

N	4	16	64	256	1024
Prove Salsa20 (s)	0.07	0.29	1.17	4.69	18.79
ZK array access (s)	0 01	0 036	0 14	0 57	2 34
Total (s)	0.08	0.326	1.31	5.26	21.13

Table 5: ZK proofs of scrypt.

views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Work of Jonathan Katz was also supported by NSF award 1563722. Work of Rafail Ostrovsky was also supported by NSF award 2001096, US-Israel BSF grant 2015782, a Google Faculty Award, a JP Morgan Faculty Award, an IBM Faculty Research Award, a Xerox Faculty Research Award, an OKAWA Foundation Research Award, a B. John Garrick Foundation Award, a Teradata Research Award, a Lockheed-Martin Research Award, and the Sunday Group. Work of Xiao Wang was also supported by NSF award 2016240, and research awards from Facebook and PlatON Network. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

REFERENCES

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM Conf. on Computer and Communications Security (CCS) 2017.* ACM Press, 2087–2104.

[2] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions *(LNCS).* Springer, 92–122.

[3] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Advances in Cryptology—Crypto 2013, Part II (LNCS, Vol. 8043).* Springer, 90–108.

[4] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security Symposium 2014.* USENIX Association, 781–796.

[5] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. 2020. Public-Coin Zero-Knowledge Arguments with (almost) Minimal Time and Space Overheads. In *17th Theory of Cryptography Conference—TCC 2020 (LNCS).* Springer, 168–197.

[6] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. 2021. Time- and Space-Efficient Arguments from Groups of Unknown Order *(LNCS).* Springer, 123–152.

[7] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. 2016. Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting. In *Advances in Cryptology—Eurocrypt 2016, Part II (LNCS, Vol. 9666).* Springer, 327–357.

[8] Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. 2018. LWE Without Modular Reduction and Improved Side-Channel Attacks Against BLISS. In *Advances in Cryptology—Asiacrypt 2018, Part I (LNCS).* Springer, 494–524.

[9] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE Symp. on Security Privacy 2018.* IEEE, 315–334.

[10] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2021. Line-Point Zero Knowledge and Its Applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021) (Leibniz International Proceedings in Informatics (LIPIcs)).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany.

[11] Jack Doerner, David Evans, and Abhi Shelat. 2016. Secure Stable Matching at Scale. In *ACM Conf. on Computer and Communications Security (CCS) 2016.* ACM Press, 1602–1613.

[12] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. 2015. Privacy-Free Garbled Circuits with Applications to Efficient Zero-Knowledge. In *Advances in Cryptology—Eurocrypt 2015, Part II (LNCS, Vol. 9057).* Springer, 191–219.

[13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Advances in Cryptology—Eurocrypt 2013 (LNCS).* Springer, 626–645.

[14] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure two-party computation in sublinear (amortized) time. In *ACM Conf. on Computer and Communications Security (CCS) 2012.* ACM Press, 513–524.

[15] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology—Eurocrypt 2016, Part II (LNCS, Vol. 9666).* Springer,

305–326.

[16] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz Zero-Knowledge Processor with BubbleRAM. In *ACM Conf. on Computer and Communications Security (CCS) 2020.* ACM Press, 2055–2074.

[17] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *Advances in Cryptology—Eurocrypt 2020, Part III (LNCS, Vol. 12107).* Springer, 569–598.

[18] David Heath, Yibin Yang, David Devecsery, Vladimir Kolesnikov, and Marco Guarnieri. 2021. Zero Knowledge for Everything and Everyone: Fast ZK Processor with Cached ORAM for ANSI C Programs. In *IEEE Symp. on Security Privacy.* IEEE.

[19] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. 2015. Efficient Zero-Knowledge Proofs of Non-algebraic Statements with Sublinear Amortized Cost. In *Advances in Cryptology—Crypto 2015, Part II (LNCS, Vol. 9216).* Springer, 150–169.

[20] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM Conf. on Computer and Communications Security (CCS) 2013.* ACM Press, 955–966.

[21] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. 2018. Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures. In *ACM Conf. on Computer and Communications Security (CCS) 2018.* ACM Press, 525–537.

[22] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. 2017. Sublinear Zero-Knowledge Arguments for RAM Programs. In *Advances in Cryptology—Eurocrypt 2017, Part I (LNCS, Vol. 10210).* Springer, 501–531.

[23] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *Advances in Cryptology—Crypto 2020, Part III (LNCS).* Springer, 704–737.

[24] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium.* The Internet Society.

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

[26] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. 2016. Secure Computation of MIPS Machine Code. In *ESORICS 2016 (LNCS).* Springer, 99–117.

[27] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. In *IEEE Symp. on Security Privacy 2021.* IEEE.

[28] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Advances in Cryptology—Crypto 2019, Part III (LNCS, Vol. 11694).* Springer, 733–764.

[29] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM Conf. on Computer and Communications Security (CCS) 2021.* ACM Press.

A SIMULATION OF MIPS-I INSTRUCTIONS

Below we provide explanations of how the MIPS instructions listed in Table 6 can be simulated using our CPU. (When simulation is straightforward we are brief, but for some of the more complex cases we provide details of the simulation.)

- Arithmetic and logical Instructions: Most of these are trivially supported by our arithmetic and logical operations. One notable exception is div. However, $\mathcal{P}$ can give a ZK proof that $c = a/b$ by proving that $c \cdot b = a$ using the MUL command. MIPS operations that act on immediate values can be simulated by two of our instructions by first performing a PUT and then performing the desired arithmetic or logical operation. Of note, the MIPS mult command fills two registers to allow for the computation of 64-bit products. Our CPU does not support this operation directly, though it can be simulated by breaking down the multiplicands and performing multiple MUL operations. Logical shift operations are slightly more complex: first we use CSF to rotate the bits of the register, followed by MSK and NLG to perform bitwise AND in order to zero out the appropriate bits. Arithmetic shift operations add an additional layer of complexity, since we must determine whether the original value is positive or negative in order to preserve the sign. Suppose we are given registers $a and $n containing values a and n, where a is the value to be shifted and n is the shift amount. Simulation of srav can then be performed as follows:
  (1) Use PUT to put the value 1 in a register;
  (2) Use PUT to put the value 0 in a register;
  (3) Use MSK to put the string $1||0^{31}$ in register $x (using the register previously set to 1);
  (4) Use MSK to put the string $1^{n}||0^{32 \quad n}$ in register $y (using the given register containing n);
  (5) Use MSK to put the string $0^{n}||1^{32 \quad n}$ in register $p (using the given register containing n);
  (6) Use MSK to put the string 0^{32} in register $z (using the register previously set to 0);
  (7) Use NLG to perform a bitwise AND on registers $a and $x, saving the result in register $w;
  (8) Use CMV to perform the operation $z ← $y if $w   0, else do nothing;
  (9) Use CSF to rotate a by n bits, saving the result in register $r;
  (10) Use NLG to perform a bitwise AND on registers $r and $p, saving the result in register $r;
  (11) Use NLG to compute $z OR $r as the final result.

  The first six steps initialize necessary registers. Step 7 checks whether a is negative, and step 8 conditionally moves the string $1^{n}||0^{32 \quad n}$ into a register if a is negative. Step 9 rotates a by the appropriate number of bits, and step 10 zeros out the bits that were shifted in on the right. Finally, step 11 fills the zeroed bits with 1s if a was negative, and leaves them as 0s otherwise. The sra instruction can similarly be simulated using one additional PUT instruction.
- Constant-manipulating instructions: The MIPS lhi instruction loads an immediate value into the upper half of a register. Given a register value a and an immediate value imm, lhi can be simulated as follows:
  (1) Use PUT to load imm into a register;
  (2) Use PUT to load the value 16 into a register;
  (3) Use MSK to load the string $0^{16}||1^{16}$ into a register (using the register previously set to 16);
  (4) Use NLG to compute the bitwise AND of the previous two values and load the first 16 bits of imm into a register;
  (5) Use CSF to rotate those bits of imm by 16;
  (6) Use NLG to compute the bitwise AND of a and $0^{16}||1^{16}$ and load the first 16 bits of a into a register;
  (7) Use XOR to combine the bits of imm with the bits of a.

  Simulating llo follows similarly.
- Comparison instructions: For two integers a, b, the prover can show that $a \leq b$ holds as follows:
  (1) Use SUB to compute $b \quad a$ and store the result in register $x;
  (2) Use PUT to put the value 1 in a register;
  (3) Use MSK to get the string $1||0^{31}$ and store it in register $y (using the register previously set to 1);
  (4) Use NLG to take the bitwise AND of $x and $y.

  If the resulting string is 0^{32} then $b \quad a \geq 0$, and thus $a \leq b$. If it is instead $1||0^{31}$, then $a > b$. The CSF command can additionally be

MIPS instruction	Semantics	CPU cycles to simulate
add $d, $s, $t	$d = $s + $t	1
addi $t, $s, i	$t = $s + i	2
and $d, $s, $t	$d = $s & $t	1
andi $t, $s, i	$t = $s & i	2
div $s, $t	lo = $s / $t; hi = $s % $t	1
mult $s, $t	hi:lo = $s * $t	1
nor $d, $s, $t	$d = ∼($s | $t)	1
or $d, $s, $t	$d = $s | $t	1
ori $t, $s, i	$t = $s | i	2
sll $d, $t, a	$d = $t ≪ a (logical)	4
sllv $d, $t, $s	$d = $t ≪ $s (logical)	3
sra $d, $t, a	$d = $t ≫ a (arithmetic)	11
srav $d, $t, $s	$d = $t ≫ $s (arithmetic)	12
srl $d, $t, a	$d = $t ≫ a (logical)	4
srlv $d, $t, $s	$d = $t ≫ $s (logical)	3
sub $d, $s, $t	$d = $s - $t	1
xor $d, $s, $t	$d = $s ⊕ $t	1
xori $d, $s, i	$d = $s ⊕ i	2
lhi $t, i	HH($t) = i	7
llo $t, i	LH($t) = i	7
slt $d, $s, $t	$d = ($s < $t)	5
slti $t, $s, i	$t = ($s < i)	6
beq $s, $t, label	if ($s == $t) pc += i	4
bgtz $s, label	if ($s > 0) pc += i	6
blez $s, label	if ($s ≤ 0) pc += i	6
bne $s, $t, label	if ($s == $t) pc += i	4
j label	pc += i	2
jal label	$31 = pc; pc += i	3
jalr $s	$31 = pc; pc = $s	2
jr $s	pc = $s	1
lw $t, i($s)	$t = MEM [$s + i]	1
sw $t, i($s)	MEM [$s + i] = $t	1
mfhi $d	$d = hi	1
mflo $d	$d = lo	1
mthi $s	hi = $s	1
mtlo $s	lo = $s	1

Table 6: Simulation of MIPS instructions by our CPU. Here we list supported MIPS operations and the number of cycles in which our CPU can simulate them. Unsigned MIPS operations are omitted, since our CPU always uses signed operations. Since our CPU instruction set is word-based, while MIPS is byte-based, some MIPS byte-manipulation instructions are omitted or modified for clarity. We note that while our JMP instruction does not directly support labels, it performs functionally equivalent behavior by jumping to the index of a particular instruction in the program. Allowing for these omissions and modifications, our CPU is able to simulate the entire MIPS instruction set other than Exception and Interrupt instructions. See text for details.

used to rotate the 31^{st} bit to the 0^{th} position, for a final result of 1 if $a > b$ and 0 otherwise.

- Branch instructions: The MIPS beq and bne instructions are straightforwardly simulated using a PUT followed by JMP. The bgtz and blez instructions can be simulated by first using MSK and NLG to isolate the most significant bit (as shown above in the simulation of comparison instructions) and then using PUT followed by JMP.
- Jump instructions: The j instruction is also straightforwardly simulated using PUT and then JMP. The jr instruction is exactly replicated by JMP with the correct parameters in imm. Further, jalr is simulated by PC followed by JMP, and similarly jal is simulated by first performing a PUT and then doing the same.
- Load and store instructions: These can be simulated straightforwardly with our LDW and STW instructions.
- Data-movement instructions: These are straightforwardly simulated using CMV.
- Exception and interrupt instructions: We do not currently support these instructions, and we envision that the prover would ensure that the executed code does not contain them.